Go 1.5 concurrent garbage collector pacing

Austin Clements
3/10/2015
golang.org/s/go15gcpacing
Comment at https://groups.google.com/forum/#!topic/golang-dev/YjoG9yJktg4

Introduction

Prior to Go 1.5, Go has used a parallel stop-the-world (STW) collector. While STW collection has many downsides, it does at least have predictable and controllable heap growth behavior. The sole tuning knob for the STW collector was "GOGC", the relative heap growth between collections. The default setting, 100%, triggered garbage collection every time the heap size doubled over the live heap size as of the previous collection, as shown in figure 1.

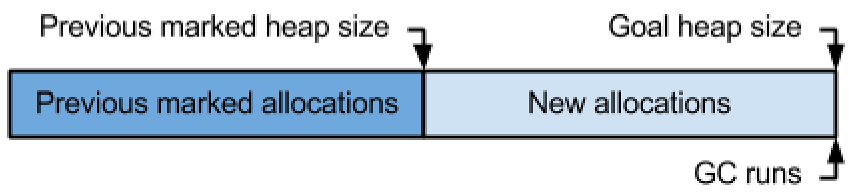

Figure 1. GC timing in the STW collector.

Go 1.5 introduces a concurrent collector. This has many advantages over STW collection, but it makes heap growth harder to control because the application can allocate memory while the garbage collector is running. To achieve the same heap growth limit the runtime must start garbage collection earlier, but how much earlier depends on many variables, many of which cannot be predicted. Start the collector too early, and the application will perform too many garbage collections, wasting CPU resources. Start the collector too late, and the application will exceed the desired maximum heap growth. Achieving the right balance without sacrificing concurrency requires carefully pacing the garbage collector.

This document proposes a mechanism to perform this pacing by adjusting the GC trigger point and scheduling the CPU to achieve the desired heap size and CPU utilization bounds.

Optimization goals

GC pacing aims to optimize along two dimensions: heap growth, and CPU utilized by the garbage collector.

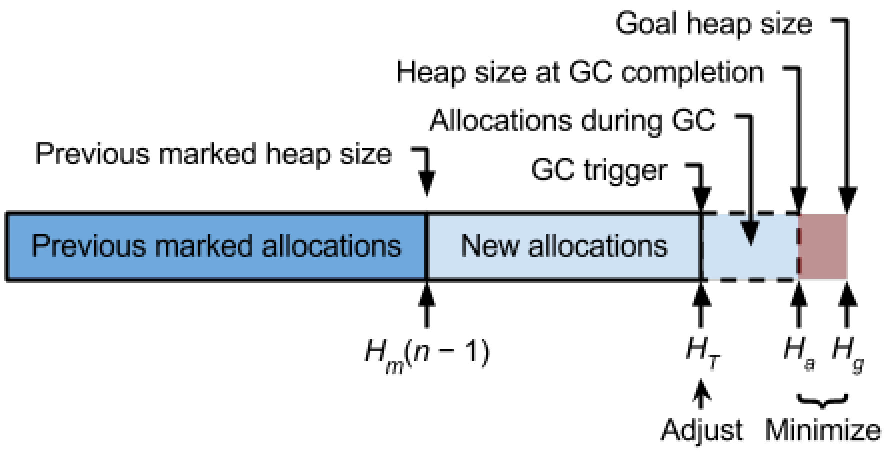

Figure 2. Concurrent garbage collection heap metrics.

A Go user expresses the desired maximum heap growth by setting GOGC to percent heap growth from one garbage collection cycle to the next. Let $h_g =$ GOGC$/100$ denote this *goal growth ratio*. That is, if $H_m(n)$ is the total size of marked objects following the nth GC cycle, then the goal heap size is $H_g(n) = H_m(n-1) \cdot (1 + h_g)$. Throughout this document, we will use the convention that $H_\square(n) = H_m(n-1) \cdot (1 + h_\square(n))$ is the absolute heap size for the heap growth ratio $h_\square$. Pacing must optimize for each cycle to terminate when the actual heap growth at the end of the cycle (prior to sweeping) $h_a(n)$ is as close as possible to the GOGC goal, as shown in figure 2.

Goal 1. Minimize $|h_g - h_a(n)|$.

