Polar Signals

OOMProf

by Tommy Reilly

OOMProf

OOMProf is a set of eBPF programs that copy the Go runtime's built in memory profiling buffers into eBPF maps when OOMs occur so the current memory usage can be analyzed after the program is killed.

Motivation

Polar Signals provides a continuous production-ready profiling solution that covers CPU, **memory**, off-CPU (aka scheduling) and GPU profiling.

There's no more important time to know what the memory profile of your program looks like as when the kernel decides your program is the "problem" and nukes it out of orbit

OOMProf has enabled our customers (and ourselves) to understand exactly why OOMs occur and fix them quickly

In practice scraping a memory profile every couple minutes can miss the root cause of an OOM, things can go off the rails quickly! Sometimes in one allocation!

The Real Motivation

Go Memory Profiler

One of Go's nicer features is a built in statistical memory profiler that samples memory allocation stacks.

Its turned on automatically if your program references the runtime.MemProfile function, typically done via the net/http/pprof package

By default it takes a sample every 512kb of allocations, which means it walks the stack a sticks a record into a giant hash table

When a profiled allocation is freed the record for that stack its recorded to enable "in_use" profiling

But Go is GC'd? How do you not count garbage?

Good question! Go stores alloc/free stats in cycle buckets and when alloc/frees happen they are accounted into a "future" bucket, after a GC sweep the future buckets are merged to an "active" bucket so the active reported memory is what the memory looked like at the end of the last sweep, as if there was never any garbage.

This is important because for oomprof we want to report the future buckets, even though some allocations may no longer be reachable they are taking up real memory until a sweep happens

Process Registration

The first step in oomprof'ing is registering PIDs that you care about in an eBPF map, this stores the PID and looks up the `"runtime.mbuckets"` address where the profiling data lives.

OOMProf can be used as a library where you self register or if you use the Polar Signals profiling agent and enable oomprof we'll automatically watch all Go processes.

The library use case is complicated because you need a separate process to be the "watcher" for your process so we'll just talk about the Polar Signals agent.

Hooking into the kernel

To know if your program is getting oom killed you can just hook into the “`oom/mark_victim`” tracepoint, this is called when your program has been picked for OOM killing. When this occurs we look up the pid to see if its a “known” Go process and if it is we make a note that that pid is about to be killed

Running eBPF in the right context

The mark_victim tracepoint can get fired when a different process requests memory, ie the process asking for memory isn't necessarily the one that gets killed. That means we can't just start reading the profilers buckets, we may be in a different process!

Luckily the process that's getting killed gets sent a signal so we can attach another eBPF program to "signal/signal_deliver" tracepoint and do eBPF map lookups to see if the signal target pid is the victim pid.

Yes this does mean that signal delivery will get marginally slower but the overhead of a uprobe and a couple eBPF map lookups is small and signals aren't typically firing at a high rate or on the critical path for real work.

Putting it all together

`parca-agent` installs eBPF programs to kernel tracepoints and creates maps to hold the the go_procs and profile records

`parca-agent` scans for Go processes and registers them w/ bucket address

OOM happens triggering mark_victim and signal delivery tracepoints

Signal delivery program copies mbuckets into bpf map and notifies `parca-agent` via `bpf_perf_event_output`

`parca-agent` is waiting on read for these events and read the bucket map and turns it into a pprof profile

Profile is uploaded to Polar Signals cloud for safe keeping

Does it work? Sometimes!

The profiler hash table has 179,999 buckets and they are collision chained, so in theory you could have an infinite number of records to record

eBPF only allows 1M instructions and our current implementation can do 3362 buckets per program

But we can use tail calls to get to 33*3362 = 110946 buckets. But to record that many buckets would require an eBPF map of ~80MB so we limit it by default to 60k buckets which takes ~40MB.

So in theory if you hit the 60k limit there's 50k headroom.

What if you have more?

If you have more bucket than we can handle we'll stop when we get to the limit and we'll include a "complete" status with the profile so you know its truncated

An incomplete profile may or may not be useful, if you know you're program and the memory allocations that caused the oom show up in the profile then it could be priceless but you have no idea what was in the missed buckets so its a bit of a crap shoot

Why Limits?

The eBPF program is a loop over the buckets that does 3 bpf_read operations per bucket

Because its a linked list of individually allocated records there is no other way, we have to read a bucket to know where the next one is

There's 3 reads because the stack size is dynamic so we have to read its length first and the record payload is after the stack (so header->stack->record)

Go by default has 1024 max stack, oomprof only reads the top 64 frames to limit memory usage

What could go wrong?

Bpf reads sometimes fail, c'est la vie! If this happens we mark that there was a read error and mark the profile "incomplete"

Sometimes the oom killer gets impatient and starts killing other processes while we're profiling, including `parca-agent`!

And probably 17 other things...

Future directions

Jemalloc/tcmalloc/mimalloc support

Stripped binary support, can we find “runtime.mbuckets” list by disassembling a function we know that references it (by consulting gopclntab)

Push profiling, usually memory profiling is done adhoc by scraping the pprof endpoint but we could have the program intelligently decide when to send profiles (ie when a heap expansion occurs or some interesting memory intensive thing happens like building an index)

Try it out!

The library:

http://github.com/parca-dev/oomprof

Complete working implementation:

http://github.com/parca-dev/parca-agent

Just add –enable-oomprof to the `parca-agent` flags

Polar Signals

Thank you for listening!

Tommy Reilly

tr@polarsignals.com

Any questions?