**The Setup**

Go is building a garbage collector (GC) not only for 2015 but for 2025 and beyond: A GC that supports today’s software development and scales along with new software and hardware throughout the next decade. Such a future has no place for stop-the-world GC pauses, which have been an impediment to broader uses of safe and secure languages such as Go.

Go 1.5, the first glimpse of this future, achieves GC latencies well below the 10 millisecond goal we set a year ago. We presented some impressive numbers in [a talk at Gophercon](https://talks.golang.org/2015/go-gc.pdf). The latency improvements have generated a lot of attention; Robin Verlangen’s blog post [*Billions of requests per day meet Go 1.5*](https://medium.com/@robin.verlangen/billions-of-request-per-day-meet-go-1-5-362bfefa0911) validates our direction with end to end results. We also particularly enjoyed [Alan Shreve’s production server graphs](https://twitter.com/inconshreveable/status/620650786662555648) and his "Holy 85% reduction" comment.

Today 16 gigabytes of RAM costs $100 and CPUs come with many cores, each with multiple hardware threads. In a decade this hardware will seem quaint but the software being built in Go today will need to scale to meet expanding needs and the next big thing. Given that hardware will provide the power to increase throughput, Go’s garbage collector is being designed to favor low latency and tuning via only a single knob. Go 1.5 is the first big step down this path and these first steps will forever influence Go and the applications it best supports. This blog post gives a high-level overview of what we have done for the Go 1.5 collector.

**The Embellishment**

To create a garbage collector for the next decade, we turned to an algorithm from decades ago. Go's new garbage collector is a *concurrent*, *tri-color*, *mark-sweep* collector, an idea first proposed by [Dijkstra in 1978](http://dl.acm.org/citation.cfm?id=359655). This is a deliberate divergence from most "enterprise" grade garbage collectors of today, and one that we believe is well suited to the properties of modern hardware and the latency requirements of modern software.

In a tri-color collector, every object is either white, grey, or black and we view the heap as a graph of connected objects. At the start of a GC cycle all objects are white. The GC visits all *roots*, which are objects directly accessible by the application such as globals and things on the stack, and colors these grey. The GC then chooses a grey object, blackens it, and then scans it for pointers to other objects. When this scan finds a pointer to a white object, it turns that object grey. This process repeats until there are no more grey objects. At this point, white objects are known to be unreachable and can be reused.

This all happens concurrently with the application, known as the *mutator*, changing pointers while the collector is running. Hence, the mutator must maintain the invariant that no black object points to a white object, lest the garbage collector lose track of an object installed in a part of the heap it has already visited. Maintaining this invariant is the job of the *write barrier*, which is a small function run by the mutator whenever a pointer in the heap is modified. Go’s write barrier colors the now-reachable object grey if it is currently white, ensuring that the garbage collector will eventually scan it for pointers.

Deciding when the job of finding all grey objects is done is subtle and can be expensive and complicated if we want to avoid blocking the mutators. To keep things simple Go 1.5 does as much work as it can concurrently and then briefly stops the world to inspect all potential sources of grey objects. Finding the sweet spot between the time needed for this final stop-the-world and the total amount of work that this GC does is a major deliverable for Go 1.6.

Of course the devil is in the details. When do we start a GC cycle? What metrics do we use to make that decision? How should the GC interact with the Go scheduler? How do we pause a mutator thread long enough to scan its stack?  How do we represent white, grey, and black so we can efficiently find and scan grey objects? How do we know where the roots are? How do we know where in an object pointers are located? How do we minimize memory fragmentation? How do we deal with cache performance issues? How big should the heap be? And on and on, some related to allocation, some to finding reachable objects, some related to scheduling, but many related to performance. Low-level discussions of each of these areas are beyond the scope of this blog post.

At a higher level, one approach to solving performance problems is to add GC knobs, one for each performance issue. The programmer can then turn the knobs in search of appropriate settings for their application. The downside is that after a decade with one or two new knobs each year you end up with the GC Knobs Turner Employment Act. Go is not going down that path. Instead we provide a single knob, called GOGC. This value controls the total size of the heap relative to the size of reachable objects. The default value of 100 means that total heap size is now 100% bigger than (i.e., twice) the size of the reachable objects after the last collection. 200 means total heap size is 200% bigger than (i.e., three times) the size of the reachable objects. If you want to lower the total time spent in GC, increase GOGC. If you want to trade more GC time for less memory, lower GOGC.

More importantly as RAM doubles with the next generation of hardware, simply doubling GOGC will halve the number of GC cycles. On the other hand since GOGC is based on reachable object size, doubling the load by doubling the reachable objects requires no retuning. The application just scales. Furthermore, unencumbered by ongoing support for dozens of knobs, the runtime team can focus on improving the runtime based on feedback from real customer applications.

**The Punchline**

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. It’s a good place to be for the next decade and beyond. For more details about the 1.5 GC and how we eliminated latency issues see the [Go GC: Latency Problem Solved presentation](https://www.youtube.com/watch?v=aiv1JOfMjm0) or [the slides](https://talks.golang.org/2015/go-gc.pdf).

*By Richard Hudson*