A cycle may under- or overshoot h_g, its goal heap growth. Pacing must minimize heap growth overshoot to avoid consuming more memory than desired. At the same time, pacing must minimize heap growth undershoot because regularly undershooting means GC is running too often, consuming more total CPU than intended, and slowing down the overall application.

In a STW collector, this goal is achieved by simply running the collector when allocated heap growth reaches h_g. In a concurrent collector, the runtime must trigger garbage collection before this point. $h_T(n)$ denotes this *trigger growth ratio*, which the runtime will adjust to achieve its pacing goals.

Pacing must also optimize scheduling to achieve the desired garbage collector CPU utilization. CPU utilization by the garbage collector during concurrent phases should be as close to 25% of GOMAXPROCS as possible. This includes time in the background collector and assists from the mutator, but not time in write barriers (simply because the accounting would increase write barrier overhead) or secondary effects like increased cache misses. Of course, if the mutator is using less than 75% CPU, the garbage collector will run in the remaining time; this idle utilization does not count against the GC CPU budget. Let $u_g = 0.25$ denote this goal utilization and $u_a(n)$ be the actual average CPU utilization achieved by the nth GC cycle (not including idle utilization).

Goal 2. Minimize $\left|u_g - u_a(n)\right|$.

As with heap size, a cycle may under- or overutilize the CPU. 25% maximum utilization is a stated goal for the Go 1.5 collector, so pacing should minimize CPU overutilization. However, this is a soft limit and necessarily so: if the runtime were to strictly enforce a 25% utilization limit, a rapidly allocating mutator can cause arbitrary heap overshoot. Pacing should also minimize CPU underutilization because using as much of the 25% budget as possible minimizes the duration of the concurrent mark phase. Since the concurrent mark phase enables write barriers, this in turn minimizes the impact of write barrier overhead on application performance. It also reduces the amount of *floating garbage*, objects that are kept by the collector because they were reachable at some point during GC but are not reachable at GC termination. The runtime will adjust how it schedules the CPU between mutators and background garbage collection to achieve the pacing goals.

Design

The design of GC pacing consists of four components: 1) an estimator for the amount of scanning work a GC cycle will require, 2) a mechanism for mutators to perform the estimated amount of scanning work by the time heap allocation reaches the heap goal, 3) a scheduler for background scanning when mutator assists underutilize the CPU budget, and 4) a proportional controller for the GC trigger.

The design balances two different views of time: *CPU time* and *heap time*. CPU time is like standard wall clock time, but passes GOMAXPROCS times faster. That is, if GOMAXPROCS is 8, then eight CPU seconds pass every wall second and GC gets two seconds of CPU time every wall second. The CPU scheduler manages CPU time. The passage of heap time is measured in bytes and moves forward as mutators allocate. The relationship between heap time and wall time depends on the allocation rate and can change constantly. Mutator assists manage the passage of heap time, ensuring the estimated scan work has been completed by the time the heap reaches the goal size. Finally, the trigger controller creates a feedback loop that ties these two views of time together, optimizing for both heap time and CPU time goals.

Scan work estimator

Because Go 1.5's collector is a mark-sweep collector, the CPU time consumed by the concurrent mark phase is dominated by scanning, the process of greying and subsequently blackening objects. Hence, to pace the collector, the runtime needs to estimate the amount of work W_e that will be performed by scanning.

Scanning time is roughly linear in the number of pointer slots scanned, so we measure scan work in scanned pointer slots. Alternatively, scan work could be estimated in total bytes scanned, including non-pointer bytes prior to the last pointer slot of an object (scanning stops after the last pointer slot). We choose to measure only scanning of pointer slots because this is more computationally expensive and far more likely to cause cache misses. We may revise scan work to count both, but to assign more weight to pointer slots.

The actual scan work $W_a(n)$ performed by the nth cycle may vary significantly from cycle to cycle with heap size. Hence, similar to the heap trigger and the heap goal, the garbage collector will track the scan work from cycle to cycle as a ratio $w = W/H_m$ of pointers per marked heap byte, which should be much more stable.

