# HEAP OPTIMIZATION FOR GO SYSTEMS

Nishant Roy, Pinterest

Apr 2023

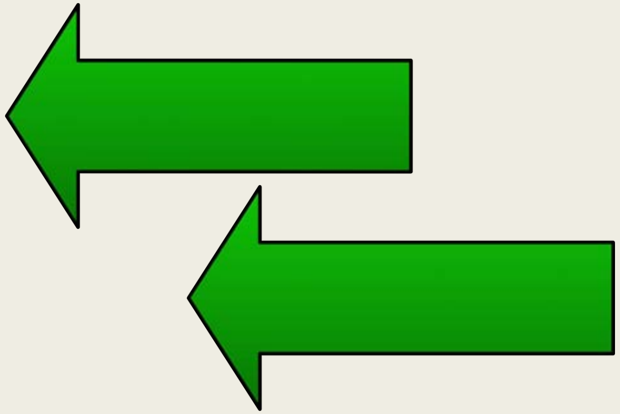
# About Me

- Engineering Manager @ Pinterest Ads Serving Platform

- Responsible for performance and reliability of ad delivery infrastructure

# How does memory management work in Go?

AUTOMATED



CONCURRENT

# How does garbage collection impact performance?

LIMITS CPU USAGE

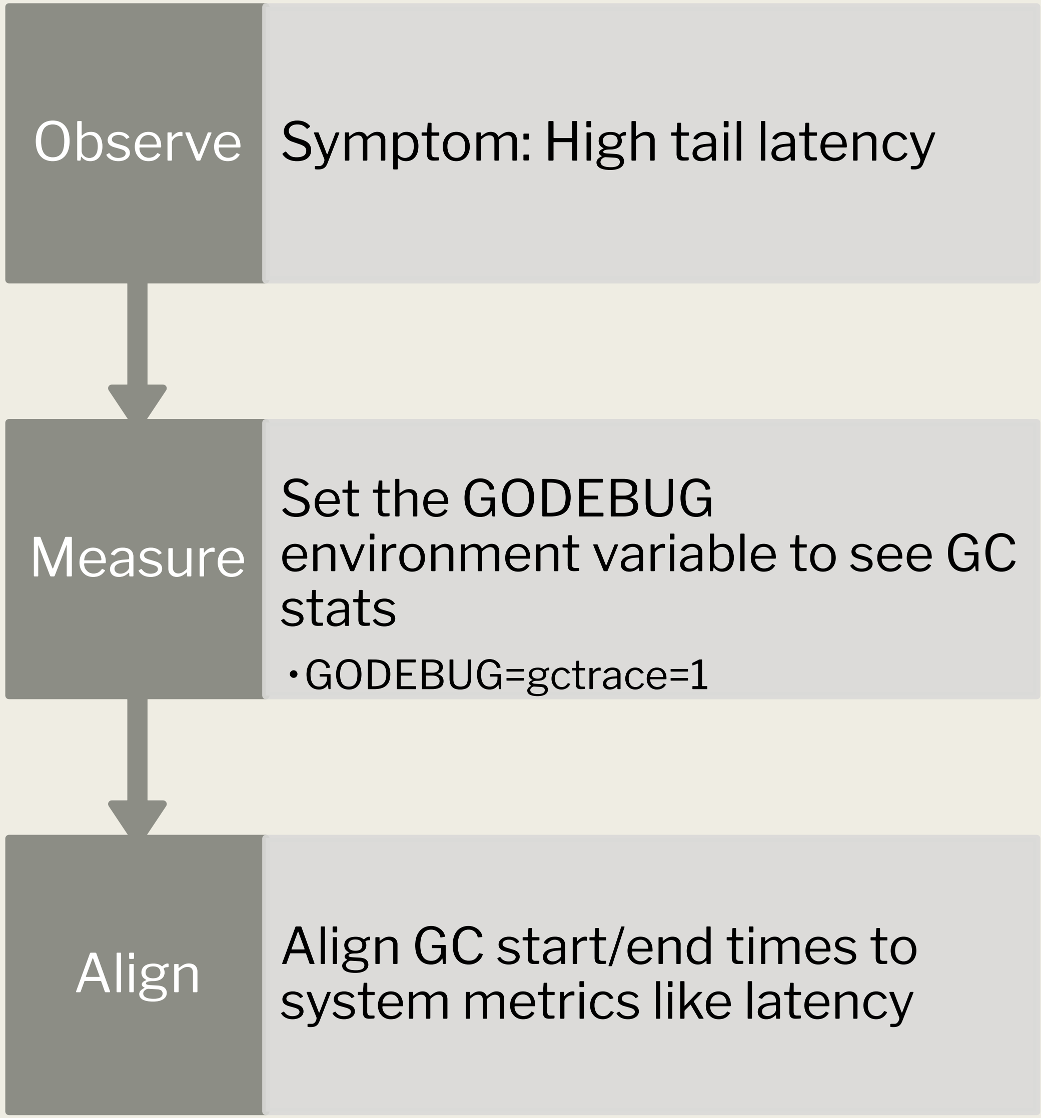
STEALS RESOURCES

# What causes GC to run slower?

SCANNING THE HEAP

NUMBER OF HEAP OBJECTS

# How to determine if GC is the problem?



Observe

Symptom: High tail latency

Measure

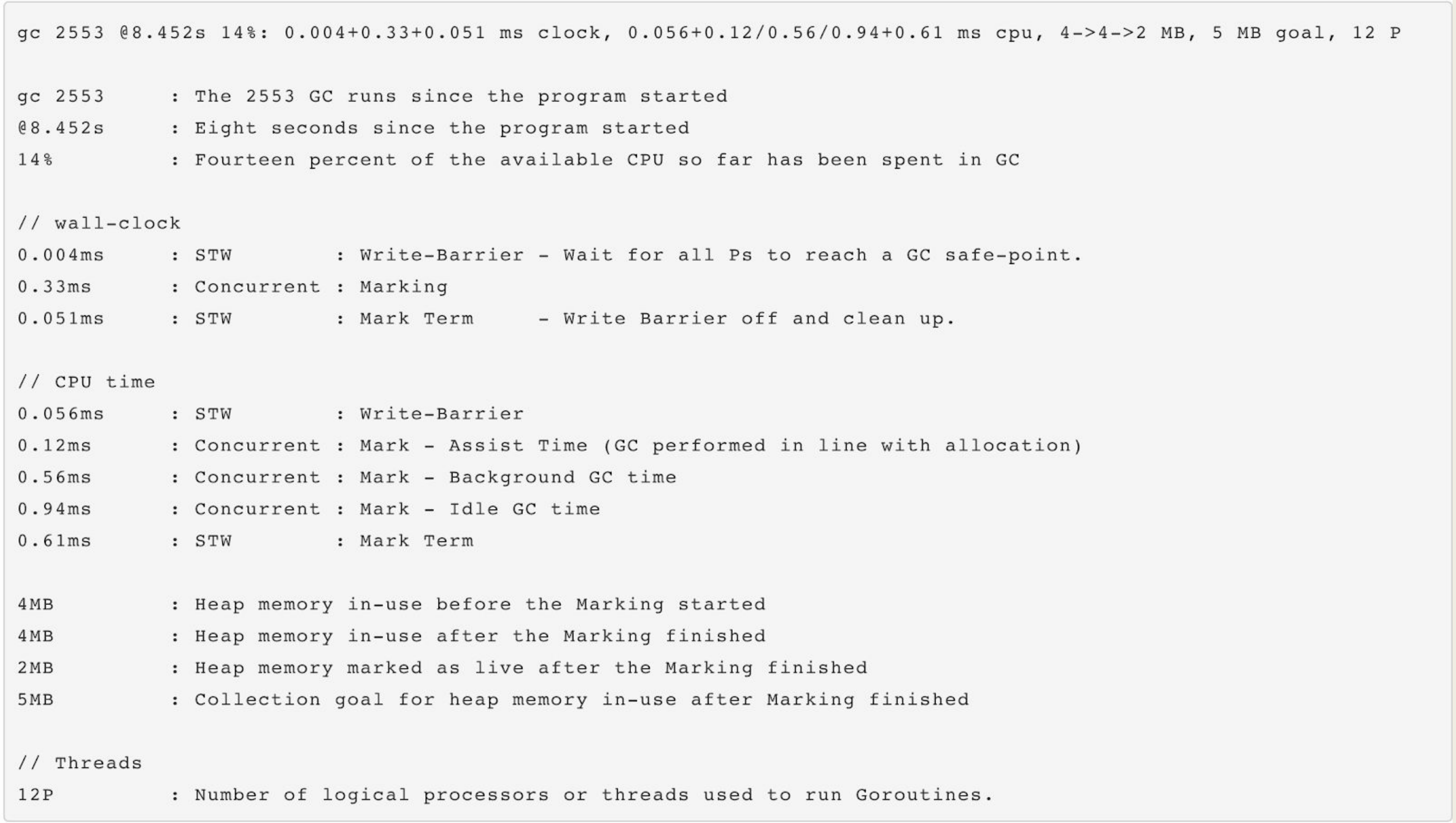
Set the GODEBUG environment variable to see GC stats

