The Go Blog

Getting to Go: The Journey of Go's Garbage Collector

12 July 2018

This is the transcript from the keynote I gave at the International Symposium on Memory Management (ISMM) on June 18, 2018. For the past 25 years ISMM has been the premier venue for publishing memory management and garbage collection papers and it was an honor to have been invited to give the keynote.

Next article

Portable Cloud Programming with Go Cloud

Previous article

Updating the Go Code of Conduct

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go on Google+
Go+ Community
Go on Twitter

Blog index

Abstract

The Go language features, goals, and use cases have forced us to rethink the entire garbage collection stack and have led us to a surprising place. The journey has been exhilarating. This talk describes our journey. It is a journey motivated by open source and Google's production demands. Included are side hikes into dead end box canyons where numbers guided us home. This talk will provide insight into the how and the why of our journey, where we are in 2018, and Go's preparation for the next part of the journey.

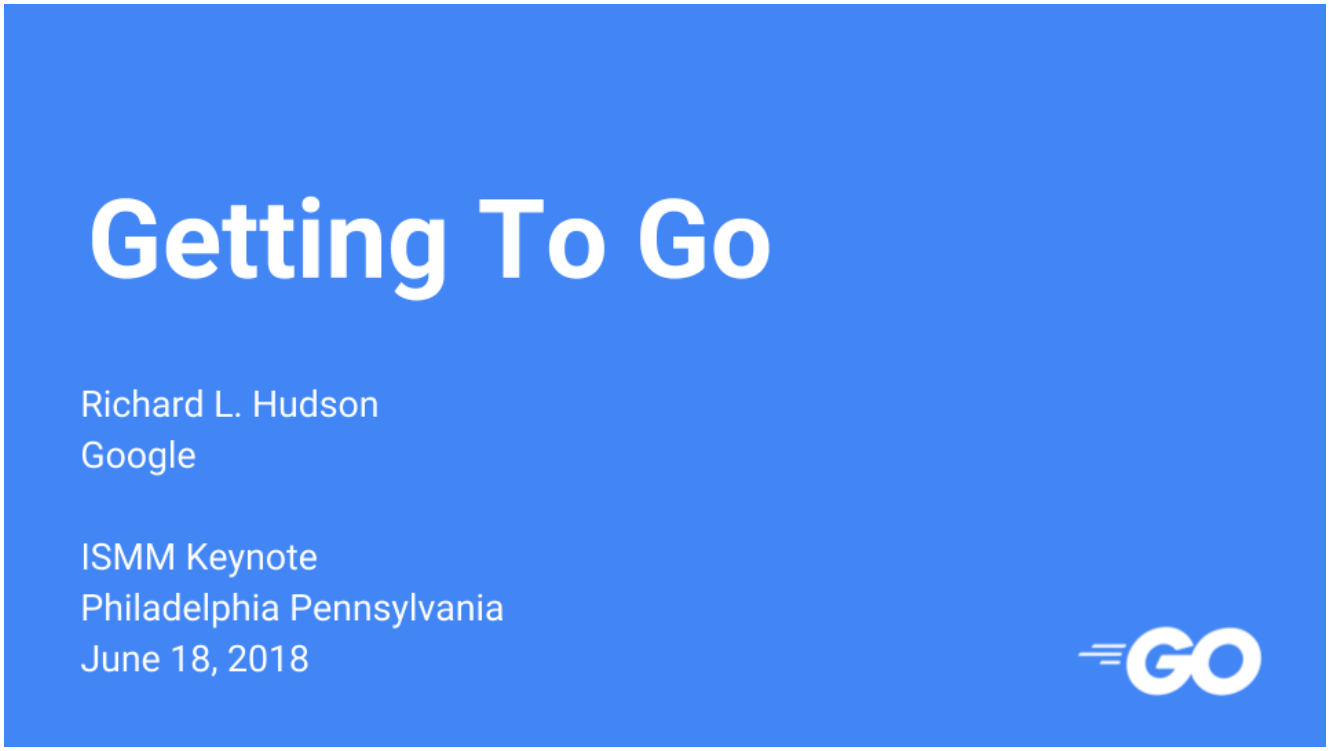
Bio

Richard L. Hudson (Rick) is best known for his work in memory management including the invention of the Train, Sapphire, and Mississippi Delta algorithms as well as GC stack maps which enabled garbage collection in statically typed languages such as Modula-3, Java, C#, and Go. Rick is currently a member of Google's Go team where he is working on Go's garbage collection and runtime issues.

Contact: rlh@golang.org

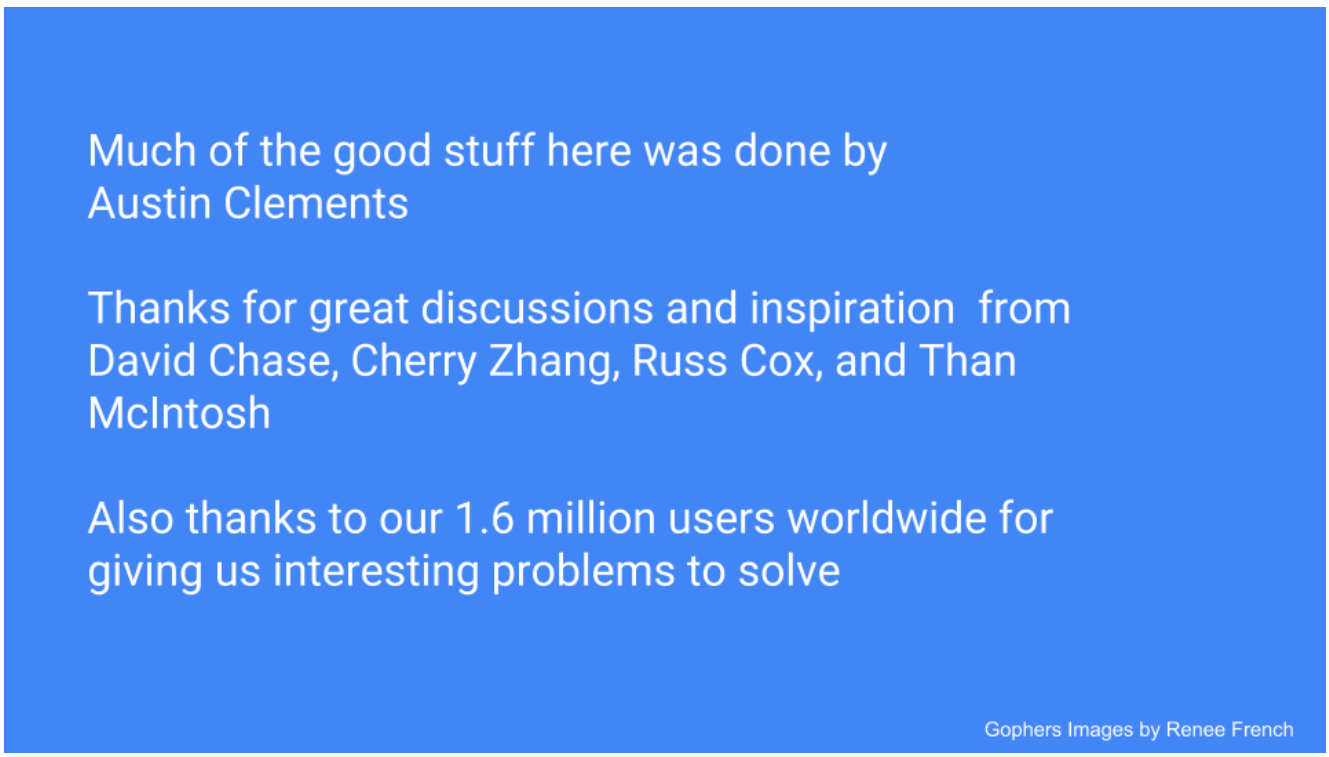
Comments: See the discussion on golang-dev.

The Transcript



Rick Hudson here.

This is a talk about the Go runtime and in particular the garbage collector. I have about 45 or 50 minutes of prepared material and after that we will have time for discussion and I'll be around so feel free to come up afterwards.



Before I get started I want to acknowledge some people.

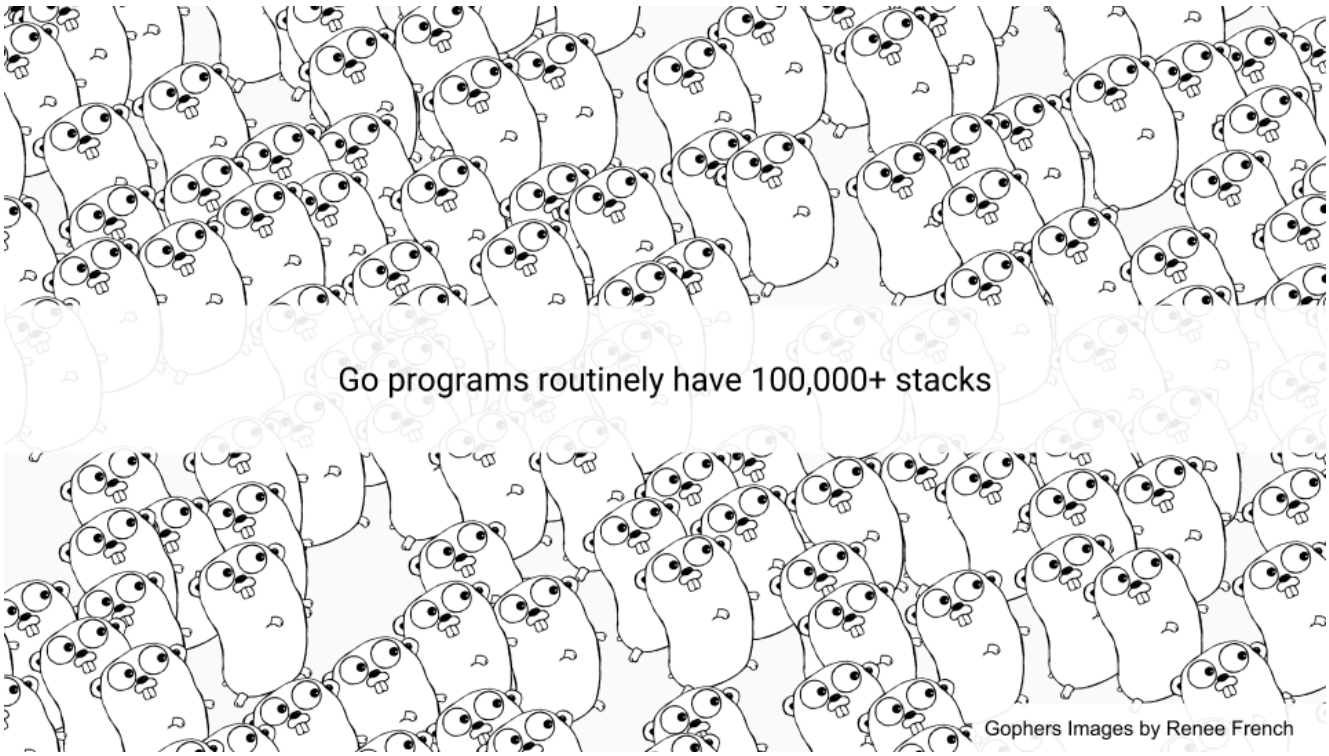
A lot of the good stuff in the talk was done by Austin Clements. Other people on the Cambridge Go team, Russ, Than, Cherry, and David have been an engaging, exciting, and fun group to work with.

We also want to thank the 1.6 million Go users worldwide for giving us interesting problems to solve. Without them a lot of these problems would never come to light.

And finally I want to acknowledge Renee French for all these nice Gophers that she has been producing over the years. You will see several of them throughout the talk.

GC's view of Go

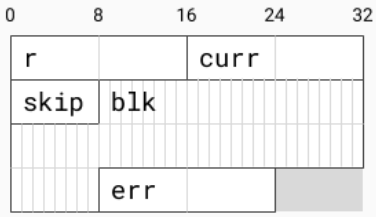
Before we get revved up and going on this stuff we really have to show what GC's view of Go looks like.



Well first of all Go programs have hundreds of thousands of stacks. They are managed by the Go scheduler and are always preempted at GC safepoints. The Go scheduler multiplexes Go routines onto OS threads which hopefully run with one OS thread per HW thread. We manage the stacks and their size by copying them and updating pointers in the stack. It's a local operation so it scales fairly well.