There are several possible approaches to estimating w and finding a good estimator will likely require some experimentation with real workloads. The worst-case estimator is 1/*pointer size*—the entire reachable heap is pointers—but this is far too pessimistic. A better estimator is the scan work performed by the previous garbage collection cycle. However, this may be too sensitive to transient changes in heap topology. Hence, to smooth this out, we will use an exponentially weighted moving average (EWMA) of scan work ratios of recent cycles,

$$w(n+1) = K_w \frac{W_a(n)}{H_m(n)} + (1 - K_w)w(n)$$

where K_w is the weighting coefficient. We'll start with $K_w = 0.75$ and tune this if necessary. At the beginning of each cycle, the garbage collector will estimate the scan work $W_e(n)$ for that cycle using this scan work ratio estimate and the marked heap size of the previous cycle as an estimate of the reachable heap in this cycle:

$$W_e(n) = w(n)H_m(n-1)\,.$$

If this proves insufficient, it should be possible to use more sophisticated models to account for trends and patterns. It may also be possible to revise the scan work estimate as collection runs, at least if it discovers more scan work than the current estimate.

Mutator assists

With only background garbage collection, a mutator may allocate faster than the garbage collector can mark. At best, this causes the heap to always overshoot and saturates the trigger point h_t at 0. At worst, this leads to unbounded heap growth.

To address this, the garbage collector will enlist the help of the mutator since allocation by the mutator is what causes the heap size to approach (and potentially exceed) the maximum heap size. Hence, allocation can assist the garbage collector by performing scanning work proportional to the size of the allocation. Let $A(x,n)$ denote the assist scan work that should be performed by an allocation of x bytes during the nth GC cycle.

The *ideal assist work* is

$$A(x,n) = \frac{W_e(n)}{H_g(n) - H_T(n)} \cdot x \, .$$

For example, if pointers are 8 bytes, the current scan work estimate W_e is 1GB/8, the trigger point H_T is 1.5GB and the heap size goal H_g is 2GB, then $A(x,n) = 0.25x$, so every 4 bytes of allocation will scan 1 pointer. Without background garbage collection, when the allocated heap size reaches 2GB, mutator assists will have performed exactly 1GB worth of scanning work. If W_e is accurate, then collection will finish at exactly the target heap size.

However, mutator assists alone may underutilize the GC CPU budget, so the collector must perform background collection in addition to mutator assists. Work performed by background collection is not accounted for above. Hence, rather than unconditionally performing $A(x,n)$ scan work per allocation, the collector will use a system of *work credit* in which scanning u pointers creates u units of credit. The background garbage collector continuously creates work credits as it scans. Mutator allocation creates $A(x,n)$ units of *work deficit*, which the mutator can correct by either stealing credit from the background collector (as long as this doesn't put the background collector into debt) or by performing its own scanning work.

This system of work credit is quite flexible. For example, it's difficult to scan exactly $A(x,n)$ pointer slots since scanning is done an object at a time, but this approach lets a mutator accumulate credit for additional scanning work that it can absorb in later allocations. We can also reduce contention by adding hysteresis: allowing a mutator to accumulate a small amount of deficit without scanning.

CPU scheduling

Mutator assists alone may under- or overutilize the GC CPU budget, depending on the mutator allocation rate. Both situations are undesirable.

To address underutilization, the runtime will track CPU time spent in mutator assists and background collection since the beginning of the concurrent mark phase. If this is below the 25% budget, it will schedule the background garbage collector thread in order to bring it up to 25%. This indirectly helps smooth out transient overutilization as well. If mutator assists briefly surpass the 25% budget, the scheduler will not run the background collector until the average comes back down below 25%. Likewise, if the background collector has built up work credit, mutator assists that would exceed the 25% budget without background credit are more likely to consume the background credit and not expend CPU time on scanning work.

However, the CPU scheduler does not address long-term overutilization, as limiting mutator assists would allow rapidly allocating mutators to grow the heap arbitrarily. Instead, this is handled by the trigger ratio controller.

Trigger ratio controller

While the runtime has direct and continuous control over GC CPU utilization, it has only indirect control over h_a, the heap growth when GC completes. Given constraints on GC CPU utilization, this indirect control comes primarily from when the runtime decides to start a GC cycle, h_T.

