Tuning Golang Garbage Collector

Table of contents

Written by: David Lin

Summary

`GOMEMLIMIT` is a tuning parameter added in v1.19 that, together with `GOGC`, dictates the behavior of GO's Garbage Collector (GC). Prior to GO v1.19, `GOGC` was the only parameter that can be used to modify the GC's behavior. This new feature aims to increase GC related performance as well as avoid GC-related out-of-memory situations.

Glossaries

Some common terminologies that will be used throughout and their respective definitions.

Heap: Subset of memory that's managed by GC.

- Memory requested by the application that the GO Compiler couldn't find a place for in compile time.
- Non-heap memory includes memory for GO Routine stacks, GC metadata, and other various GO Runtime data structures.

Live: Memory reachable by program.

- Memory that the GC discovers is actively used by the program.
- GC is basically a bunch of code that needs to be executed to make this discovery.

New: New Memory that may or may not be live.

- Memory that the application has asked runtime to allocate for it since the last time the GC ran. Hence, the liveness of it has not yet been determined.

Total Heap Memory:

$$Total\ heap\ memory\ =\ Live\ heap\ +\ New\ heap\ memory$$

Total GC CPU Cost: derived from GC Cost Model

$$Total\ GC\ CPU\ cost\ =\ (GC\ CPU\ cost\ per\ cycle)\ *\ (GC\ frequency)\ *\ Time\ Period,\ T$$

Written by: David Lin

Garbage Collection in Golang

GC Cost Model

Before we dive deep, consider this model of GC cost based on three concepts.

1. The GC involves only two resources: CPU time, and physical memory.
2. The GC's memory costs consist of live heap memory, new heap memory and space for metadata that are small in comparison.
3. The GC's CPU costs are modeled as a fixed cost per cycle, and a marginal cost that scales proportionally with the size of the live heap.

Although this model accurately classifies the major components of the GC, it does not consider the size of these components nor their interactions. To model that, we'll use what is referred to as a `steady-state`, with the following assumptions:

1. The rate at which the application allocates new memory (in bytes per second) is constant.
2. The marginal costs of GC are constant.
   Note: the steady-state may seem contrived, but it's representative of the behavior of an application under some constant workload. Naturally, workloads can change even while an application is executing, but typically application behavior looks like a bunch of these steady-states strung together with some transient behavior in between.

In the `steady-state`, while the live heap size is constant, every GC cycle will look identical in the cost model as long as the GC executes after the same amount of time has passed. That's because in that fixed amount of time, with a fixed rate of allocation by the application, a fixed amount of new heap memory will be allocated. Hence, assuming live heap size constant, and that new heap memory constant, memory use is always going to be the same.

If the GC execution frequency is lowered, more memory would be allocated but with each GC cycle incurring the same CPU cost, resulting in a lower overall CPU cost over time. The inverse is true if the GC execution frequency is raised. In which case, less memory would be allocated and CPU costs would be incurred more often.

This scenario is clear to demonstrate the fundamental trade-off between CPU and memory costs that a GC incurs, governed by the execution frequency of the GC. The GC collection frequency is configurable using the `GOGC` tuning parameter, modifiable by the user.

Written by: David Lin

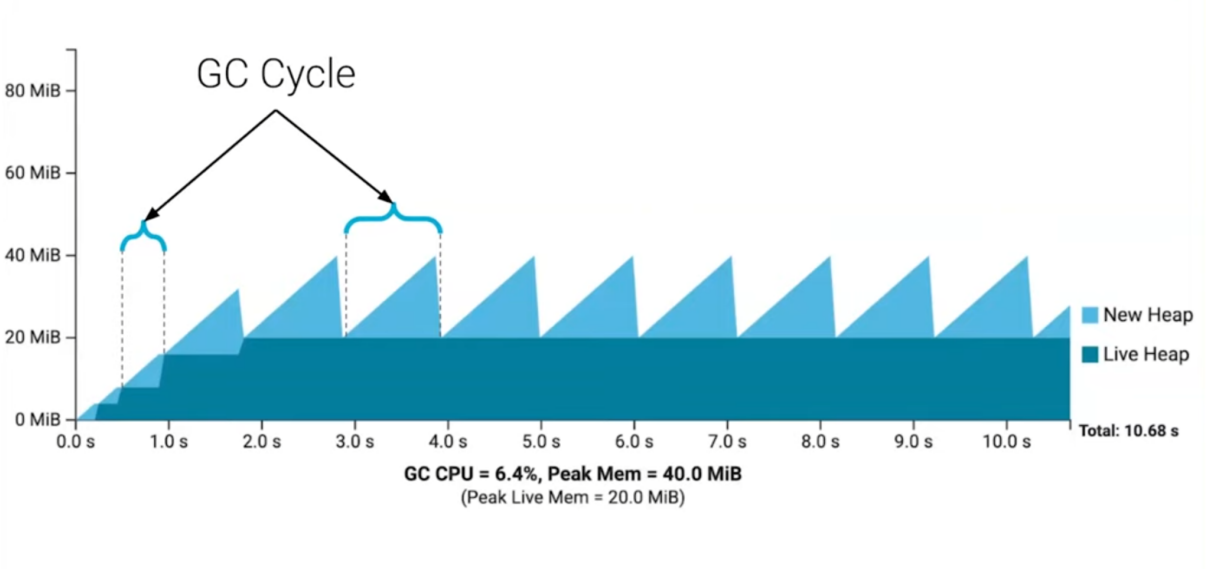
What is `GOGC`?

`GOGC` on the high level allows the user to take the trade-off between CPU and Memory usage, taking into account which is costlier for the business case.

On the lower level, `GOGC` sets the garbage collection target percentage, where a collection is triggered when the ratio of freshly allocated data to live data remaining after the previous collection reaches this percentage. For example, a `GOGC` value of 100 would mean that the target percentage would be 100% and that the size of the total heap is allowed to grow to 2x the size of the live heap. It is also crucial to note that GC cycles cost CPU resources to run.