- GODEBUG=gctrace=1

Align

Align GC start/end times to system metrics like latency

# How to read **gctrace** output?



```
gc 2553 @8.452s 14%: 0.004+0.33+0.051 ms clock, 0.056+0.12/0.56/0.94+0.61 ms cpu, 4->4->2 MB, 5 MB goal, 12 P

gc 2553     : The 2553 GC runs since the program started
@8.452s     : Eight seconds since the program started
14%         : Fourteen percent of the available CPU so far has been spent in GC

// wall-clock
0.004ms     : STW        : Write-Barrier - Wait for all Ps to reach a GC safe-point.
0.33ms      : Concurrent : Marking
0.051ms     : STW        : Mark Term     - Write Barrier off and clean up.

// CPU time
0.056ms     : STW        : Write-Barrier
0.12ms      : Concurrent : Mark - Assist Time (GC performed in line with allocation)
0.56ms      : Concurrent : Mark - Background GC time
0.94ms      : Concurrent : Mark - Idle GC time
0.61ms      : STW        : Mark Term

4MB         : Heap memory in-use before the Marking started
4MB         : Heap memory in-use after the Marking finished
2MB         : Heap memory marked as live after the Marking finished
5MB         : Collection goal for heap memory in-use after Marking finished

// Threads
12P         : Number of logical processors or threads used to run Goroutines.
```

# How to profile heap usage?

Use built-in tools to study heap usage

**runtime.MemStats** – Memory allocator statistics

**pprof** – System profile visualizer

# MemStats

```
// Number of allocated heap objects.
HeapObjects uint64
```

```
// Bytes of allocated heap objects.
HeapAlloc uint64
```

```
// Total bytes of memory obtained from the OS.
HeapSys uint64
```

Source:
*Go docs*

# MemStats

```go
func main() {
    PrintMemstats()

    var arr [][]int

    for i := 0; i<4; i++ {
        vec := make([]int, 0, 25000)
        overall = append(arr, vec)
    }

    overall = nil
    PrintMemstats()

    runtime.GC()
    PrintMemstats()
}
```

```go
func PrintMemstats() {
        var m runtime.MemStats
        runtime.ReadMemStats(&m)
        fmt.Printf("HeapAlloc = %v", (m.HeapAlloc))
        fmt.Printf("\tHeapObjects = %v", (m.HeapObjects))
        fmt.Printf("\tHeapSys = %v", (m.Sys))
        fmt.Printf("\tNumGC = %v\n", m.NumGC)

}
```

Source:
*Golang Code*

# MemStats

```
$ go run main.go

HeapAlloc = 106392    HeapObjects = 133    HeapSys = 69928960    NumGC = 0
HeapAlloc = 312528    HeapObjects = 142    HeapSys = 69928960    NumGC = 0
HeapAlloc = 517928    HeapObjects = 150    HeapSys = 69928960    NumGC = 0
HeapAlloc = 723112    HeapObjects = 158    HeapSys = 71631096    NumGC = 0
HeapAlloc = 928400    HeapObjects = 164    HeapSys = 71631096    NumGC = 0
HeapAlloc = 928736    HeapObjects = 170    HeapSys = 71631096    NumGC = 0
HeapAlloc = 112032    HeapObjects = 153    HeapSys = 71958776    NumGC = 1
```