Go is value-oriented

```
type Reader struct { // archive/tar
    r    io.Reader
    curr numBytesReader
    skip int64
    blk  block
    err  error
}
type block [64]byte
```



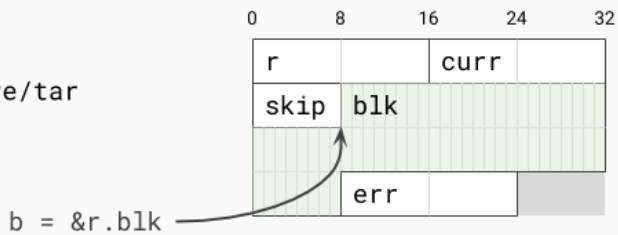
The next thing that is important is the fact that Go is a value-oriented language in the tradition of C-like systems languages rather than reference-oriented language in the tradition of most managed runtime languages. For example, this shows how a type from the tar package is laid out in memory. All of the fields are embedded directly in the Reader value. This gives programmers more control over memory layout when they need it. One can collocate fields that have related values which helps with cache locality.

Value-orientation also helps with the foreign function interfaces. We have a fast FFI with C and C++. Obviously Google has a tremendous number of facilities available but they are written in C++. Go couldn't wait to reimplement all of these things in Go so Go had to have access to these systems through the foreign function interface.

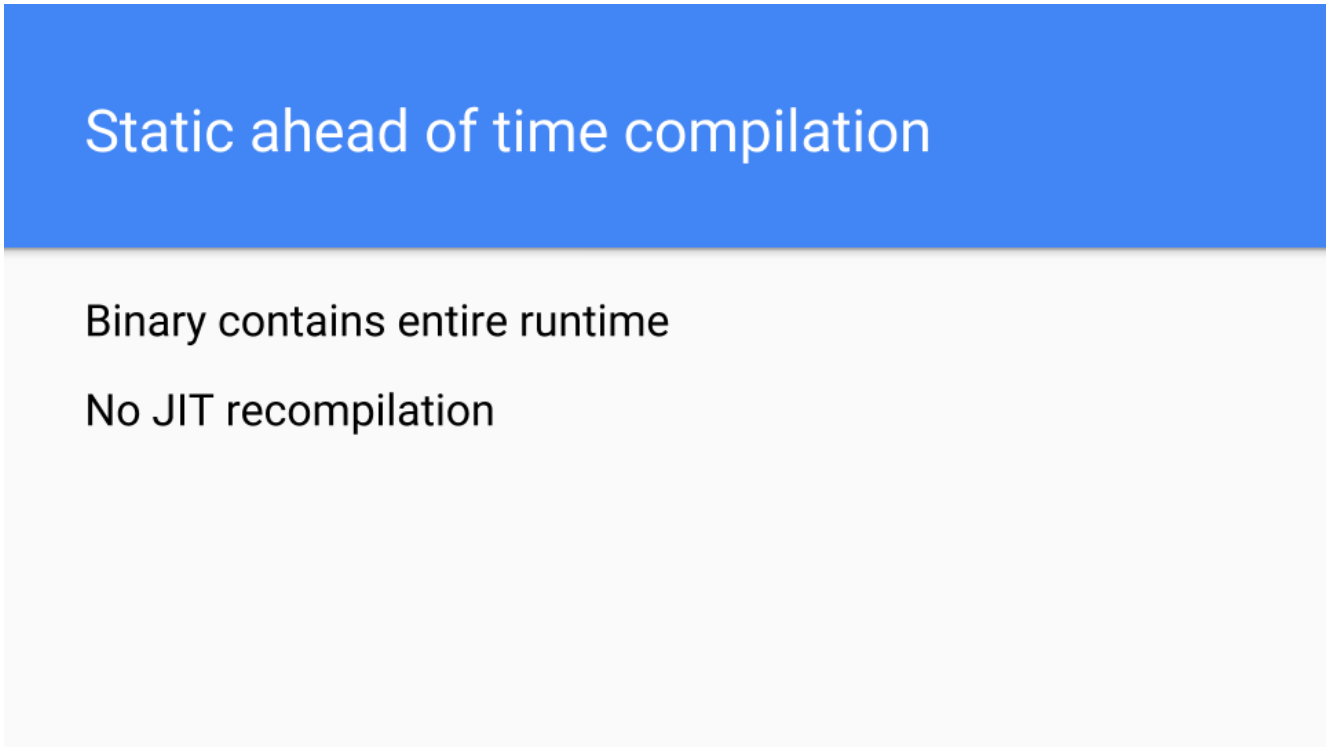
This one design decision has led to some of the more amazing things that have to go on with the runtime. It is probably the most important thing that differentiates Go from other GCed languages.

Go allows interior pointers

```
type Reader struct { // archive/tar
    r    io.Reader
    curr numBytesReader
    skip int64
    blk  block
    err  error
}
type block [64]byte
```



Of course Go can have pointers and in fact they can have interior pointers. Such pointers keep the entire value live and they are fairly common.

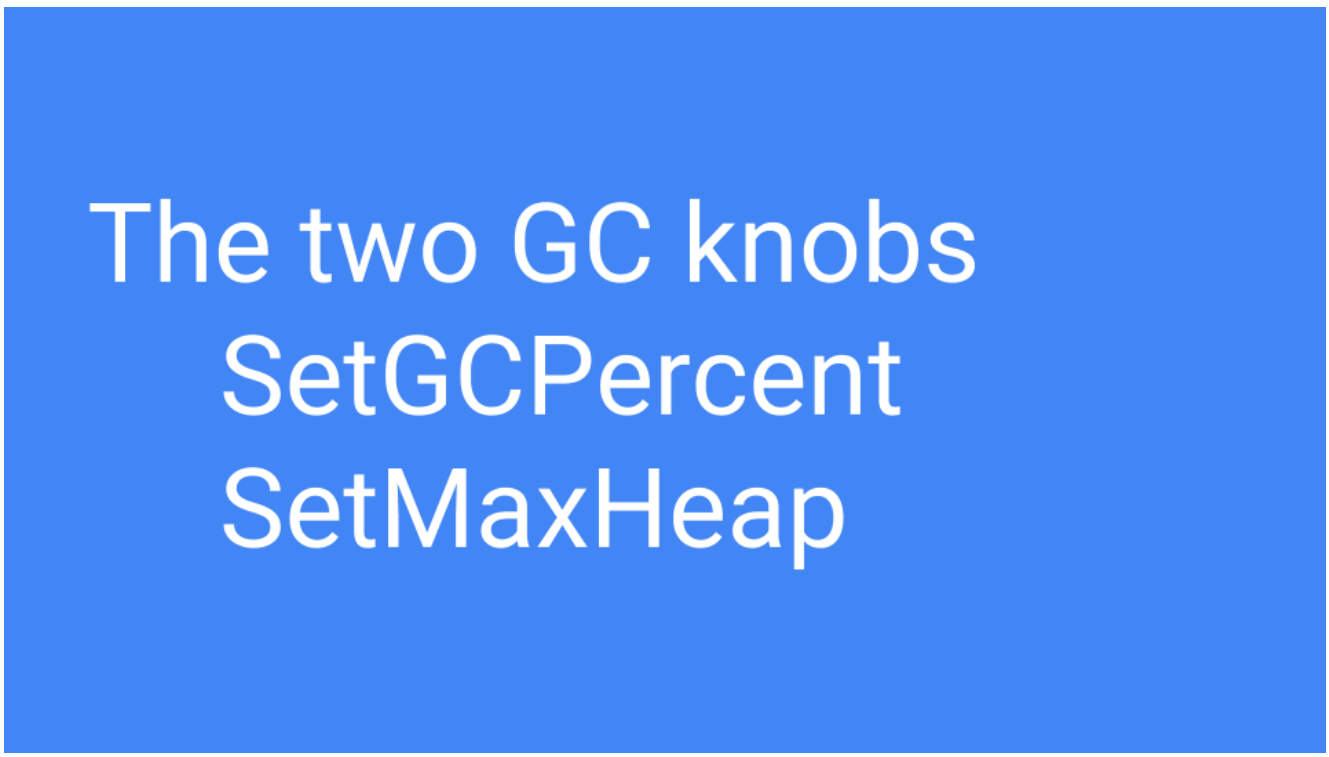


We also have a way ahead of time compilation system so the binary contains the entire runtime.

There is no JIT recompilation. There are pluses and minuses to this. First of all, reproducibility of program execution is a lot easier which makes moving forward with compiler improvements much faster.

On the sad side of it we don't have the chance to do feedback optimizations as you would with a JITed system.

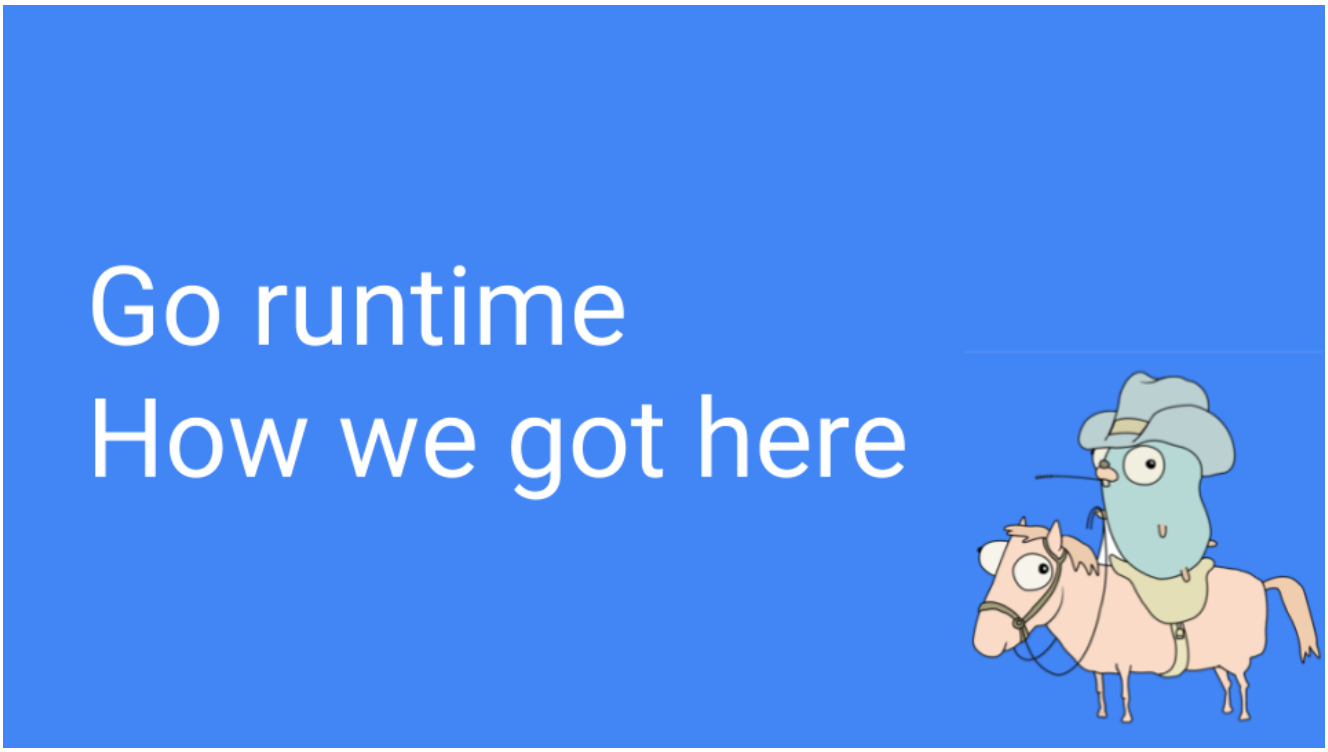
So there are pluses and minuses.