Visualizing GO GC

GO docs representation



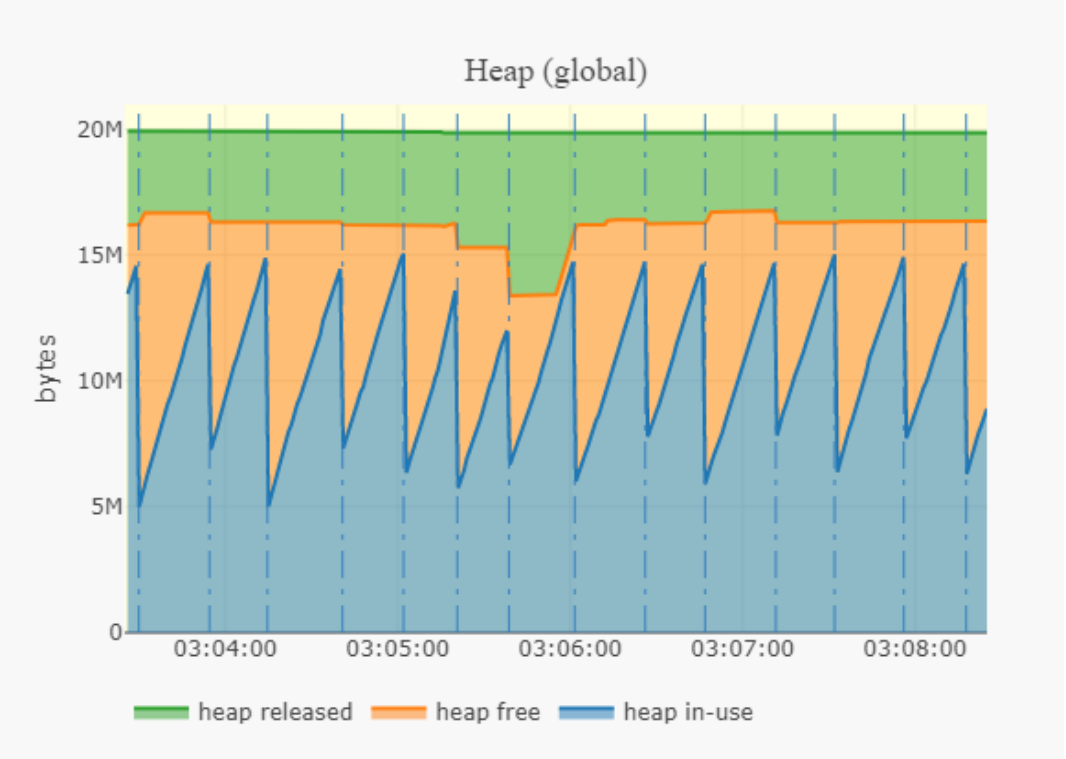
Common simplified representation by Golang Team.

This representation does not take into account the other memory beyond that of live and new heap. The new heap size grows constantly and at some point, the GC decides to execute and at this point, the live heap might change but the new heap always drops to zero.

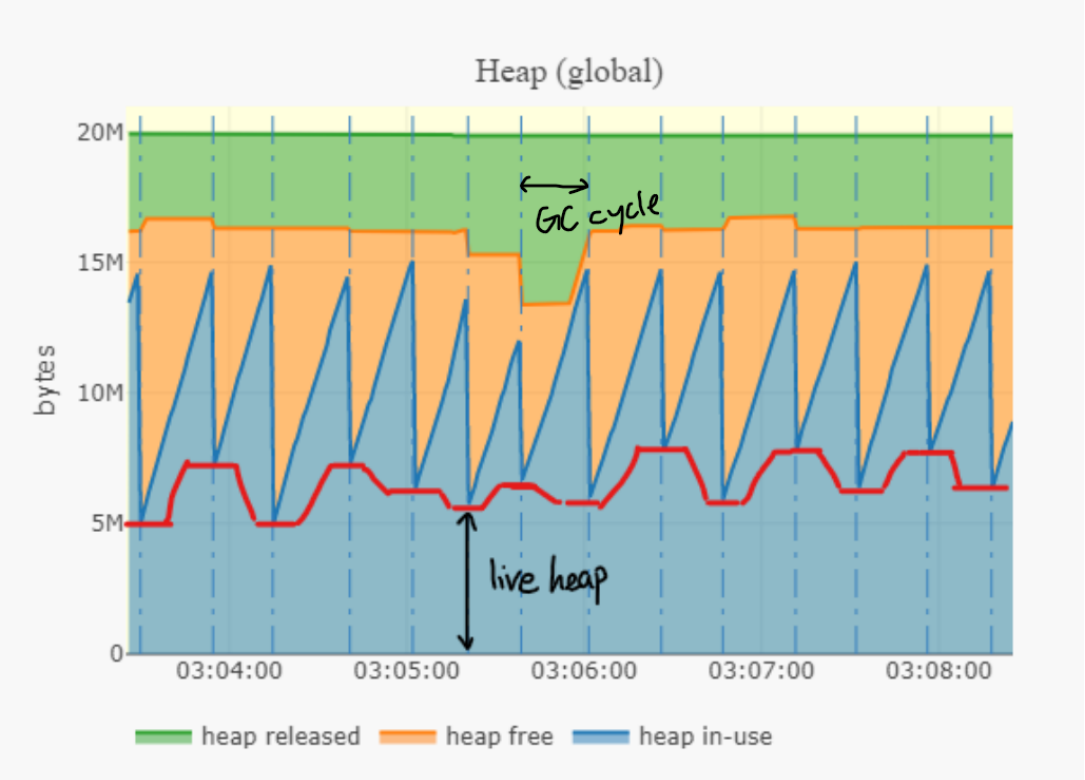
Written by: David Lin

Memory cycles in practice (Statsviz)

The following references will utilize the Statsviz visualized memory cycles, referencing the global heap measured from an isolated container with a singular GO Binary running.