# pprof

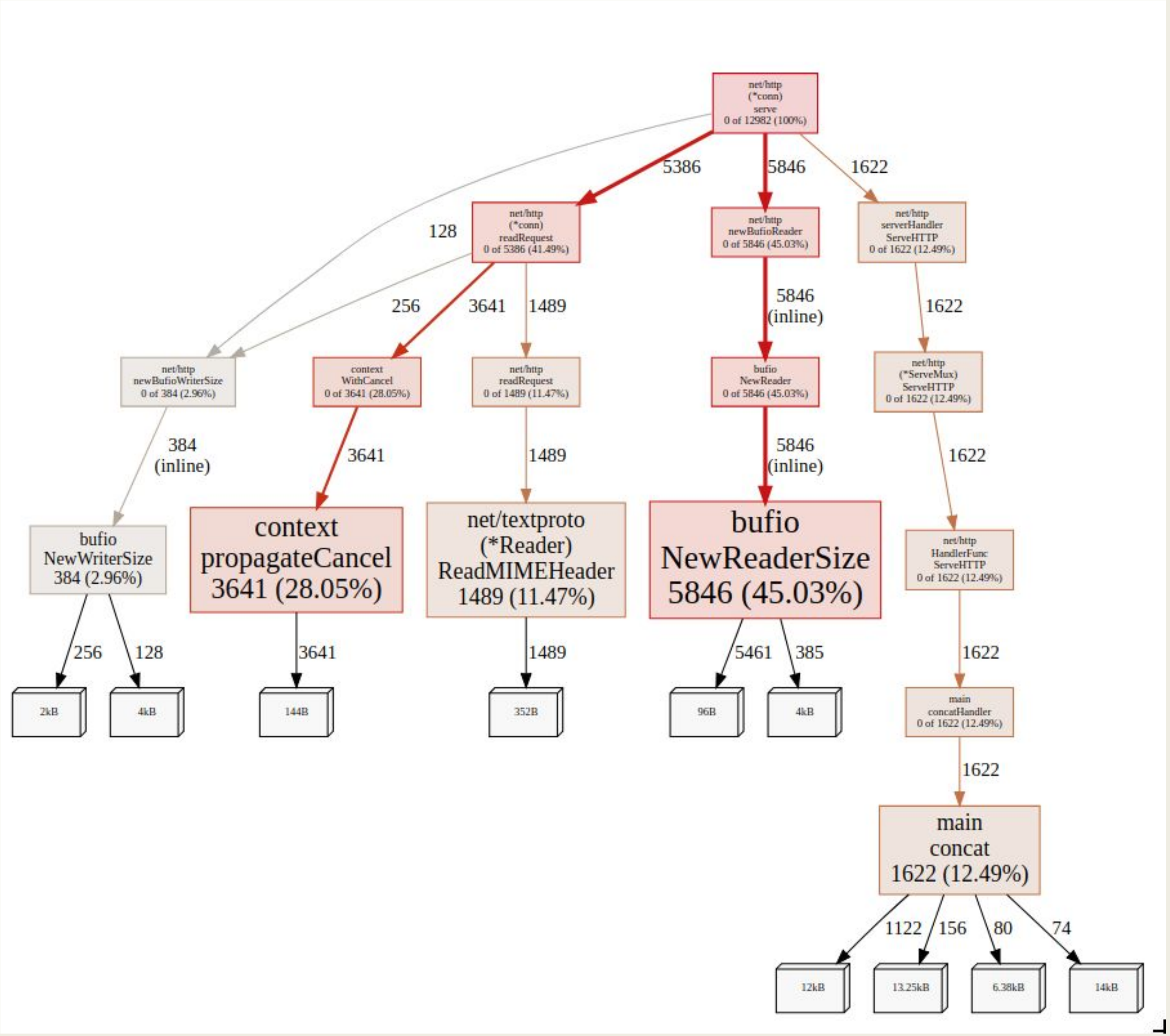
```
go tool pprof [options]
http://localhost:6060/debug/pprof/heap

// Available options
-inuse_space      Display in-use memory size
-inuse_objects    Display in-use object counts
-alloc_space      Display allocated memory size
-alloc_objects    Display allocated object counts

go tool pprof -http=localhost:<port>
/path/to/profile.pb.gz
```

