Golang’s Garbage

Andrey Sibiryov, Uber
In Uber, we run over 1700 microservices in production, written in different languages. At this scale and fanout, performance of each one of them matters.

- The team I work with runs a few CPU and memory intensive Go services processing a ton of requests from all around the infrastructure.
  
- Tens of millions datapoints ingested per second.

- Seventy five years of time series data queried per second.
«Go is building a garbage collector not only for 2015 but for 2025 and beyond <…> Go 1.5’s GC ushers in a future where stop-the-world pauses are no longer a barrier to moving to a safe and secure language. It is a future where applications scale effortlessly along with hardware and as hardware becomes more powerful the GC will not be an impediment to better, more scalable software». 
<table>
<thead>
<tr>
<th>Children</th>
<th>Self Command</th>
<th>Command</th>
<th>Shared Object</th>
<th>Object Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 90.92%</td>
<td>0.01%</td>
<td>m3dbnode</td>
<td>m3dbnode</td>
<td>[.]</td>
</tr>
</tbody>
</table>

- runtime.goexit
  + 42.48% runtimegcBgMarkWorker
  + 33.24% code.uber.internal/infra/statsdex/vendor/github.com/uber/tchannel
  + 5.12% code.uber.internal/infra/statsdex/vendor/github.com/uber/tchannel
  + 4.28% code.uber.internal/infra/statsdex/vendor/github.com/m3db/m3db/pers
  + 4.20% code.uber.internal/infra/statsdex/vendor/github.com/uber/tchannel
  + 2.69% runtime.mcall
  + 2.14% code.uber.internal/infra/statsdex/vendor/github.com/m3db/m3db/client
  + 2.06% code.uber.internal/infra/statsdex/vendor/github.com/m3db/m3db/cont
  + 1.19% code.uber.internal/infra/statsdex/vendor/github.com/uber/tchannel
  + 0.90% code.uber.internal/infra/statsdex/vendor/github.com/m3db/m3db/client
  + 0.70% runtime.bgsweep

GC scales effortlessly
GC scales even more effortlessly
Modern GC

• **Generational** – memory is divided into regions based on certain tenuring policies: no need to scan everything every time.

• **Compacting** – during each GC phase memory is defragmented: data locality improves, allocations become free.

• **Exact** – GC knows exactly what each byte of memory is: a pointer or a typed value. GC can move things around and update references.

• **Goal-based** – GC aims to keep a certain metric in user-specified bounds, e.g. pause time, heap size, etc.
Golang GC

- **Generational**—memory is divided into regions based on certain tenuring policies: no need to scan everything every time.

- **Compacting**—during each GC phase memory is defragmented: data locality improves, allocations become free.

- **Exact**—GC knows exactly what each byte of memory is: a pointer or a typed value. GC can move things around and update references.

- **Goal-based**—GC aims to keep a certain metric in user-specified bounds, e.g. pause time, heap size, etc.
50 years
WTF

• Reducing stop-the-world pause times with each Go release has a cost which Google’s marketing people don’t like to talk about.

• Background **GC CPU usage is linear with the number of pointers on the heap.**

• On a ~96GB heap populated with simple structures and interfaces, **Go spends up to 75% of total program runtime inside GC.**
### GODEBUG=gctrace=1

<table>
<thead>
<tr>
<th>STF</th>
<th>BGMSF</th>
<th>MTF</th>
<th>ASST</th>
<th>BG</th>
<th>IDLE</th>
<th>START</th>
<th>END</th>
<th>LIVE</th>
<th>CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4</td>
<td>17949</td>
<td>5.8</td>
<td>ms</td>
<td>clock,</td>
<td>17</td>
<td>60052/53289/3364+69 ms cpu,</td>
<td>38334 -&gt; 39982 -&gt; 31776 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>21556</td>
<td>3.7</td>
<td>ms</td>
<td>clock,</td>
<td>30</td>
<td>5929/64643/53115+45 ms cpu,</td>
<td>38339 -&gt; 40397 -&gt; 32694 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>17143</td>
<td>7.5</td>
<td>ms</td>
<td>clock,</td>
<td>22</td>
<td>58984/50885/5663+90 ms cpu,</td>
<td>39817 -&gt; 41613 -&gt; 32402 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>21064</td>
<td>4.3</td>
<td>ms</td>
<td>clock,</td>
<td>57</td>
<td>28487/62624/21636+52 ms cpu,</td>
<td>38882 -&gt; 41093 -&gt; 33218 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>18280</td>
<td>4.4</td>
<td>ms</td>
<td>clock,</td>
<td>16</td>
<td>8069/54668/49076+53 ms cpu,</td>
<td>39878 -&gt; 41529 -&gt; 32903 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>15909</td>
<td>3.5</td>
<td>ms</td>
<td>clock,</td>
<td>9.2</td>
<td>6840/47614/53832+24 ms cpu,</td>
<td>40200 -&gt; 41653 -&gt; 32746 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>16361</td>
<td>4.2</td>
<td>ms</td>
<td>clock,</td>
<td>17</td>
<td>36463/48581/34116+50 ms cpu,</td>
<td>40462 -&gt; 41596 -&gt; 32379 MB,</td>
<td>12 P</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>16683</td>
<td>3.9</td>
<td>ms</td>
<td>clock,</td>
<td>223</td>
<td>1433/49757/63019+31 ms cpu,</td>
<td>39572 -&gt; 40527 -&gt; 32266 MB,</td>
<td>12 P</td>
<td></td>
</tr>
</tbody>
</table>
Heap Bitmap