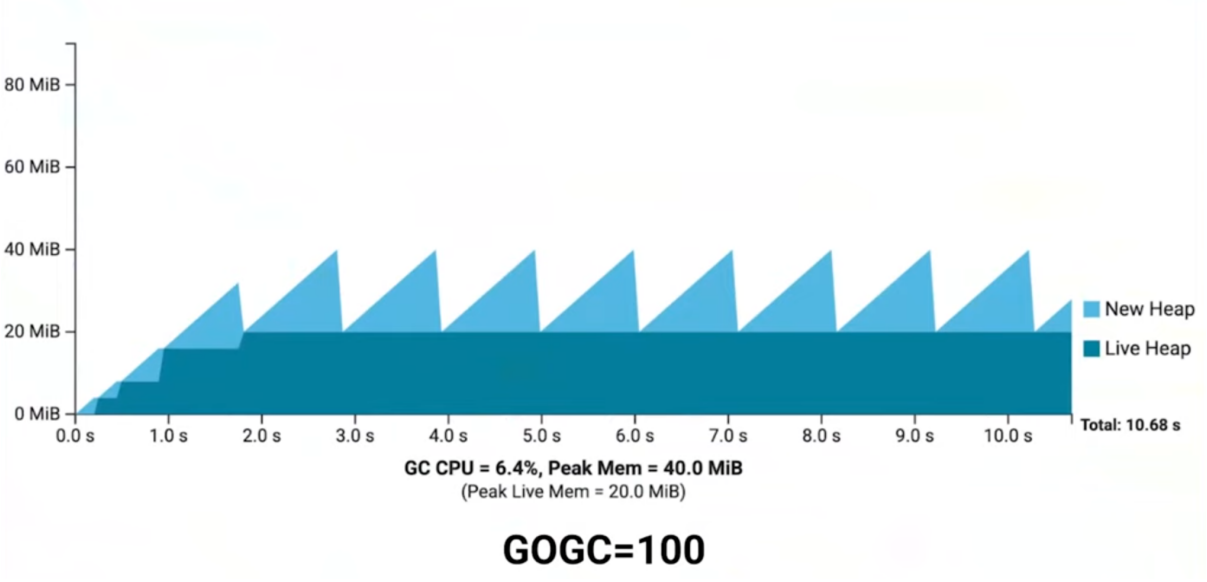
GOGC=100

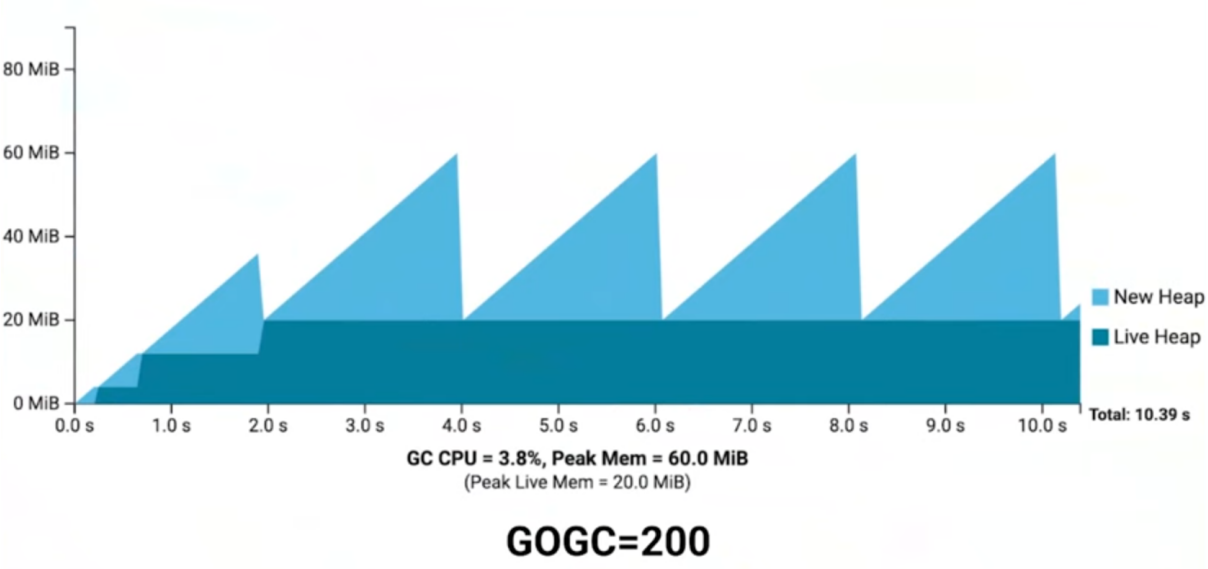


GOGC=100

Written by: David Lin

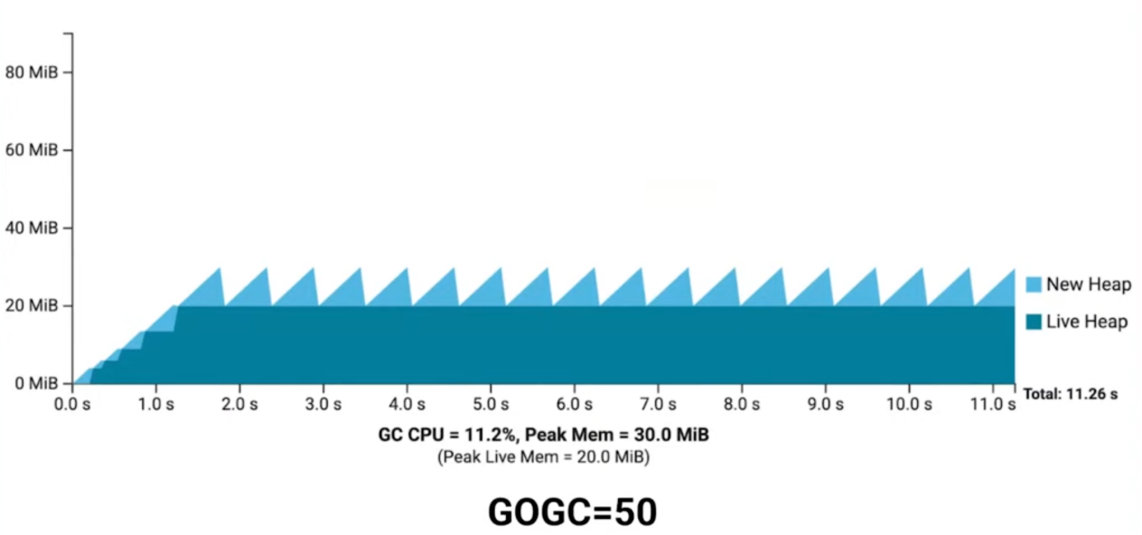
Visualizing effects of GOGC (In Theory)