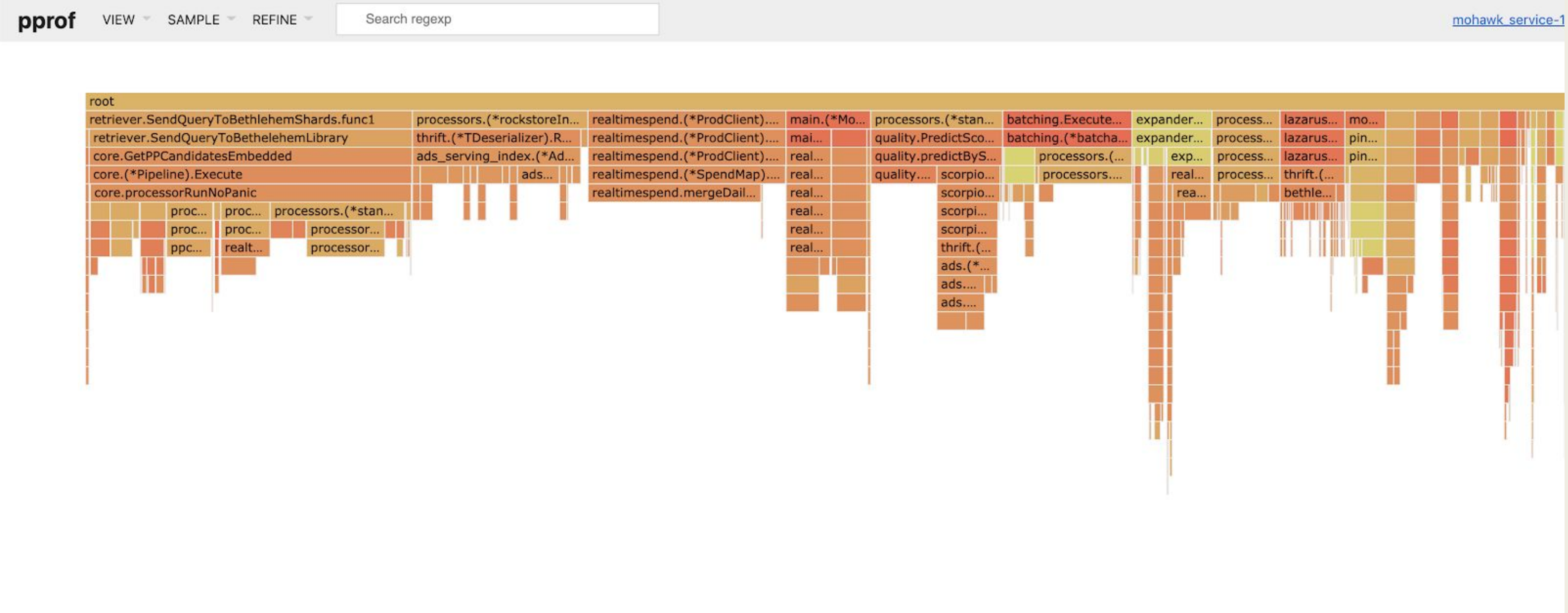
Source:
Go docs

# pprof


net/http (*conn) serve 0 of 12982 (100%)
5386
5846
1622
128
net/http (*conn) readRequest 0 of 5386 (41.49%)
net/http newBufioReader 0 of 5846 (45.03%)
net/http serverHandler ServeHTTP 0 of 1622 (12.49%)
256
3641
1489
5846 (inline)
1622
net/http newBufioWriterSize 0 of 384 (2.96%)
context WithCancel 0 of 3641 (28.05%)
net/http readRequest 0 of 1489 (11.47%)
bufio NewReader 0 of 5846 (45.03%)
net/http (*ServeMux) ServeHTTP 0 of 1622 (12.49%)
384 (inline)
3641
1489
5846 (inline)
1622
bufio NewWriterSize 384 (2.96%)
context propagateCancel 3641 (28.05%)
net/textproto (*Reader) ReadMIMEHeader 1489 (11.47%)
bufio NewReaderSize 5846 (45.03%)
net/http HandlerFunc ServeHTTP 0 of 1622 (12.49%)
256
128
3641
1489
5461
385
1622
2kB
4kB
144B
352B
96B
4kB
main concatHandler 0 of 1622 (12.49%)
1622
main concat 1622 (12.49%)
1122
156
80
74
12kB
13.25kB
6.38kB
14kB

Source: [matoski.com](matoski.com)

# pprof


pprof
VIEW
SAMPLE
REFINE
Search regexp
mohawk_service-1
root
retriever.SendQueryToBethlehemShards.func1
processors.(*rockstoreIn...
realtimespend.(*ProdClient)....
main.(*Mo...
processors.(*stan...
batching.Execute...
expander...
process...
lazarus...
mo...
retriever.SendQueryToBethelehemLibrary
thrift.(*TDeserializer).R...
realtimespend.(*ProdClient)....
mai...
quality.PredictSco...
batching.(*batcha...
expander...
process...
lazarus...
pin...
core.GetPPCandidatesEmbedded
ads_serving_index.(*Ad...
realtimespend.(*ProdClient)....
real...
quality.predictByS...
processors.(...
exp...
process...
lazarus...
pin...
core.(*Pipeline).Execute
ads...
realtimespend.(*SpendMap)....
real...
quality....
scorpio...
processors....
real...
process...
thrift.(...
core.processorRunNoPanic
realtimespend.mergeDail...
real...
scorpio...
rea...
bethle...
proc...
proc...
processors.(*stan...
real...
scorpi...
proc...
proc...
processor...
real...
scorpi...
ppc...
realt...
processor...
real...
thrift.(...
ads.(*...
ads....
ads....

# pprof

```