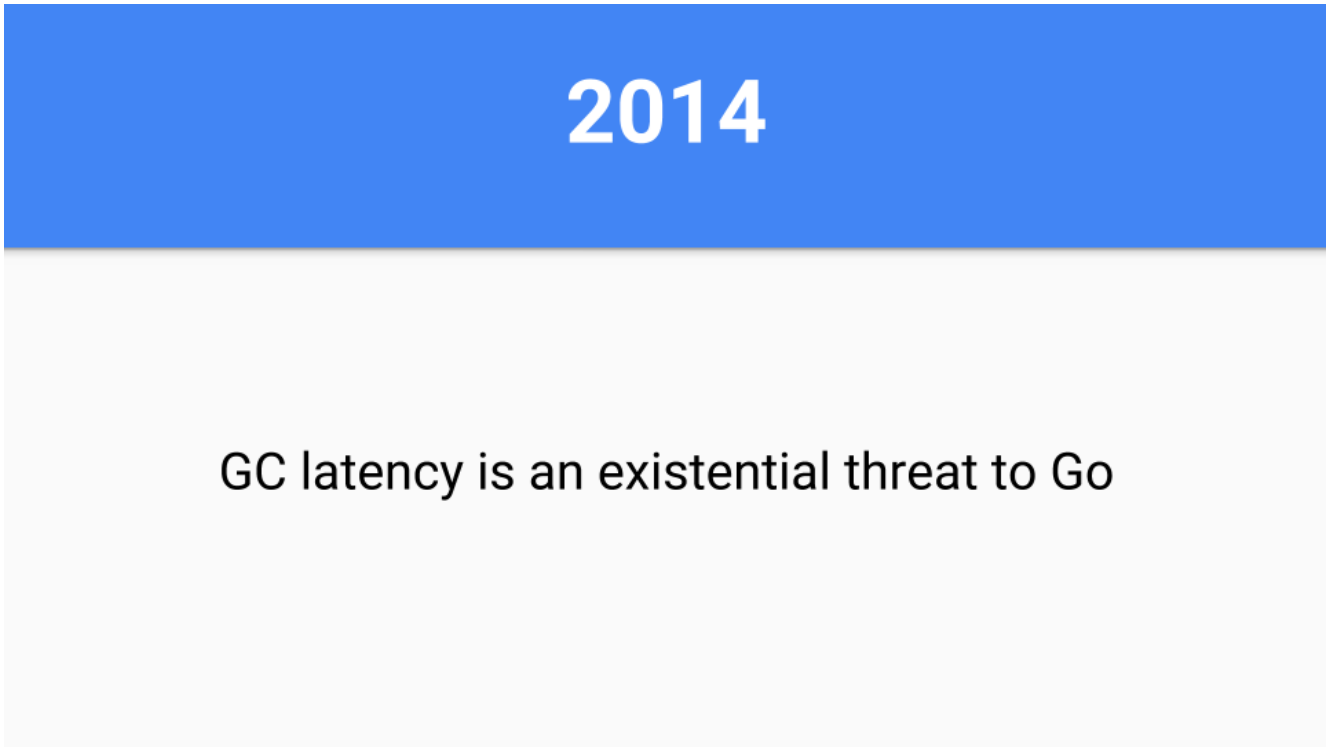
Go comes with two knobs to control the GC. The first one is GCPercent. Basically this is a knob that adjusts how much CPU you want to use and how much memory you want to use. The default is 100 which means that half the heap is dedicated to live memory and half the heap is dedicated to allocation. You can modify this in either direction.

MaxHeap, which is not yet released but is being used and evaluated internally, lets the programmer set what the maximum heap size should be. Out of memory, OOMs, are tough on Go; temporary spikes in memory usage should be handled by increasing CPU costs, not by aborting. Basically if the GC sees memory pressure it informs the application that it should shed load. Once things are back to normal the GC informs the application that it can go back to its regular load. MaxHeap also provides a lot more flexibility in scheduling. Instead of always being paranoid about how much memory is available the runtime can size the heap up to the MaxHeap.

This wraps up our discussion on the pieces of Go that are important to the garbage collector.



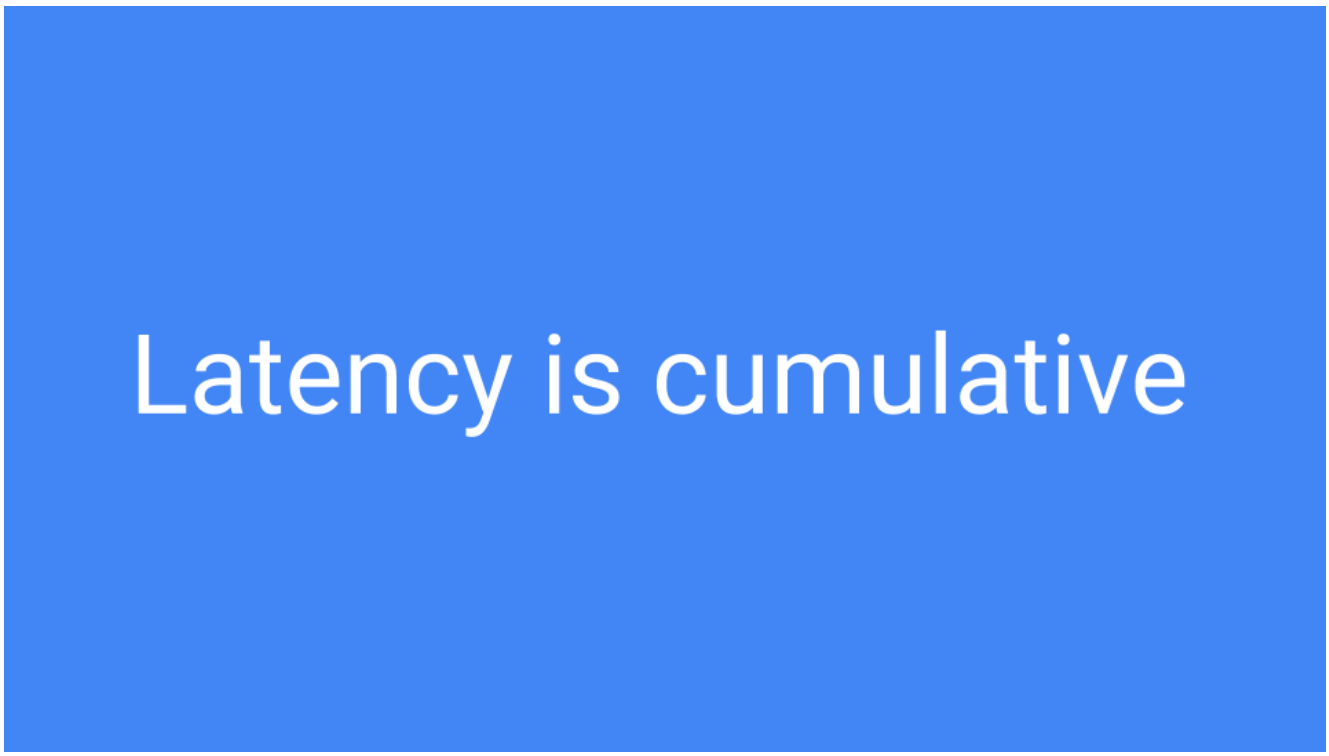
So now let's talk about the Go runtime and how did we get here, how we got to where we are.



So it's 2014. If Go does not solve this GC latency problem somehow then Go isn't going to be successful. That was clear.

Other new languages were facing the same problem. Languages like Rust went a different way but we are going to talk about the path that Go took.

Why is latency so important?



The math is completely unforgiving on this.

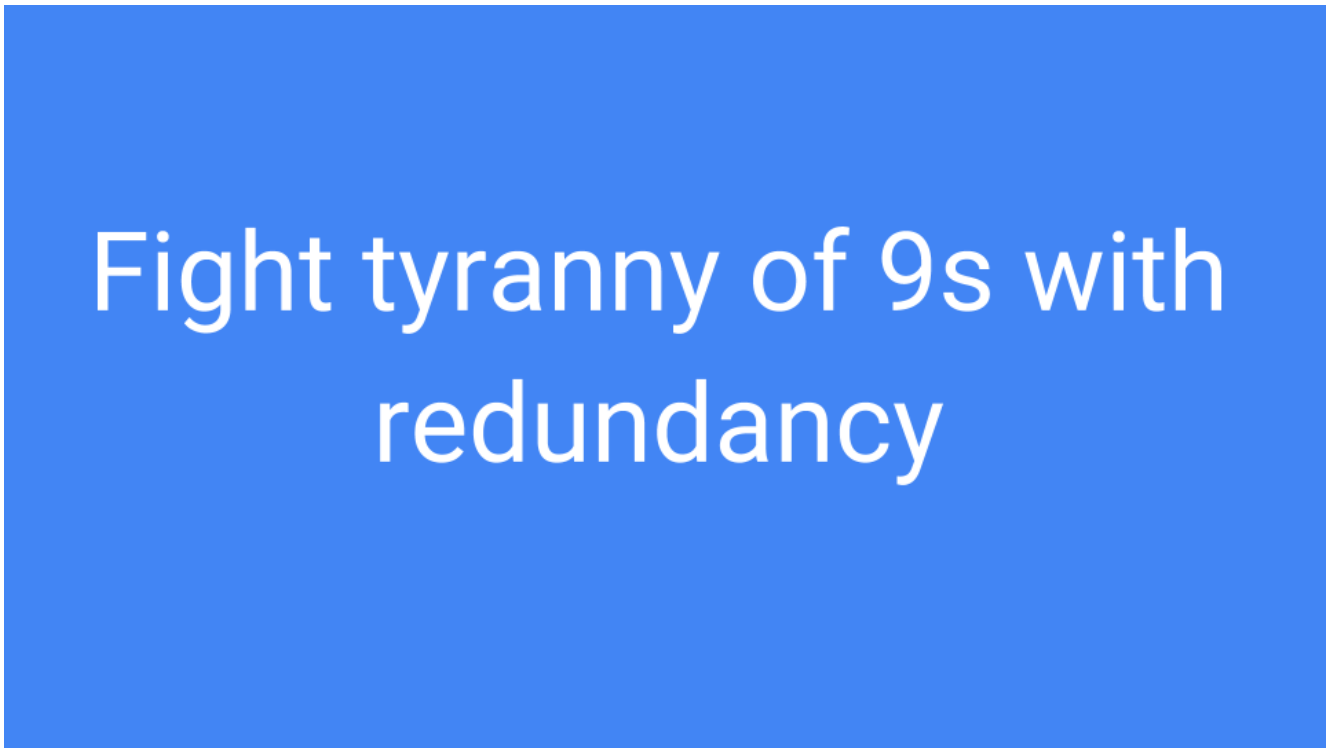
A 99%ile isolated GC latency service level objective (SLO), such as 99% of the time a GC cycle takes < 10ms, just simply doesn't scale. What matters is latency during an entire session or through the course of using an app many times in a day. Assume a session that browses several web pages ends up making 100 server requests during a session or it makes 20 requests and you have 5 sessions

packed up during the day. In that situation only 37% of users will have a consistent sub 10ms experience across the entire session.

If you want 99% of those users to have a sub 10ms experience, as we are suggesting, the math says you really need to target 4 9s or the 99.99%ile.

So it's 2014 and Jeff Dean had just come out with his paper called 'The Tail at Scale' which this digs into this further. It was being widely read around Google since it had serious ramifications for Google going forward and trying to scale at Google scale.

We call this problem the tyranny of the 9s.



So how do you fight the tyranny of the 9s?

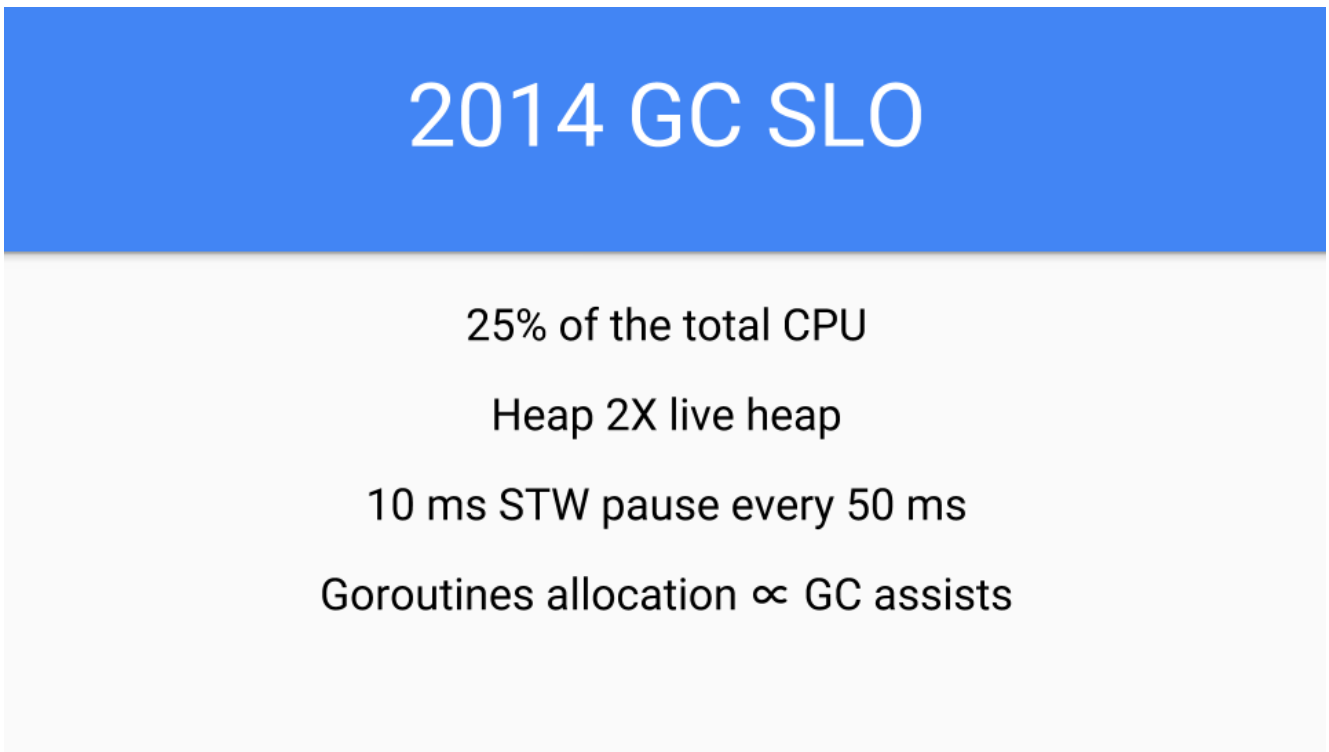
A lot of things were being done in 2014.