As `GOGC` value increases from 100 to 200, fewer peaks are observed and peak mem is increased from 40 MiB to 60 MiB (from 2x live heap size to 3x live heap size) and CPU time is reduced by almost half.

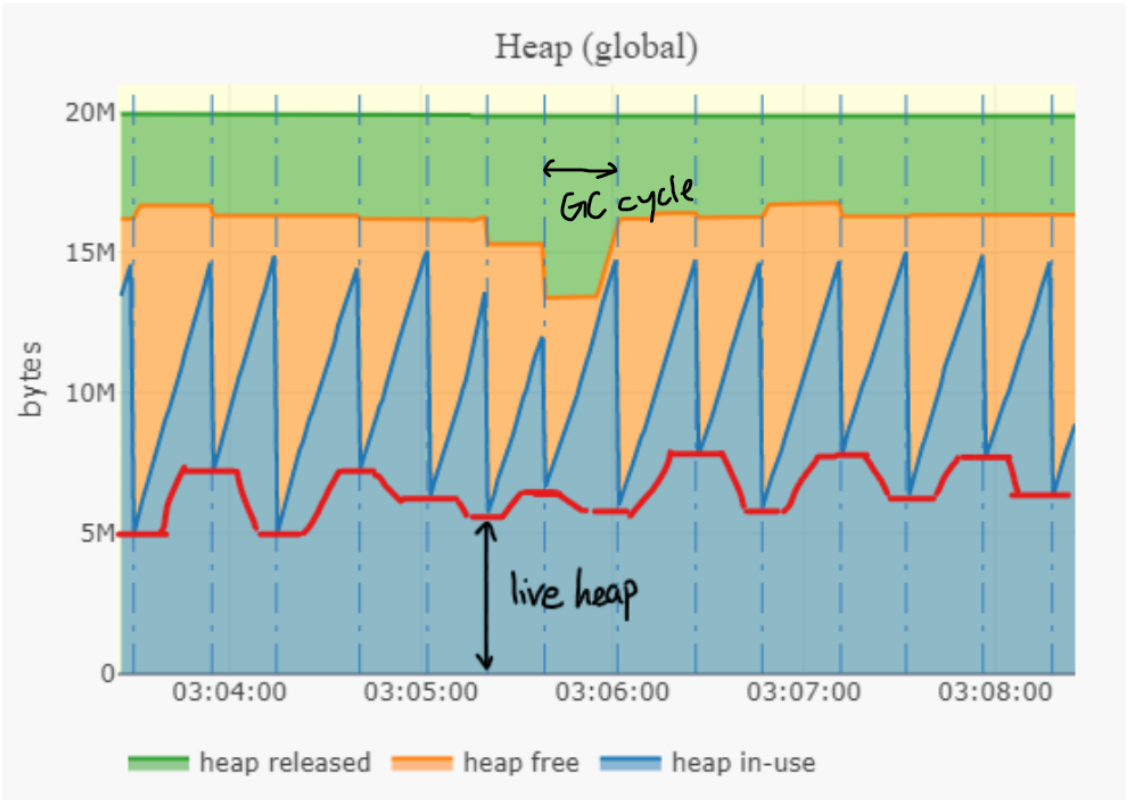
Written by: David Lin



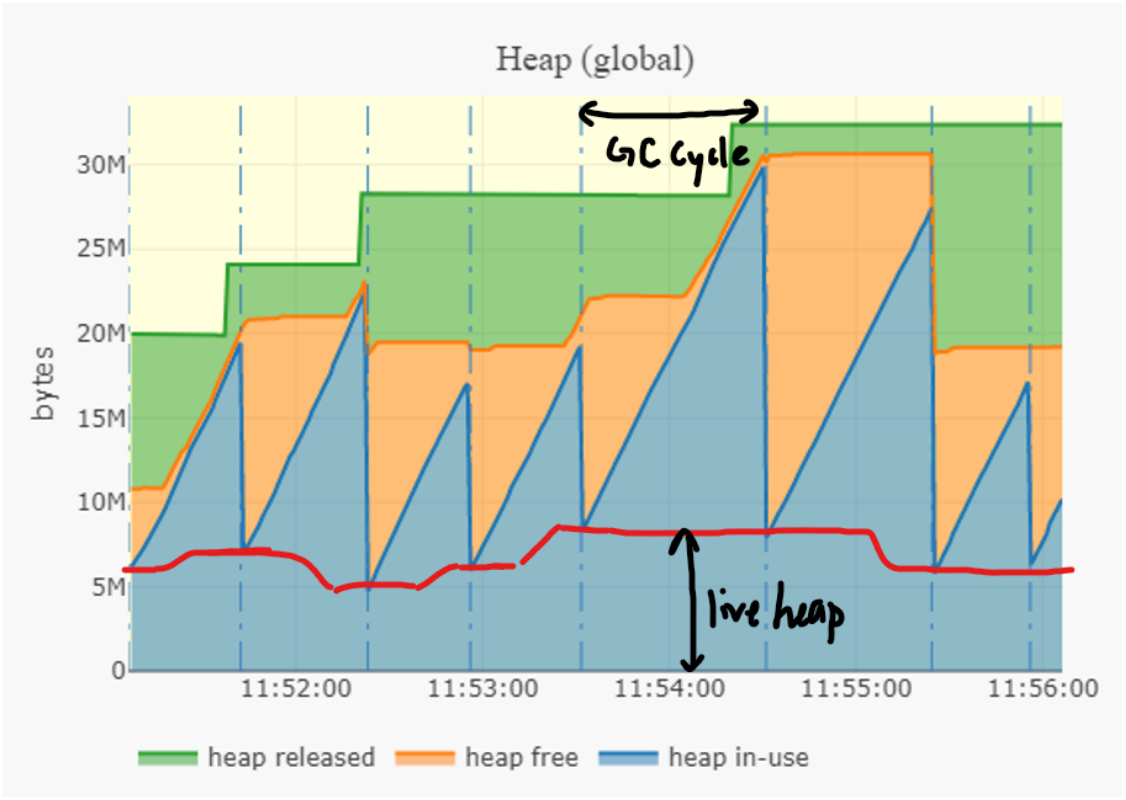
As `GOGC` value decreases from 100 to 50, more peaks are observed and peak mem is reduced from 40 MiB to 30 MiB (from 2x live heap size to 1.5x live heap size) and CPU time is increased by almost twice.