(pprof) list createCatalogMap
Total: 132263423
ROUTINE ======================== <CODE_PATH>
  105268459  105268459 (flat, cum) 79.59% of Total
          .  63815675    233: product := BuildProduct(productID, productPrice, productSellerID)
          .         .    234: if productPrice < minProductPrice {
          .         .    235:  minProductPrice = productPrice
          .         .    236: }
          .  20726392    237: catalogListing := catalogs.CreateListing(product, contextFeatures)
          .         .    238:
          .         .    239: // Create listing key by encoding productID and sellerID
          .         .    240: catalogListingKey := catalogs.CreateListingKey(productID, sellerID)
   20726392  20726392    241: catalogMap[catalogListingKey] = catalogListing
          .         .    242: return catalogMap
```

# How to limit the impact of GC?

Lower the number of objects on heap

Reduce the rate of object allocation

Optimize data structures for minimal memory usage

# Reduce long-living heap objects

Create objects on demand

Be mindful of using pointers

Strings & byte arrays are also pointers!

# Impact of removing strings

```
(pprof) list createCatalogMap
Total: 132263423
ROUTINE ======================== <CODE_PATH>
  105268459  105268459 (flat, cum) 79.59% of Total
          .   63815675    233: product := BuildProduct(productID, productPrice, productSellerID)
          .          .    234: if productPrice < minProductPrice {
          .          .    235:  minProductPrice = productPrice
          .          .    236: }
          .   20726392    237: catalogListing := catalogs.CreateListing(product, contextFeatures)
          .          .    238:
          .          .    239: // Create listing key by encoding productID and sellerID
          .          .    240: catalogListingKey := catalogs.CreateListingKey(productID, sellerID)
   20726392   20726392    241: catalogMap[catalogListingKey] = catalogListing
          .          .    242: return catalogMap
```