The appropriate value of h_T to avoid heap under- or overshoot depends on several factors that will vary between applications and during execution. Hence, the runtime will use a proportional controller to adapt h_T after every garbage collection:

$$h_T(n+1) = \max(0,\ h_T(n) + K_T e(n))$$

where $K_T \in [0,1]$ is the trigger controller's proportional gain and $e(n)$ is the error term as a heap ratio delta. The value of $h_T(0)$ is unlikely to have significant impact. Based on current heuristics, we'll set $h_T(0) = 7/8$ and adjust if this is too aggressive. K_T may also require some tuning. We'll start with $K_T = 0.5$.

This leaves the error term. Perhaps the obvious way would be to adjust h_T according to how much the heap over- or undershot, $e^*(n) = h_g - h_a(n)$. However, this doesn't account for CPU utilization, which leads to instability: if the heap undershoots, this will increase the trigger size, which will increase the amount of scanning work done by mutator assists per allocation byte, increasing the GC CPU utilization and probably causing the heap to undershoot again.

Instead, the runtime will adjust h_T based on an estimate of what the heap growth *would have been* if GC CPU utilization was $u_g = 0.25$. This leads to the error term

$$e(n) = h_g - h_T(n) - \tfrac{u_a(n)}{u_g}(h_a(n) - h_T(n)).$$

The details of deriving this equation are in appendix A. Note that this reduces to the simpler error term above, e^*, if CPU utilization is exactly the goal utilization; that is, if $u_a(n) = u_g$. Otherwise, it uses a *scaled heap growth ratio* to account for CPU over/underutilization; for example, if utilization is 50%, this assumes the heap would have grown twice as much during garbage collection if utilization were limited to 25%.

Combined with mutator assists and CPU scheduling, the trigger ratio controller creates a feedback loop that couples CPU utilization and heap growth optimization to achieve the optimization goals. If the trigger is too high, mutator assists will handle the estimated scan work by the time heap size reaches the heap goal, but will force GC CPU utilization over 25%. As a result, the scaled heap growth in the error term will exceed the heap goal, so the trigger controller will decrease the trigger for the next cycle. This will spread the assist scan work over a longer period of heap growth in the next cycle, decreasing its GC CPU utilization. On the other hand, if the trigger is too low, CPU utilization from mutator assists will be low, so the CPU scheduler will schedule background GC to ensure utilization is at least 25%. This will cause the heap to undershoot, and because utilization was forced to 25%, the error will simply be the difference between the actual heap growth and the goal, causing the trigger controller to increase the trigger for the next cycle.

Appendix A. Derivation of the trigger controller error term

We start with the assumption that GC CPU utilization and heap growth during the concurrent mark phase are inversely proportional. Using this, the estimated heap size X supposing GC CPU utilization had been u_g instead of u_a is

$$(X - H_T(n)) \cdot u_g = (H_a(n) - H_T(n)) \cdot u_a(n)$$
$$\Rightarrow X = \tfrac{u_a(n)}{u_g}(H_g(n) - H_T(n)) - H_T(n)\,.$$

Given this, the absolute error in the trigger heap size is,

$$E(n) = H_g(n) - X\,.$$

Using the relation between absolute heap sizes and heap growth ratios,

$$H_{\square}(n) = H_m(n-1) \cdot (1 + h_{\square}(n))$$

or

$$h_{\square}(n) = \tfrac{H_{\square}(n)}{H_m(n-1)} - 1\,,$$

we can translate this into a trigger ratio error,

$$\begin{aligned}
e(n) &= \left(\tfrac{H_g(n)}{H_m(n-1)} - 1\right) - \left(\tfrac{X}{H_m(n-1)} - 1\right) \\
&= \tfrac{1}{H_m(n-1)}\,(H_g(n) - X) \\
&= \tfrac{1}{H_m(n-1)}\left(H_g(n) - H_T(n) - \tfrac{u_a(n)}{u_g}\,(H_g(n) - H_T(n))\right) \\
&= (1 + h_g) - (1 + h_T(n)) - \tfrac{u_a(n)}{u_g}\,((1 + h_a(n)) - (1 + h_T(n))) \\
&= h_g - h_T(n) - \tfrac{u_a(n)}{u_g}(h_a(n) - h_T(n))\,.
\end{aligned}$$