Written by: David Lin

Visualizing effects of GOGC (In Practice)



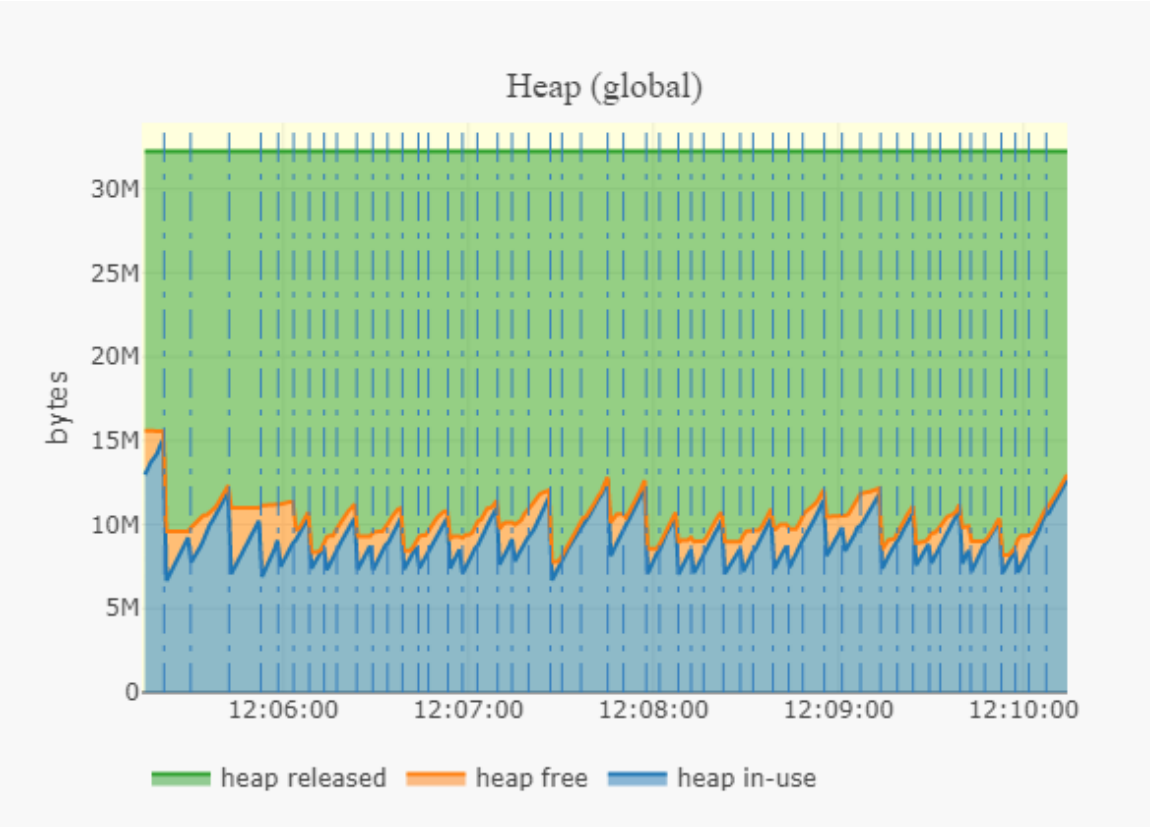
GOGC=100



GOGC=200

Written by: David Lin

As `GOGC` value increases from 100 to 200, we can make similar observations in practice. Fewer peaks are observed, peak memory is increased from 15 MiB to 30 MiB (from 2x live heap size to 3x live heap size) and CPU time is reduced by almost half.



GOGC=50

The converse holds true for `GOGC` values from 100 to 50.