If you want 10 answers ask for several more and take the first 10 and those are the answers you put on your search page. If the request exceeds 50%ile reissue or forward the request to another server. If GC is about to run, refuse new requests or forward the requests to another server until GC is done. And so forth and so on.

All these are workarounds come from very clever people with very real problems but they didn't tackle the root problem of GC latency. At Google scale we had to tackle the root problem. Why?

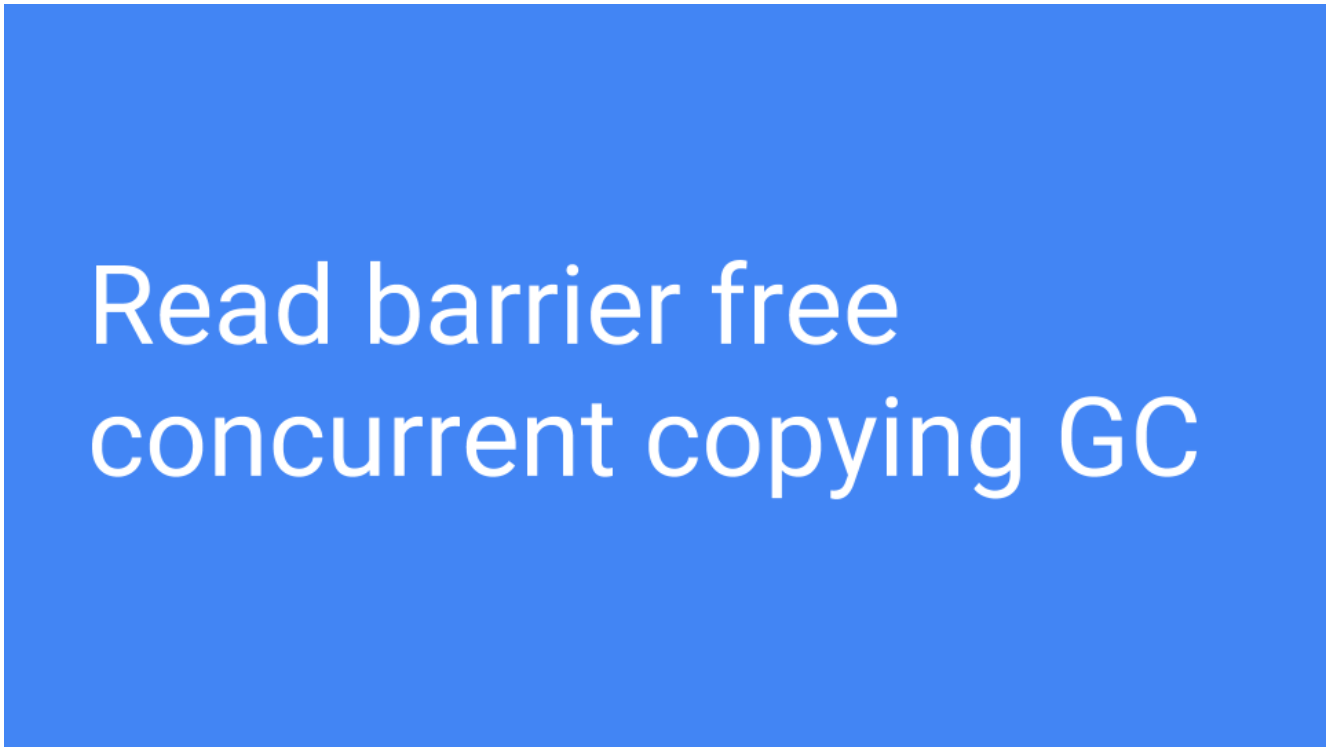
Redundancy wasn't going to scale, redundancy costs a lot. It costs new server farms.

We hoped we could solve this problem and saw it as an opportunity to improve the server ecosystem and in the process save some of the endangered corn fields and give some kernel of corn the chance to be knee high by the fourth of July and reach its full potential.



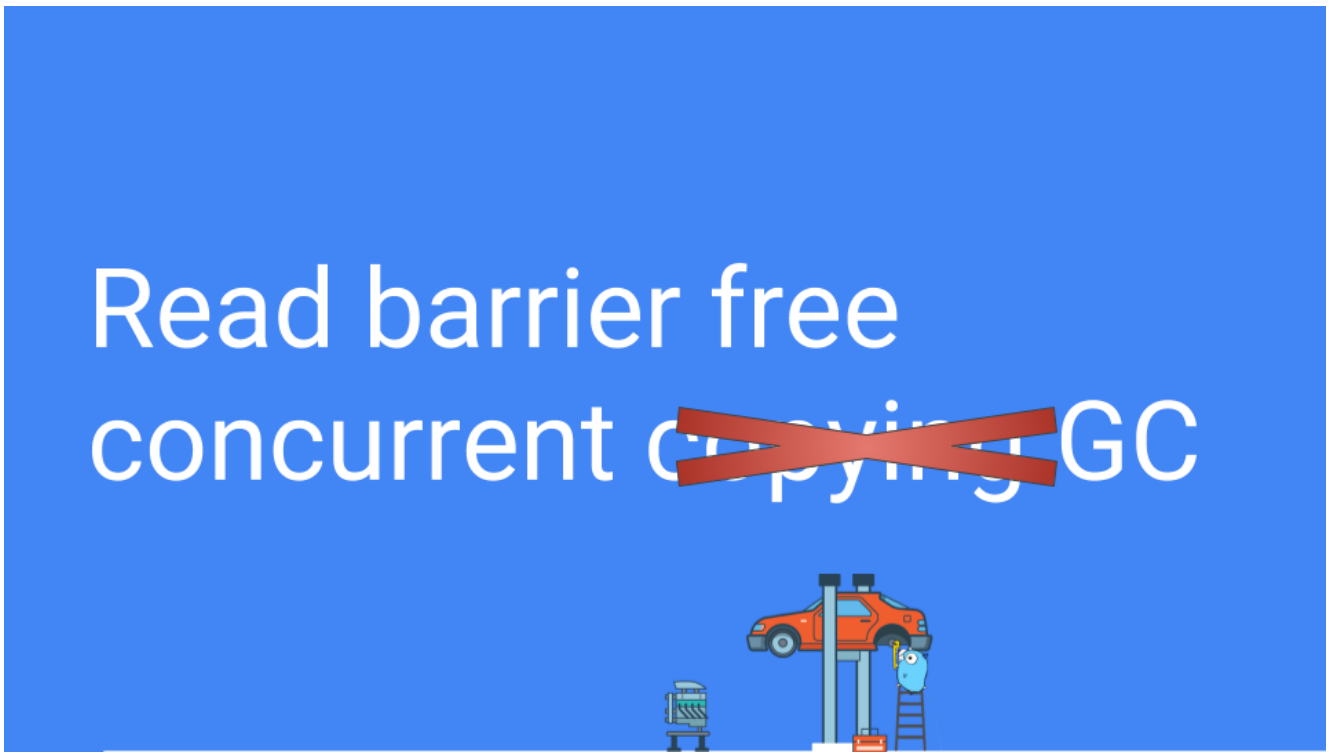
So here is the 2014 SLO. Yes, it was true that I was sandbagging, I was new on the team, it was a new process to me, and I didn't want to over promise.

Furthermore presentations about GC latency in other languages were just plain scary.



The original plan was to do a read barrier free concurrent copying GC. That was the long term plan. There was a lot of uncertainty about the overhead of read barriers so Go wanted to avoid them.

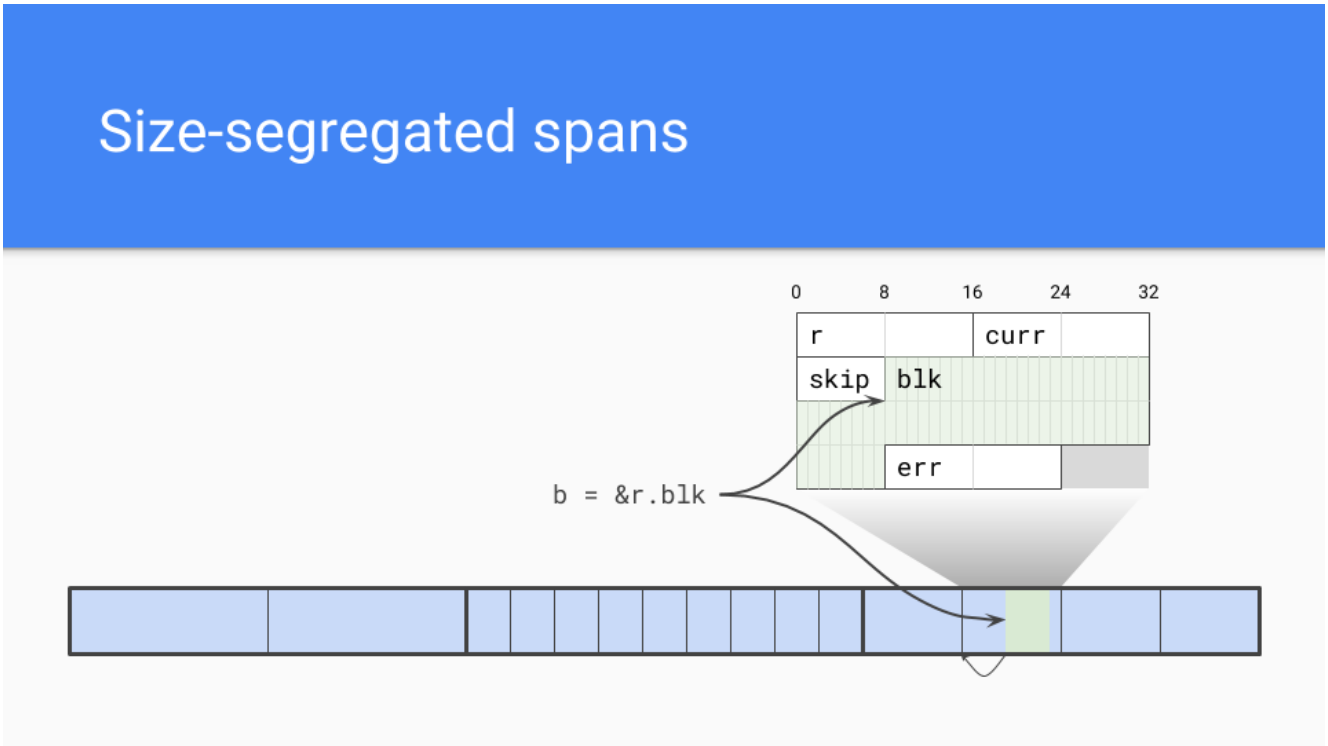
But short term 2014 we had to get our act together. We had to convert all of the runtime and compiler to Go. They were written in C at the time. No more C, no long tail of bugs due to C coders not understanding GC but having a cool idea about how to copy strings. We also needed something quickly and focused on latency but the performance hit had to be less than the speedups provided by the compiler. So we were limited. We had basically a year of compiler performance improvements that we could eat up by making the GC concurrent. But that was it. We couldn't slow down Go programs. That would have been untenable in 2014.



So we backed off a bit. We weren't going to do the copying part.

The decision was to do a tri-color concurrent algorithm. Earlier in my career Eliot Moss and I had done the journal proofs showing that Dijkstra's algorithm worked with multiple application threads. We also showed we could knock off the STW problems, and we had proofs that it could be done.

We were also concerned about compiler speed, that is the code the compiler generated. If we kept the write barrier turned off most of the time the compiler optimizations would be minimally impacted and the compiler team could move forward rapidly. Go also desperately needed short term success in 2015.