- Heap bitmap is a data structure that Go runtime uses to keep track of the memory's underlying type information. It's pre-generated by the compiler.

- **For each word** in memory, it keeps two bits of metadata — in most cases, whether it's a pointer or not and a debug bit.

- GC uses this bitmap to populate its working set for each cycle by recursively adding all pointers starting from those found in root objects — globals and stacks.
More Pointers = Slower Code
Easy Tricks

• **Channels** of pointerless structures or plain values are allocated as a single block and marked as non-scannable.

• **Maps** where both keys and values are pointerless structures are marked as non-scannable.

• **Pointers** can be stored as `uintptrs` and casted to actual types via `unsafe.Pointer` to also be marked as non-scannable.

• **Closures** force all stack variables of the enclosing function to escape to the heap. Don’t use closures – use functors.
Pooling

<table>
<thead>
<tr>
<th>sync.Pool</th>
<th>ObjectPool</th>
</tr>
</thead>
<tbody>
<tr>
<td>unbounded</td>
<td>bounded</td>
</tr>
<tr>
<td>separate per-P pools</td>
<td>one shared pool</td>
</tr>
<tr>
<td>purged on GC</td>
<td>retained on GC</td>
</tr>
<tr>
<td>lazily populated</td>
<td>eagerly populated</td>
</tr>
</tbody>
</table>
Uneasy Tricks

How can we dramatically reduce the number of pointers on the heap?
Native Heap

• **Step 1**: Allocate a huge block of memory via `mmap(2)`.

• **Step 2**: Slice it into chunks of the specified type’s size and a little bit extra for the header.

• **Step 3**: ???

• **Step 4**: Profit!
mmap(2)

HDR

[N]byte

...
Native Heap

• The **native heap is invisible to the Go runtime**, so GC ignores it even if it sees a pointer into it. This also means that `free(3)` is back.

• For the same reason, **native heap pointers cannot keep objects on the GC heap alive** (think weak pointers).

• It means **all pointers** in objects on the native heap **must point to the native heap**, unless there’s another root somewhere.

• Since the internal structure of builtin types is not available to the user code, **maps, channels and other builtins won’t work**.
Native Slices

// Make a heap of 1024 16-byte blocks.

h := heap.New(1024,
  reflect.ArrayOf(16, reflect.TypeOf((byte)(0)))
)

// Allocate a ninja slice.

v := *(*[]byte)(unsafe.Pointer(&reflect.SliceHeader{
  Data: reflect.ValueOf(h.get()).Pointer(),
  Len:  0,
  Cap:  16,
})))
// Make a heap of 1024 16-byte blocks.
var h = heap.New(1024,
    reflect.ArrayOf(16, reflect.TypeOf((byte)(0))))

// Allocate a ninja slice.
var v = *(*[][byte])(unsafe.Pointer(&reflect.SliceHeader{
    Data: reflect.ValueOf(h).Pointer(),
    Len: 0,
    Cap: 16,
}))
Native Slices

type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
// Make a heap of 1024 16-byte blocks.
h := heap.New(1024, reflect.ArrayOf(16, reflect.TypeOf((byte)(0)))

// Allocate a ninja slice.
3-4x improvements in end-to-end latency and query response times
What’s Next?

• Figuring out more ways to have less objects or move more objects to the Native Heap.

• **ROC** – Request Oriented Collector. It’s a generational CMS with a special tenuring policy based on request-response hypothesis. It’s nineties all over again!

• Porting performance-critical code paths to **C++** or **Rust** and calling into it via CGo or local IPC.
Thank you!

Andrey Sibiryov, SRE, Uber New York

kobolog@uber.com @kobolog