Written by: David Lin

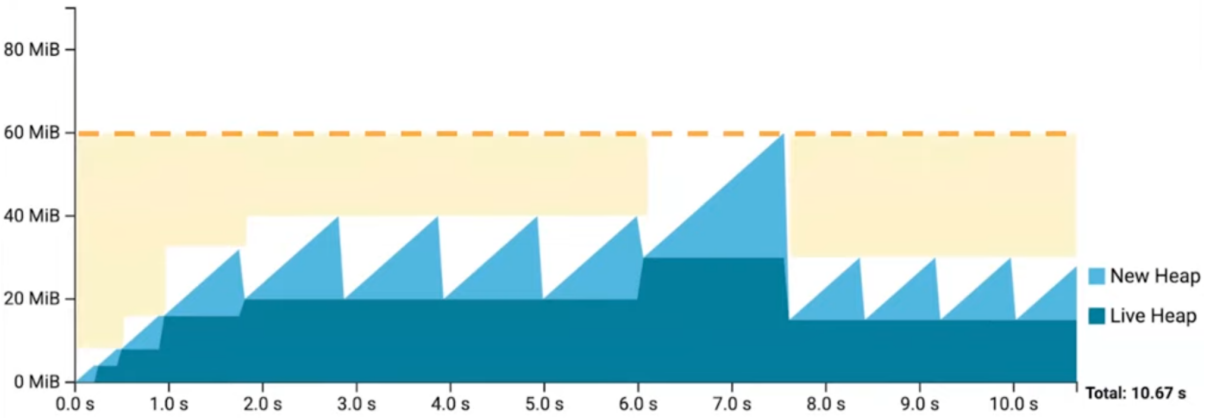
Prior to 1.19

GOGC Limitations

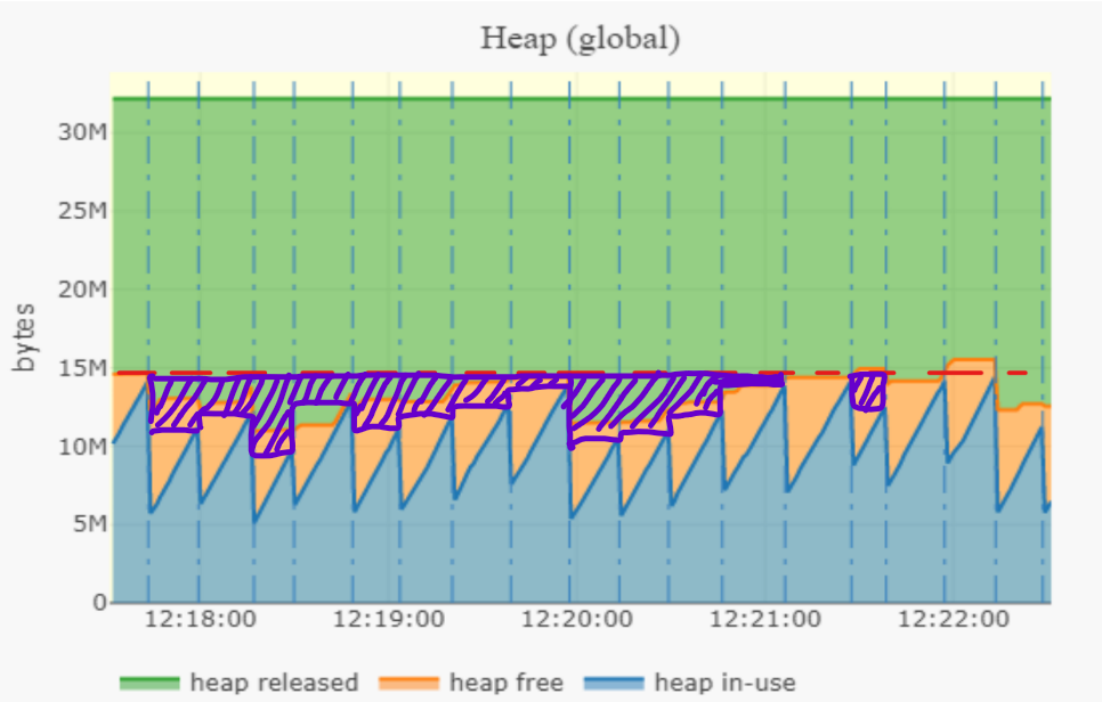
To fully grasp the limitations of `GOGC`, consider the following scenario:

- Assume that we have an application container that has 60 MiB in capacity
- Assume that this container only runs a singular pure GO Binary
- Assume that live heap size is about 20 MiB with a transient 10 MiB increase

With `GOGC` at 100, a bulk of memory available is wasted (see yellow) as heap is only allowed to grow 2x of the live heap size of the last cycle.



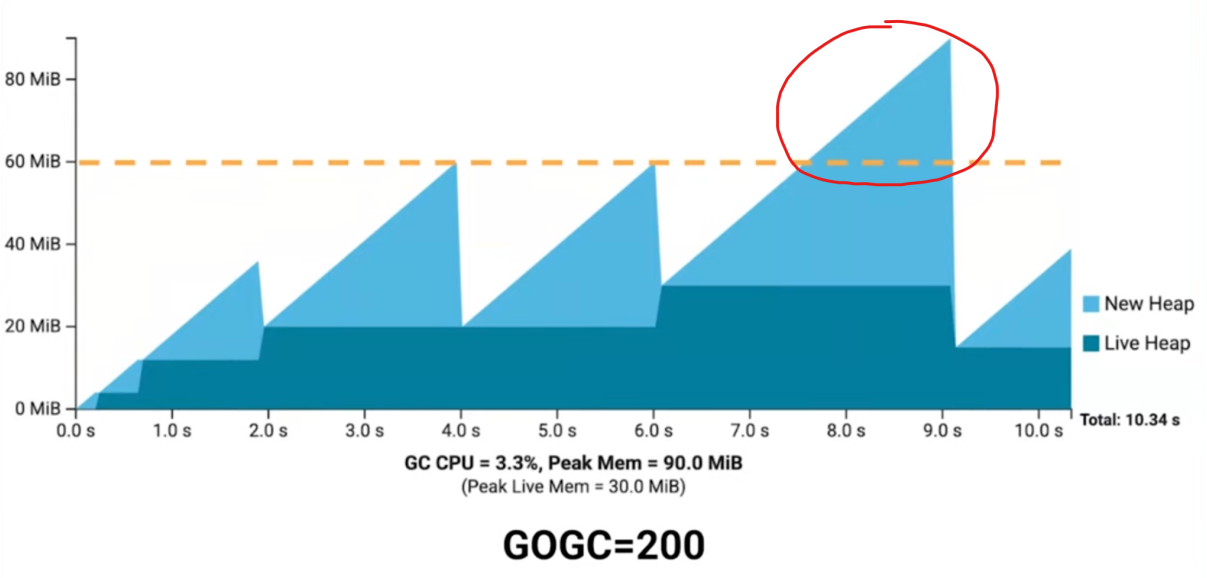
GOGC=100



GOGC=100, Memory Limit=20 MiB

Written by: David Lin

Turning `GOGC` up to 200 makes better use of available memory but the application exceeds memory capacity in the brief moment of transient increase.