So let's look at some of the things we did.

We went with a size segregated span. Interior pointers were a problem.

The garbage collector needs to efficiently find the start of the object. If it knows the size of the objects in a span it simply rounds down to that size and that will be the start of the object.

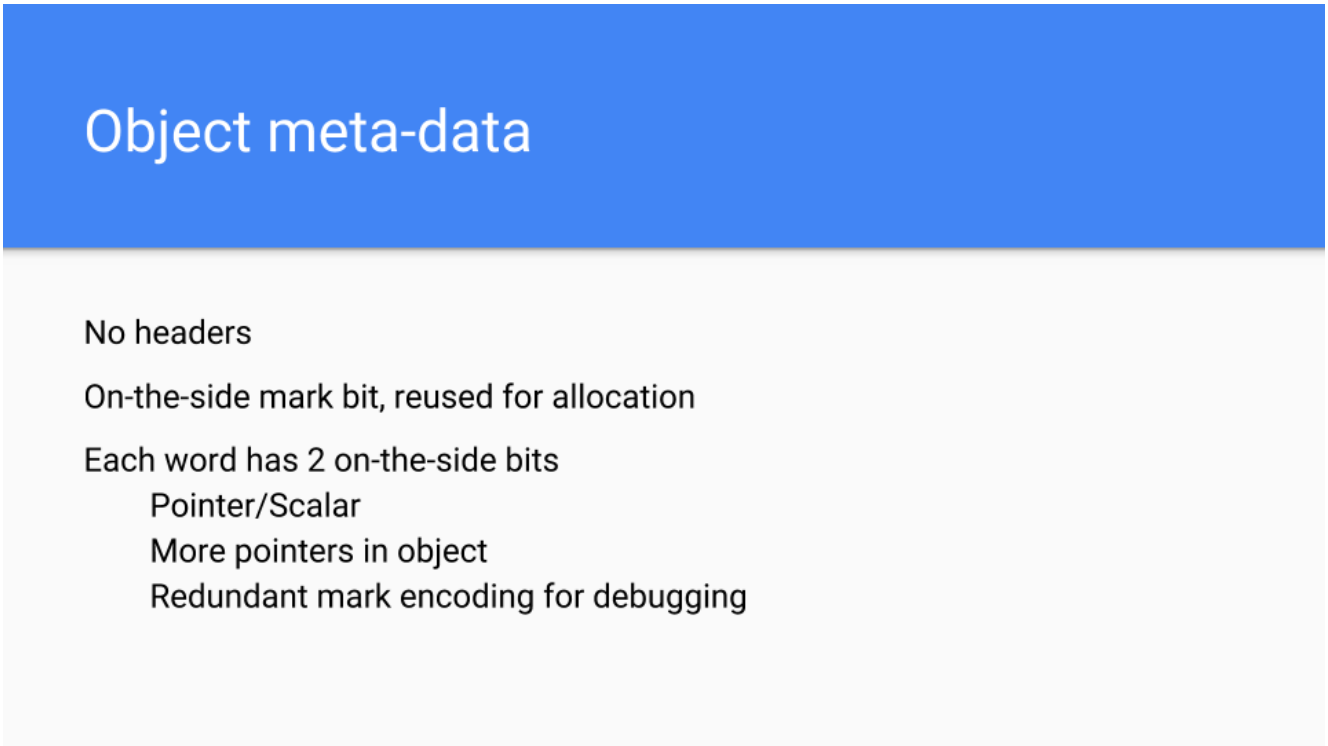
Of course size segregated spans have some other advantages.

Low fragmentation: Experience with C, besides Google's TCMalloc and Hoard, I was intimately involved with Intel's Scalable Malloc and that work gave us confidence that fragmentation was not going to be a problem with non-moving allocators.

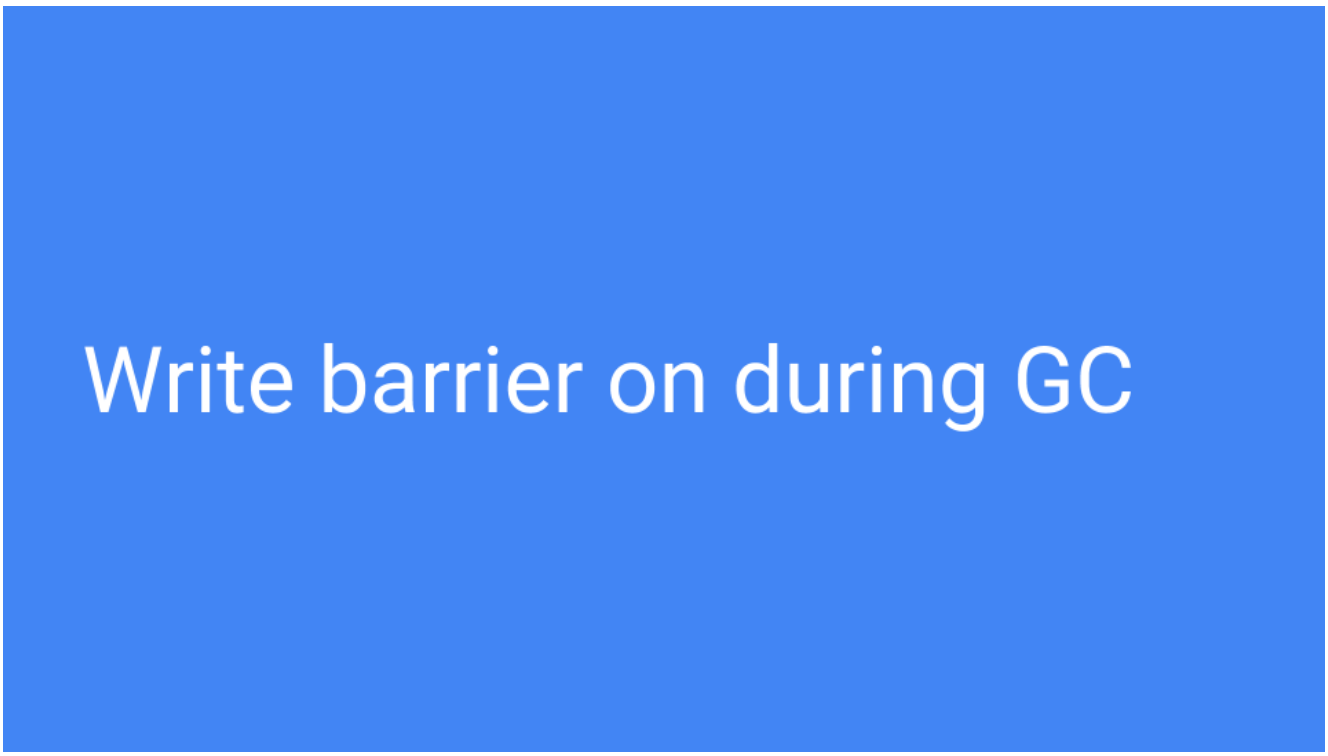
Internal structures: We fully understood and had experience with them. We understood how to do size segregated spans, we understood how to do low or zero contention allocation paths.

Speed: Non-copy did not concern us, allocation admittedly might be slower but still in the order of C. It might not be as fast as bump pointer but that was OK.

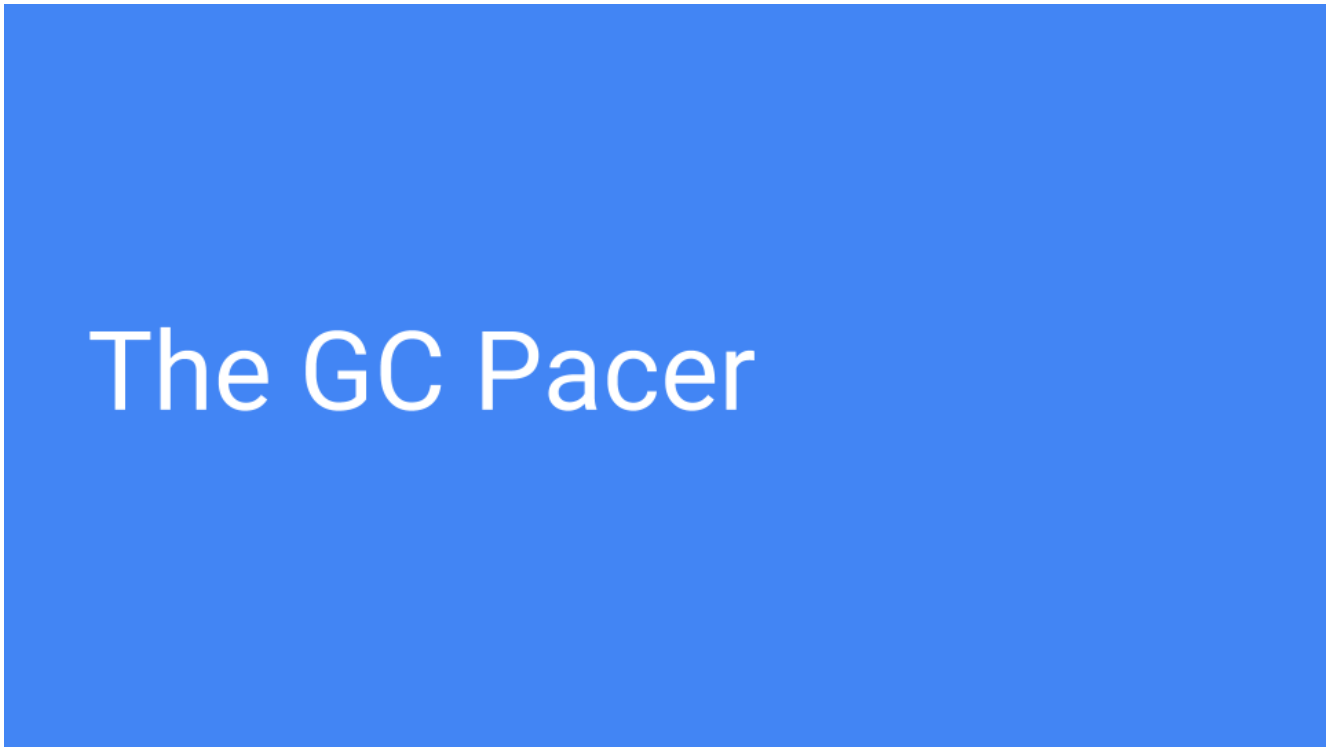
We also had this foreign function interface issue. If we didn't move our objects then we didn't have to deal with the long tail of bugs you might encounter if you had a moving collector as you attempt to pin objects and put levels of indirection between C and the Go object you are working with.



The next design choice was where to put the object's metadata. We needed to have some information about the objects since we didn't have headers. Mark bits are kept on the side and used for marking as well as allocation. Each word has 2 bits associated with it to tell you if it was a scalar or a pointer inside that word. It also encoded whether there were more pointers in the object so we could stop scanning objects sooner than later. We also had an extra bit encoding that we could use as an extra mark bit or to do other debugging things. This was really valuable for getting this stuff running and finding bugs.



So what about write barriers? The write barrier is on only during the GC. At other times the compiled code loads a global variable and looks at it. Since the GC was typically off the hardware correctly speculates to branch around the write barrier. When we are inside the GC that variable is different, and the write barrier is responsible for ensuring that no reachable objects get lost during the tri-color operations.



The other piece of this code is the GC Pacer. It is some of the great work that Austin did. It is basically based on a feedback loop that determines when to best start a GC cycle. If the system is in a steady state and not in a phase change, marking will end just about the time memory runs out.

That might not be the case so the Pacer also has to monitor the marking progress and ensure allocation doesn't overrun the concurrent marking.