# Impact of removing strings

```
(pprof) list createCatalogMap
Total: 106261986
ROUTINE ======================== <CODE_PATH>
     34768   84576835 (flat, cum) 79.59% of Total
         .   63815675    233: product := BuildProduct(productID, productPrice, productSellerID)
         .          .    234: if productPrice < minProductPrice {
         .          .    235:  minProductPrice = productPrice
         .          .    236: }
         .   20726392    237: catalogListing := catalogs.CreateListing(product, contextFeatures)
         .          .    238:
         .          .    239: structKey := CatalogKeyStruct{
         .          .    240:  ProductID:      productID,
         .          .    241:  SellerID:       productSellerID,
         .          .    242: }
     34768      34768    243: catalogMap[structKey] = catalogListing

         .          .    244: return catalogMap
```

# Reduce the rate of allocation

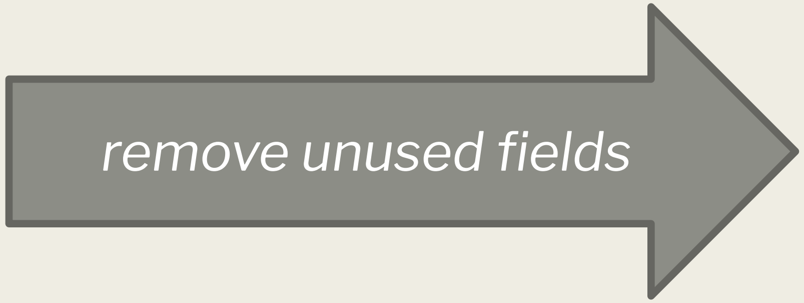
Utilize object pooling

*Warning: Can cause memory/data leaks if not used properly*

# Clean up unused data fields

**64 bytes**

```
type BadObject struct {
    A bool
    B int64
    C int32
    D bool
    E int32
    F bool
    G int32
    H bool
    I int64 // unused
    J bool // unused
    K int32 // unused
    L int64 // unused
}
```



*remove unused fields*

**40 bytes**

```
type GoodObject struct {
    A bool
    B int64
    C int32
    D bool
    E int32
    F bool
    G int32
    H bool
}
```

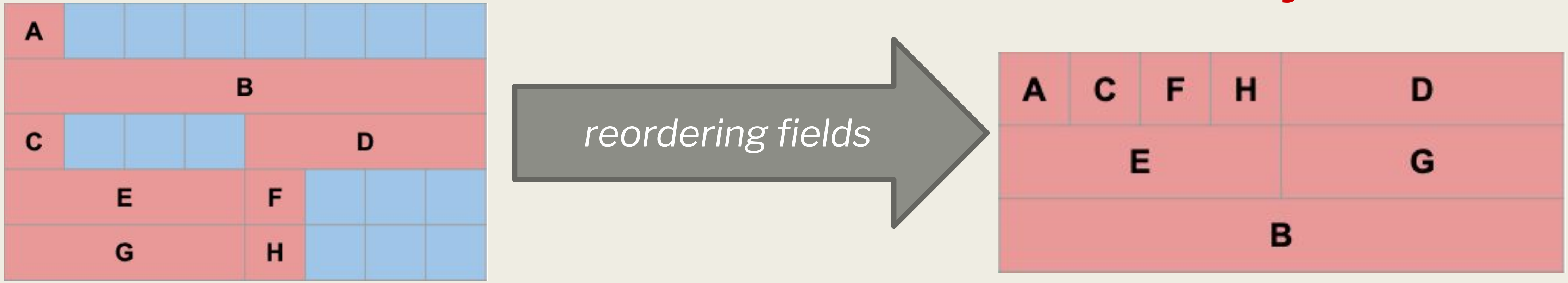
# Reorder fields for proper data alignment

```
type BadObject struct {
      A bool
      B int64
      C bool
      D int32
      E int32
      F bool
      G int32
      H bool
}
```

```
type GoodObject struct {
      A bool
      C bool
      F bool
      H bool
      D int32
      E int32
      G int32
      B int64
}
```

**40 bytes**

**24 bytes**



# Conclusion

Go GC is very powerful, but it’s not perfect

Go has great built-in tools to debug GC problems

GC optimizations can significantly improve performance for heavy use cases!

# THANK YOU!

*nroy@pinterest.com*