This is not ideal as assuming the app runs 90% on 20 MiB live heap, and only 10% is expected to spike, the 200 value would allow for better fit of memory capacity, but at risk of complete failure of the app if it ever exceeds that memory capacity, even it only occurs for a brief 10% of the time.

`GOGC` will never allow use of that extra exceeded memory as the program is tied to peak memory use and also considering that the GC operates in frequency proportional to the `GOGC` which is proportional to heap overhead and inversely proportional to CPU costs hence it is completely unaware of memory limits and purely acts in proportion.

Written by: David Lin

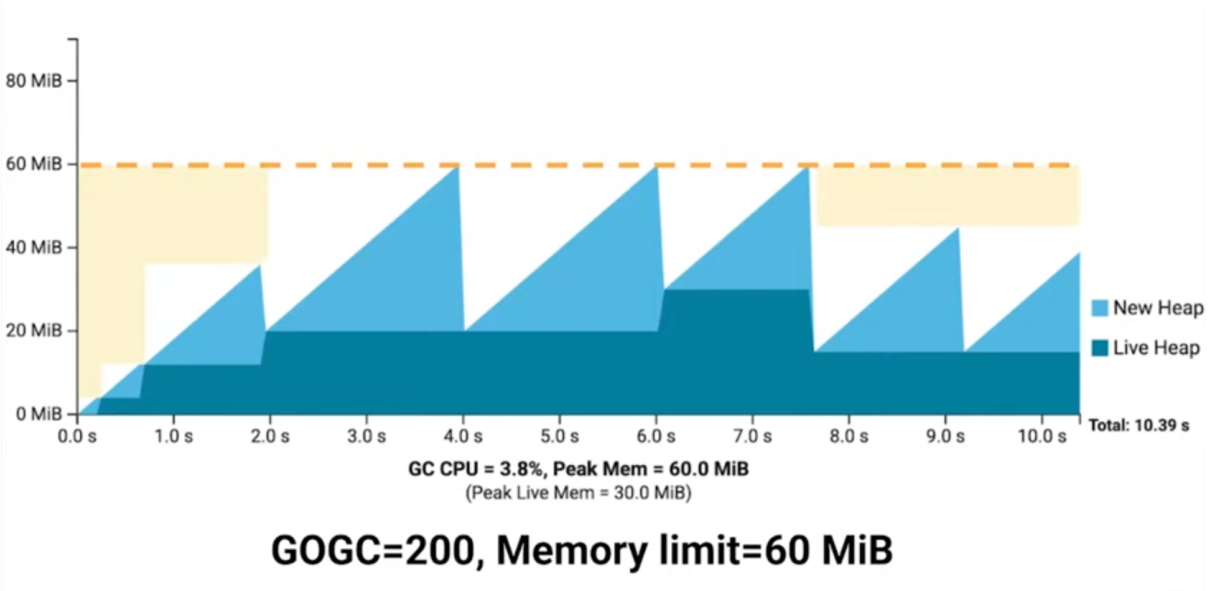
1.19 and Beyond

Implementation of GOMEMLIMIT

`GOMEMLIMIT` is essentially a soft total memory limit tracked by the GO Runtime. It works in conjunction with `GOGC` to define the GC behavior, more specifically, when it decides to run the GC cycle. It, however, does not track external memory sources (C or Kernel Memory) so users should consider leaving a 5-10% headroom from container limit to account for potential external memory use.

How `GOMEMLIMIT` addresses GOGC 1.18 limitations

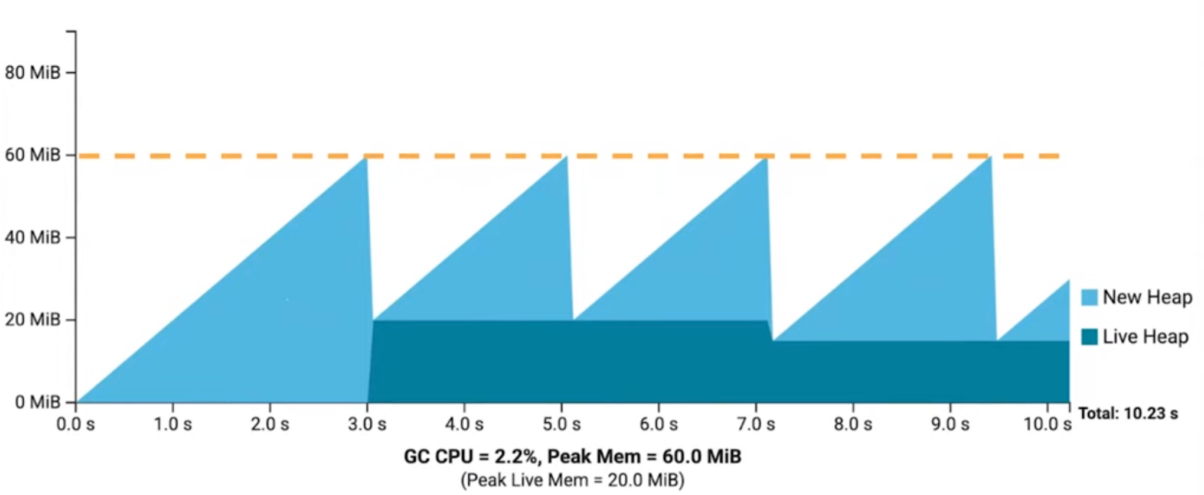
Consider the previous scenario underlined at `GOGC` limitations. With `GOGC` set at 200, where some transient part of the process exceeds the memory capacity at 60 MiB, with the implementation of a soft memory limit, once the heap size approaches that memory limit, the GC is set to become more aggressive and attempts to run more frequent cycles. Conversely, as the heap size is further from the memory limit, the GC is set to be less aggressive and attempts to run less frequent cycles, saving up CPU time and making full use of the allocated memory resources. This results in a more frequent cycle as the application experiences the transient spike in live heap size, capping the total heap at 60 MiB and adding more collection cycles during that spike, which prevents the application from crashing by paying a higher CPU cost to stay within memory capacity. On top of this, the application is still able to benefit from the `GOGC=200` average case.