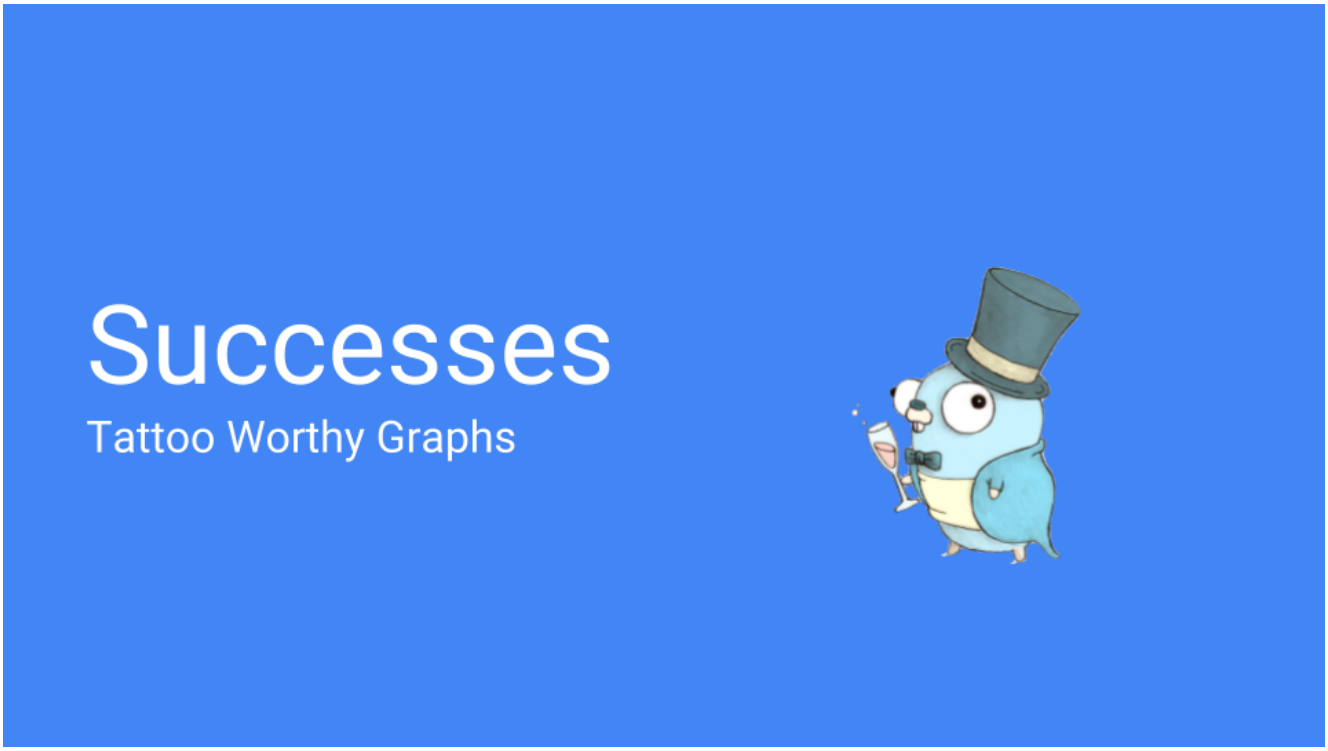
If need be, the Pacer slows down allocation while speeding up marking. At a high level the Pacer stops the Goroutine, which is doing a lot of the allocation, and puts it to work doing marking. The amount of work is proportional to the Goroutine's allocation. This speeds up the garbage collector while slowing down the mutator.

When all of this is done the Pacer takes what it has learnt from this GC cycle as well as previous ones and projects when to start the next GC.

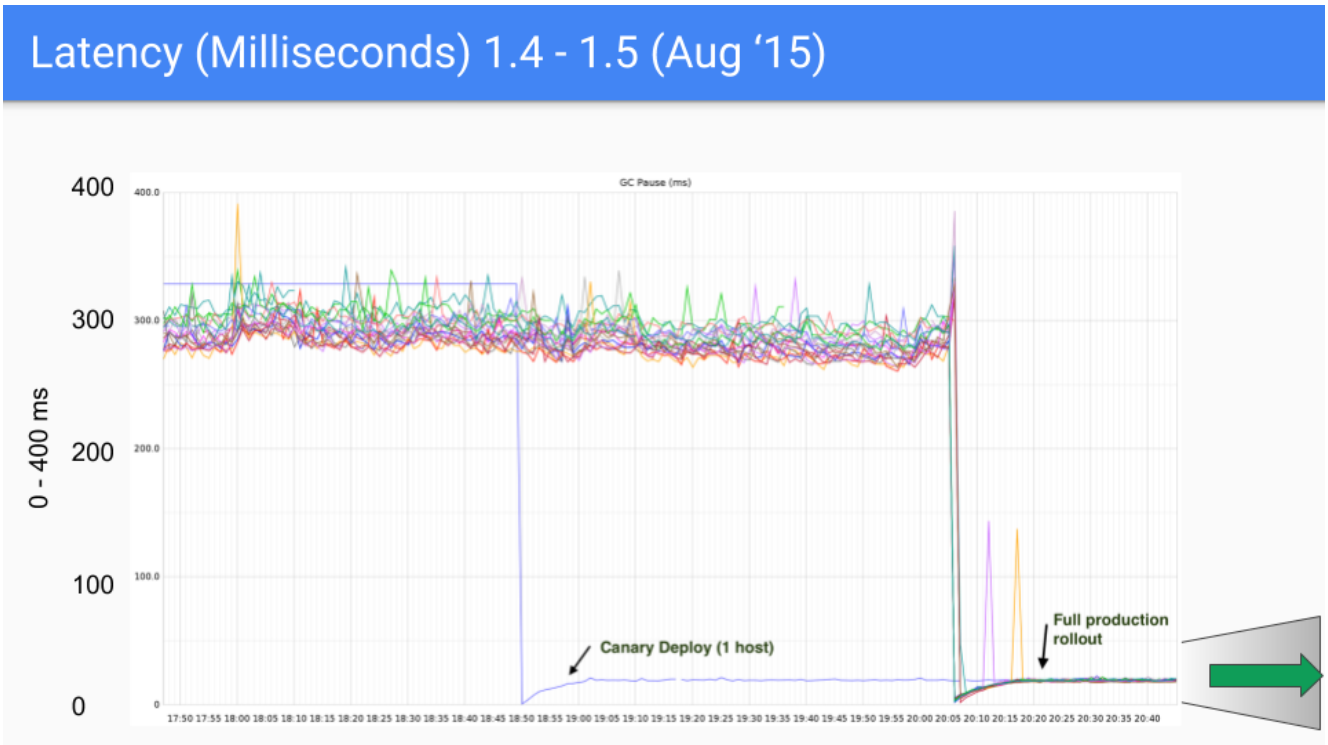
It does much more than this but that is the basic approach.

The math is absolutely fascinating, ping me for the design docs. If you are doing a concurrent GC you really owe it to yourself to look at this math and see if it's the same as your math. If you have any suggestions let us know.

*Go 1.5 concurrent garbage collector pacing and Proposal: Separate soft and hard heap size goal



Yes, so we had successes, lots of them. A younger crazier Rick would have taken some of these graphs and tattooed them on my shoulder I was so proud of them.

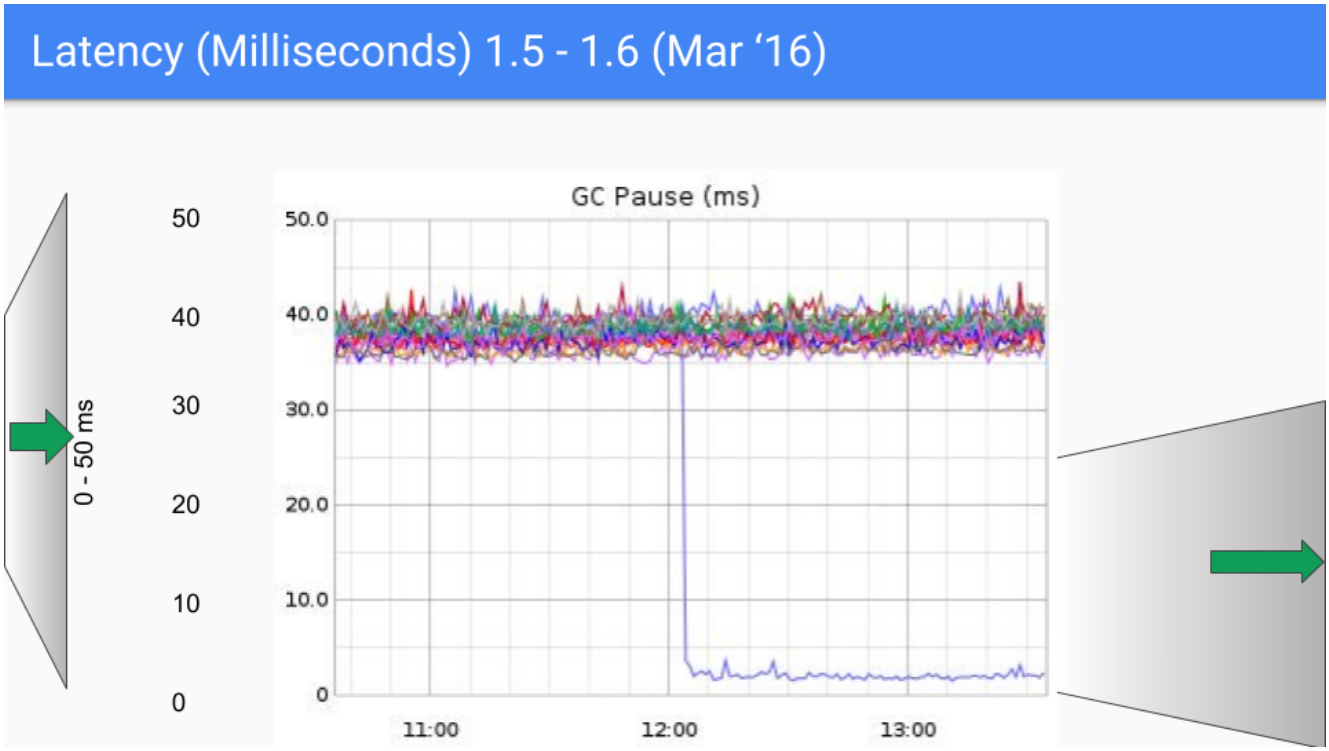


This is a series of graphs that was done for a production server at Twitter. We of course had nothing to do with that production server. Brian Hatfield did these measurements and oddly enough tweeted about them.

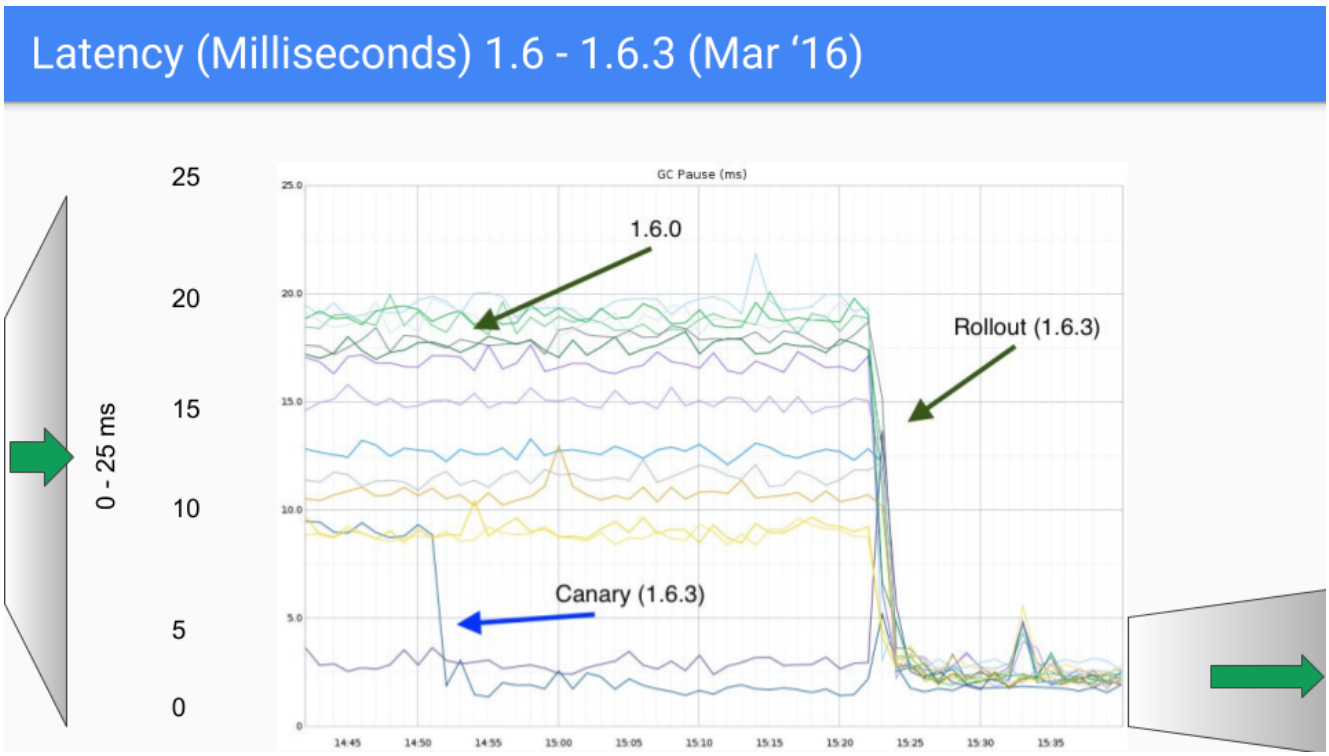
On the Y axis we have GC latency in milliseconds. On the X axis we have time. Each of the points is a stop the world pause time during that GC.