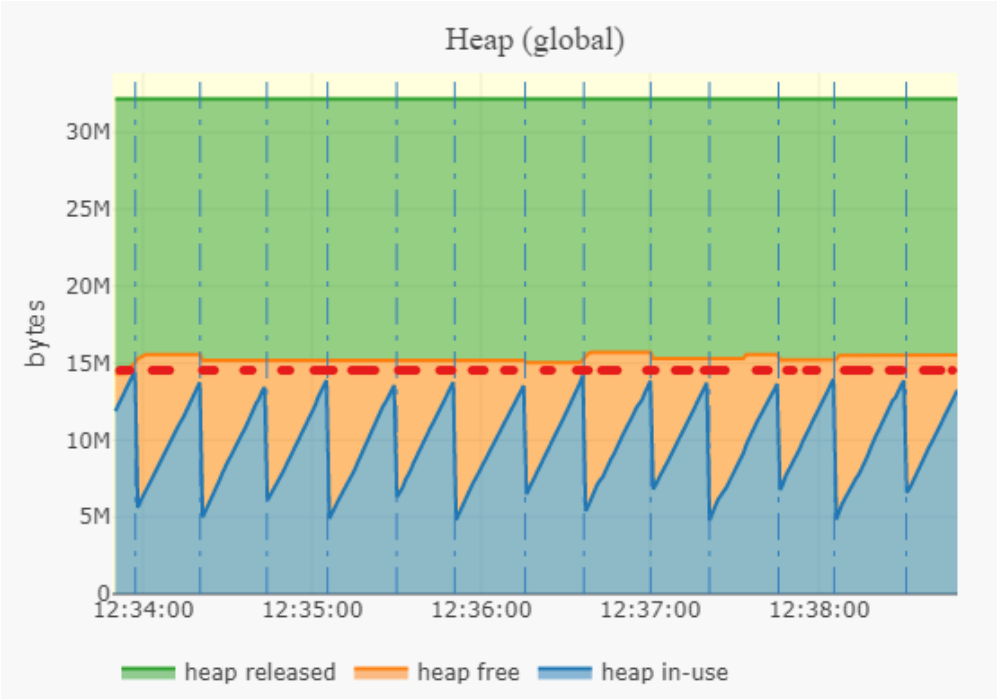
Looking at the graph, there are still some regions of memory that are yet to be fully utilized. The key then would be to increase `GOGC` to be as high as possible, in which case would be to set it to “off”. This means that the GC cycle will only trigger when heap size is near the defined memory limit, `GOMEMLIMIT`.

Written by: David Lin

In the following scenario, where `GOGC=off`, all the previously available regions of memory are now fully utilized, in other words, the container's allocated memory is fully utilized. It is also worth noting that this is accomplished without ever risking a possibility of running out of memory.



GOGC=off, Memory limit=60 MiB



GOGC=off, Memory Limit=20MiB

By maximizing memory capacity, taking into account the GC Cost Model, it is then conclusive that CPU cost by GC is minimized as the frequency of the GC cycle is minimized. This would benefit the program as more CPU resources are available for use.

Written by: David Lin

Hesitations of a Memory Limit Implementation

A large part of the hesitation to implement a memory limit was to figure out an effective mitigation strategy against `thrashing`.

Consider a scenario where the memory limit is set close to the live heap size. In that case, the GC will have to be much more aggressive and trigger very frequently to keep new heap size small such that the total heap does not exceed the memory limit. As the memory limit approaches the live heap size, the frequency of GC cycle increases indefinitely to the point that the GO Runtime is not able to maintain the limit as it requires some headroom to continue its process. Given the high frequency of GC cycles, the CPU cost is very much taking over the available CPU resources to the point where the application struggles to make any progress at all. This behavior is known as `thrashing` and it is a major issue since it is arguably worse than running out of memory as it can cause the application to stall indefinitely.

Another major concern for the GO team was that a new tuning parameter would greatly impact the testing and maintenance of an already active myriad of configurations for the GO Runtime, which is not only a large burden on the GO team but also on the ecosystem as a whole.

Mitigating Thrashing Conditions

The main mitigation strategy derived by the GO team was to limit the GC CPU time directly and force the application to run. By doing this, the GC must allow the application to continue allocating new memory, potentially past the defined memory limit.

Best Practices

1. Main intended use case is to improve resource efficiency with single pure GO binary running in a container. In which case, the `GOGC=off` with `GOMEMLIMIT`= 90% of container memory would be ideal.
2. Only set limit when we have full control of deployment environment (eg GO program running alone in a container)
3. Avoid setting memory limits in environments with unpredictable changes in external memory use (eg CLI app or desktop app)

References

- [GO Official GC Guide](https://go.dev/doc/gc-guide)
- [GopherCon 2022: Madhav Jivrajani - Control Theory and Concurrent Garbage Collection Deep Dive](https://youtu.be/We-8RSk4eZA)
- [Statsviz](https://github.com/arl/statsviz)
- [Weaviate article on GOMEMLIMT](https://weaviate.io/blog/2022/08/GOMEMLIMIT-a-Game-Changer-for-High-Memory-Applications.html)

Written by: David Lin