On our first release, which was in August of 2015, we saw a drop from around 300 - 400 milliseconds down to 30 or 40 milliseconds. This was good, order of magnitude good.

We are going to change the Y-axis here radically from 0 to 400 milliseconds down to 0 to 50 milliseconds.

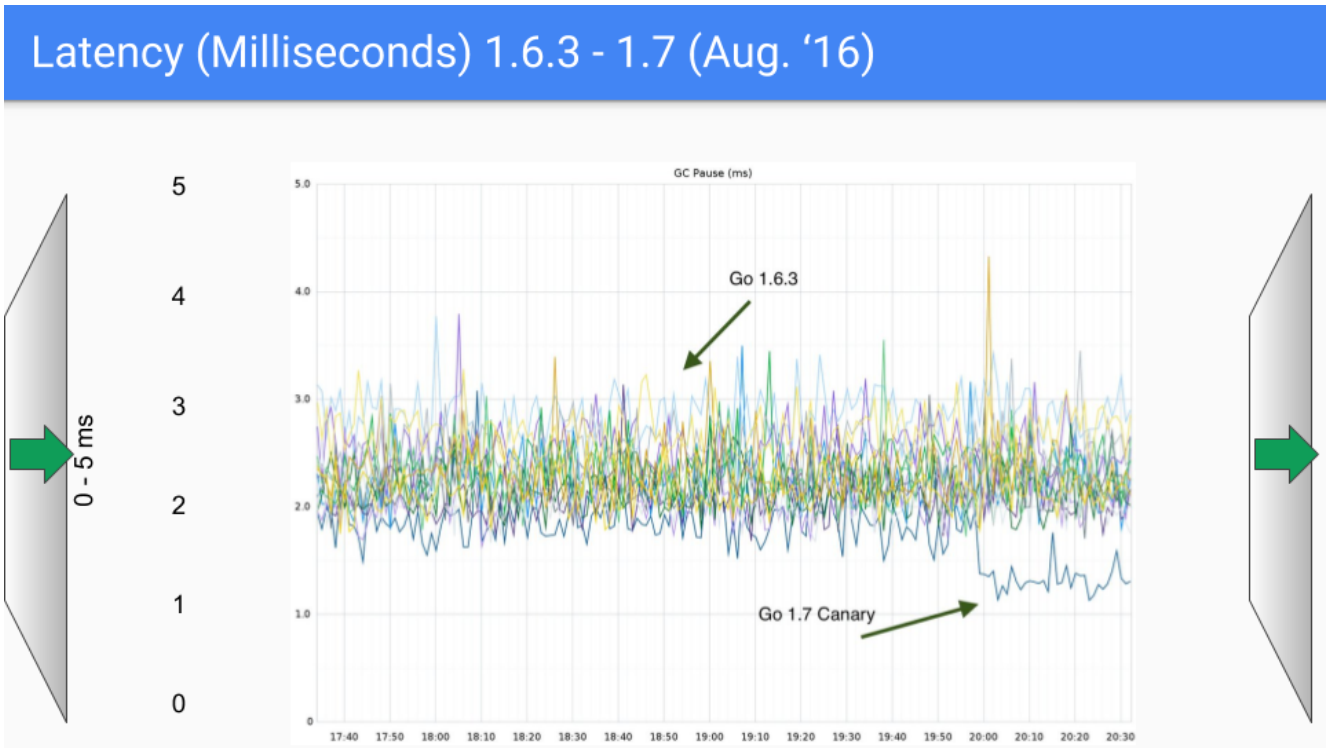


This is 6 months later. The improvement was largely due to systematically eliminating all the O(heap) things we were doing during the stop the world time. This was our second order of magnitude improvement as we went from 40 milliseconds down to 4 or 5.

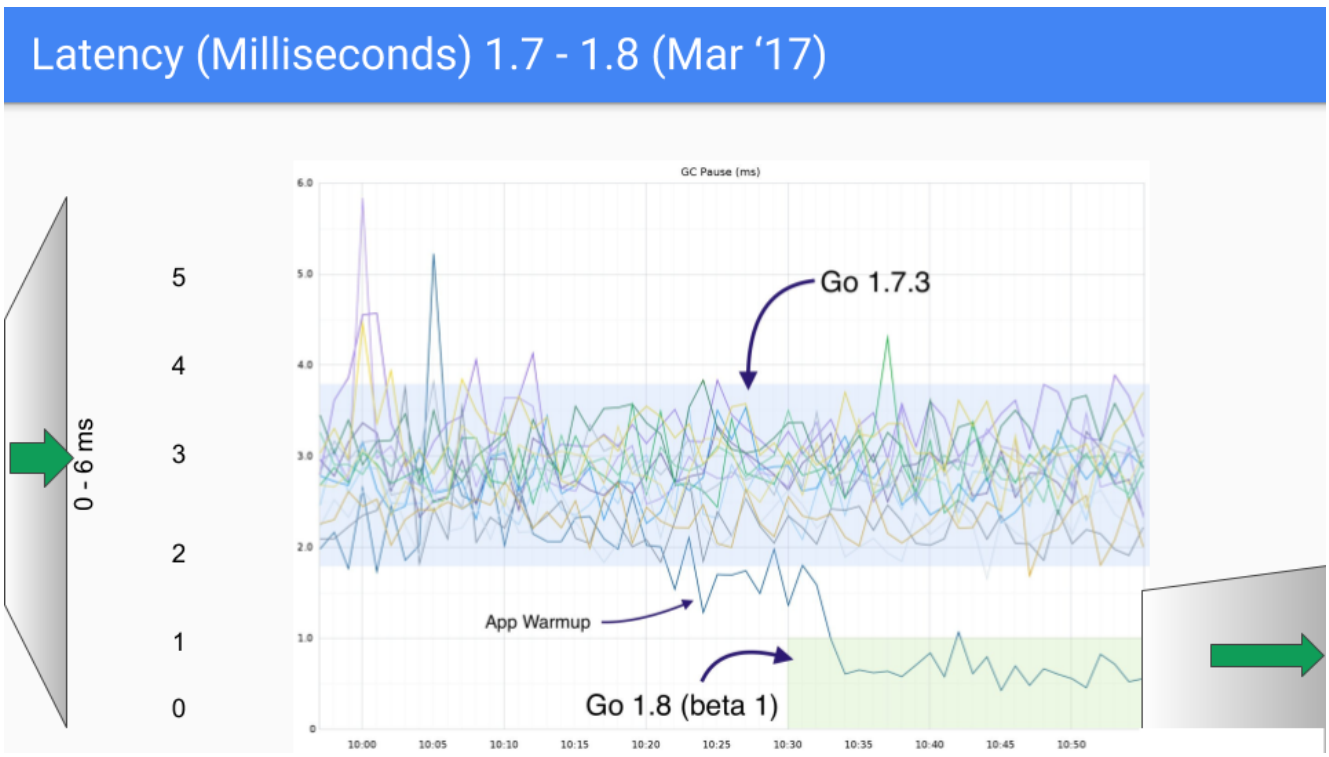


There were some bugs in there that we had to clean up and we did this during a minor release 1.6.3. This dropped latency down to well under 10 milliseconds, which was our SLO.

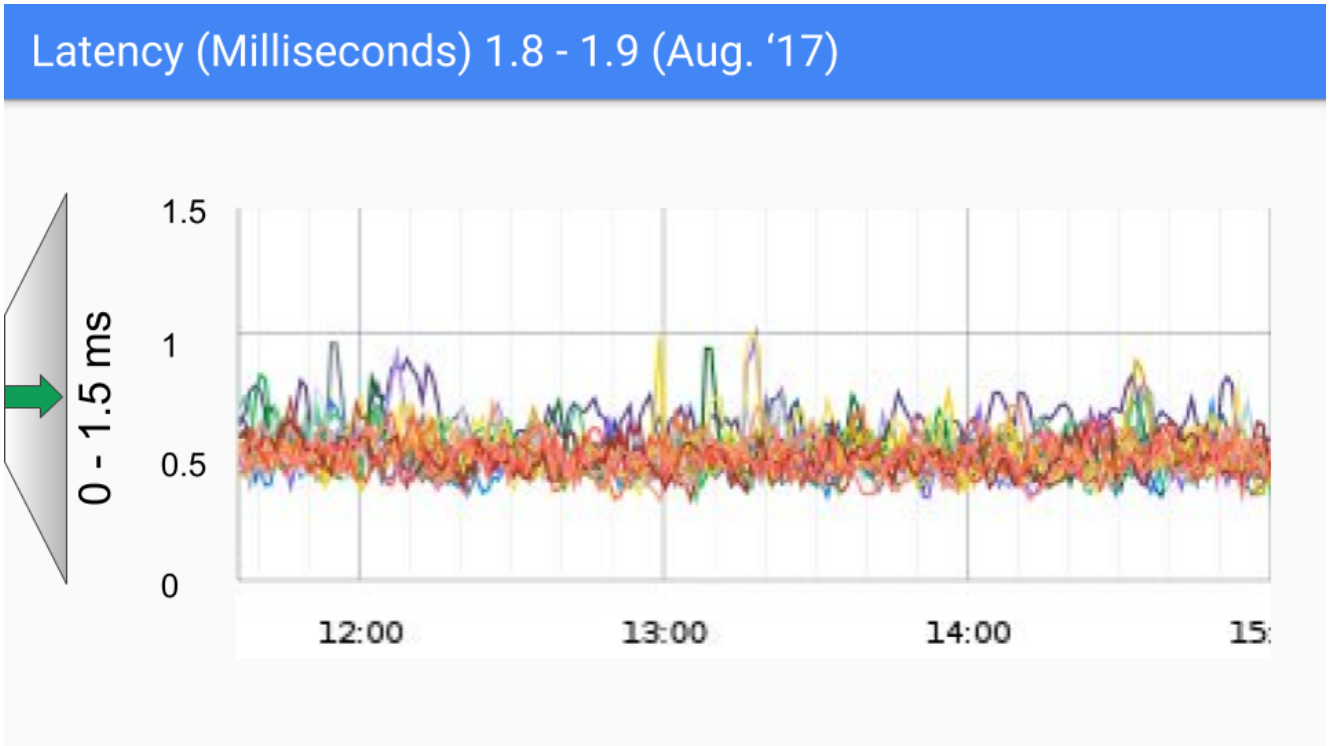
We are about to change our Y-axis again, this time down to 0 to 5 milliseconds.



So here we are, this is August of 2016, a year after the first release. Again we kept knocking off these O(heap size) stop the world processes. We are talking about an 18Gbyte heap here. We had much larger heaps and as we knocked off these O(heap size) stop the world pauses, the size of the heap could obviously grow considerable without impacting latency. So this was a bit of a help in 1.7.

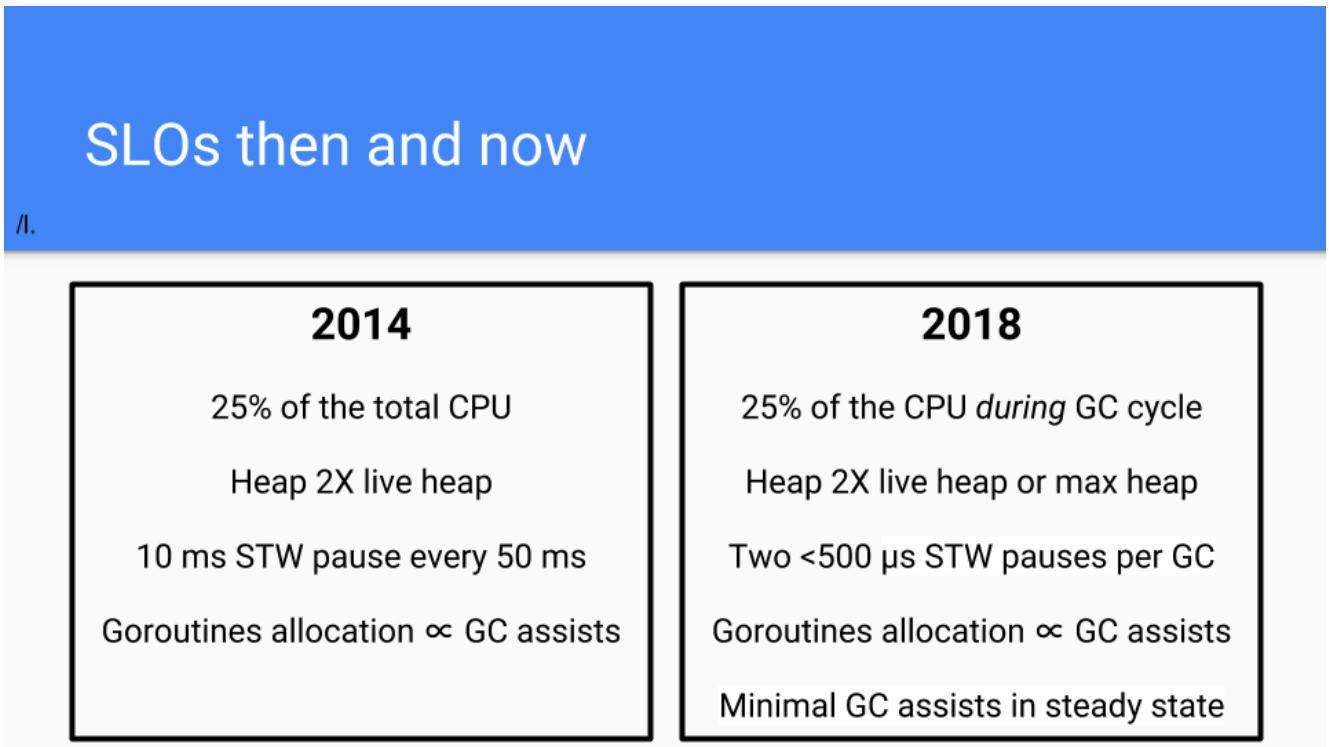


The next release was in March of 2017. We had the last of our large latency drops which was due to figuring out how to avoid the stop the world stack scanning at the end of the GC cycle. That dropped us into the sub-millisecond range. Again the Y axis is about to change to 1.5 milliseconds and we see our third order of magnitude improvement.



The August 2017 release saw little improvement. We know what is causing the remaining pauses. The SLO whisper number here is around 100-200 microseconds and we will push towards that. If you see anything over a couple hundred microseconds then we really want to talk to you and figure out whether it fits into the stuff we know about or whether it is something new we haven't looked into. In any case there seems to be little call for lower latency. It is important to note these latency levels can happen for a wide variety of non-GC reasons and as the saying goes "You don't have to be faster than the bear, you just have to be faster than the guy next to you."

There was no substantial change in the Feb'18 1.10 release just some clean-up and chasing corner cases.



So a new year and a new SLO This is our 2018 SLO.

We have dropped total CPU to CPU used during a GC cycle.

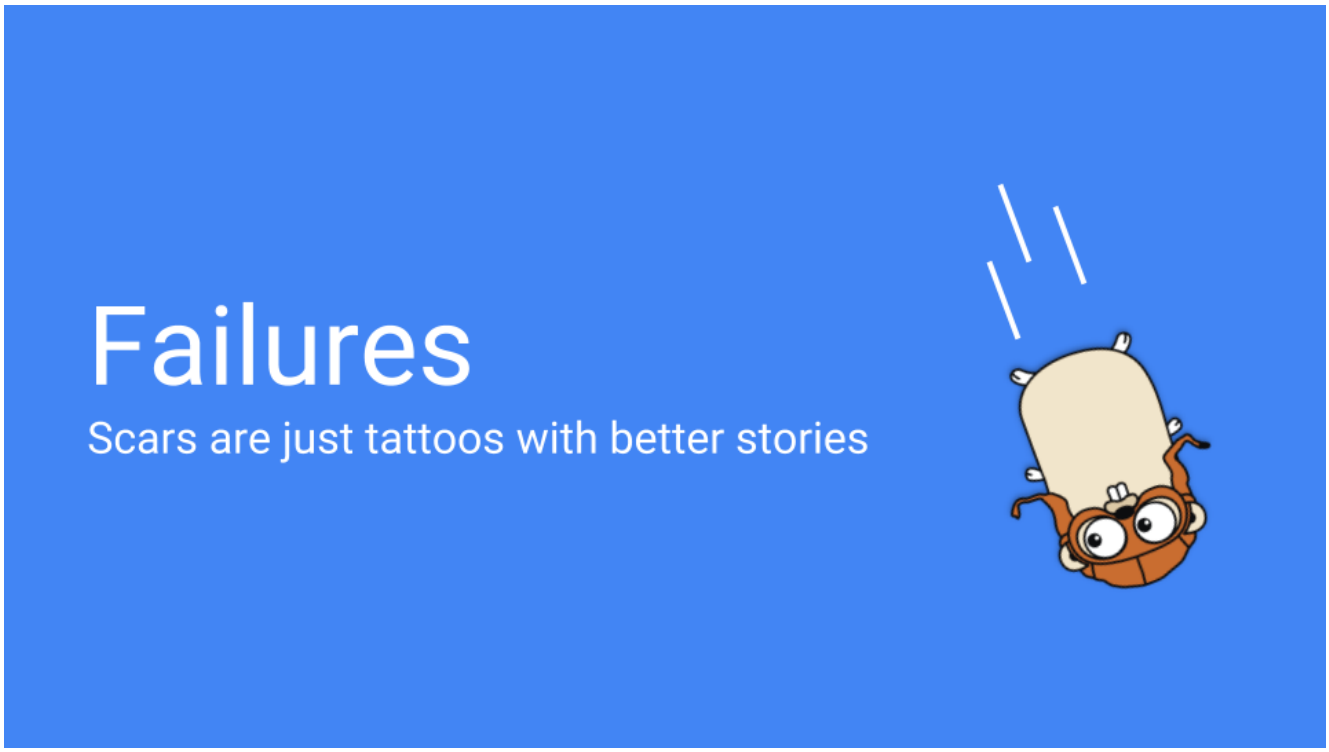
The heap is still at 2x.

We now have an objective of 500 microseconds stop the world pause per GC cycle. Perhaps a little sandbagging here.

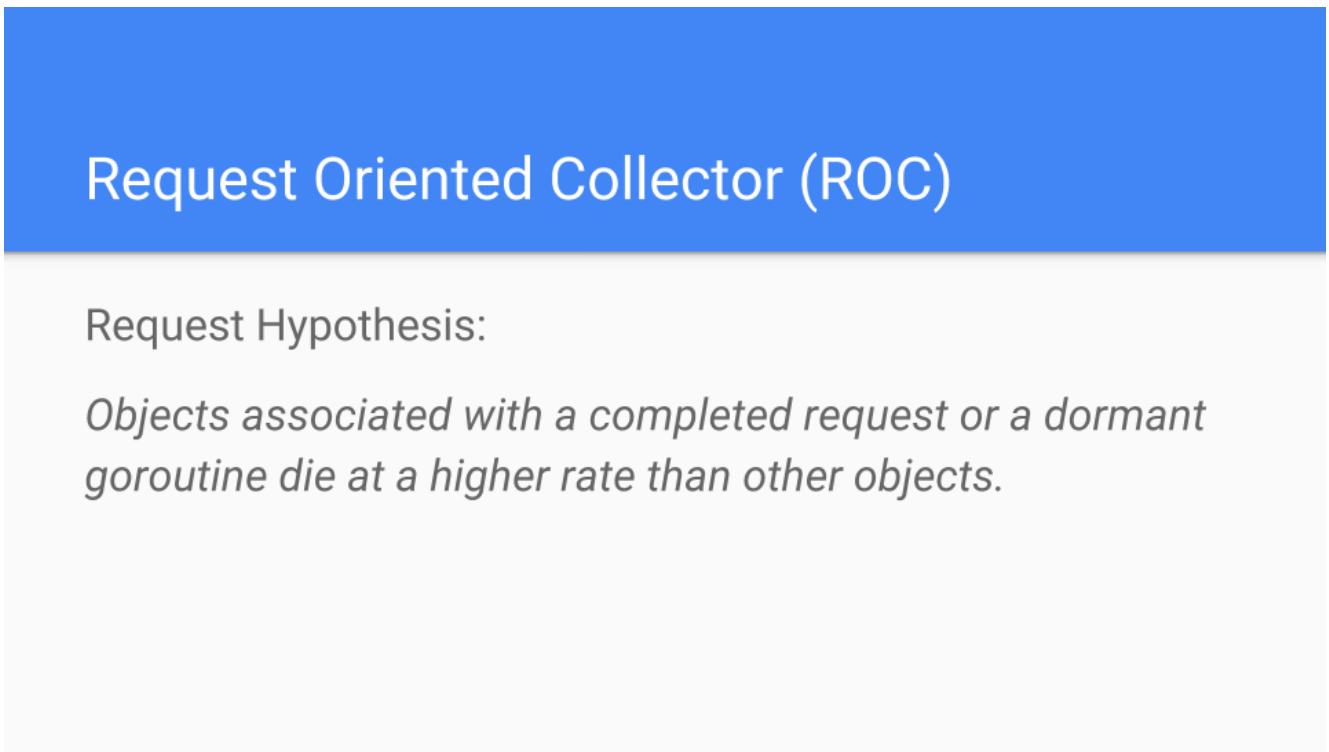
The allocation would continue to be proportional to the GC assists.

The Pacer had gotten much better so we looked to see minimal GC assists in a steady state.

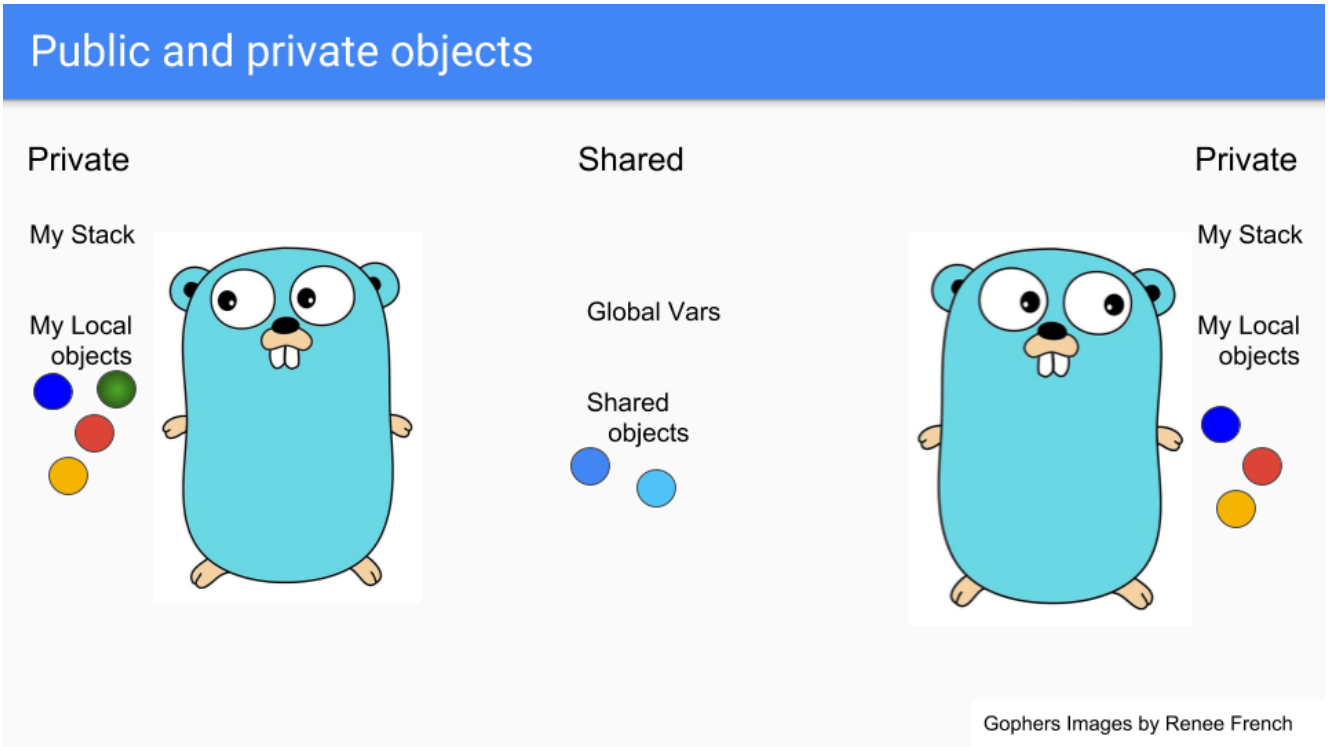
We were pretty happy with this. Again this is not an SLA but an SLO so it's an objective, not an agreement, since we can't control such things as the OS.



That's the good stuff. Let's shift and start talking about our failures. These are our scars; they are sort of like tattoos and everyone gets them. Anyway they come with better stories so let's do some of those stories.

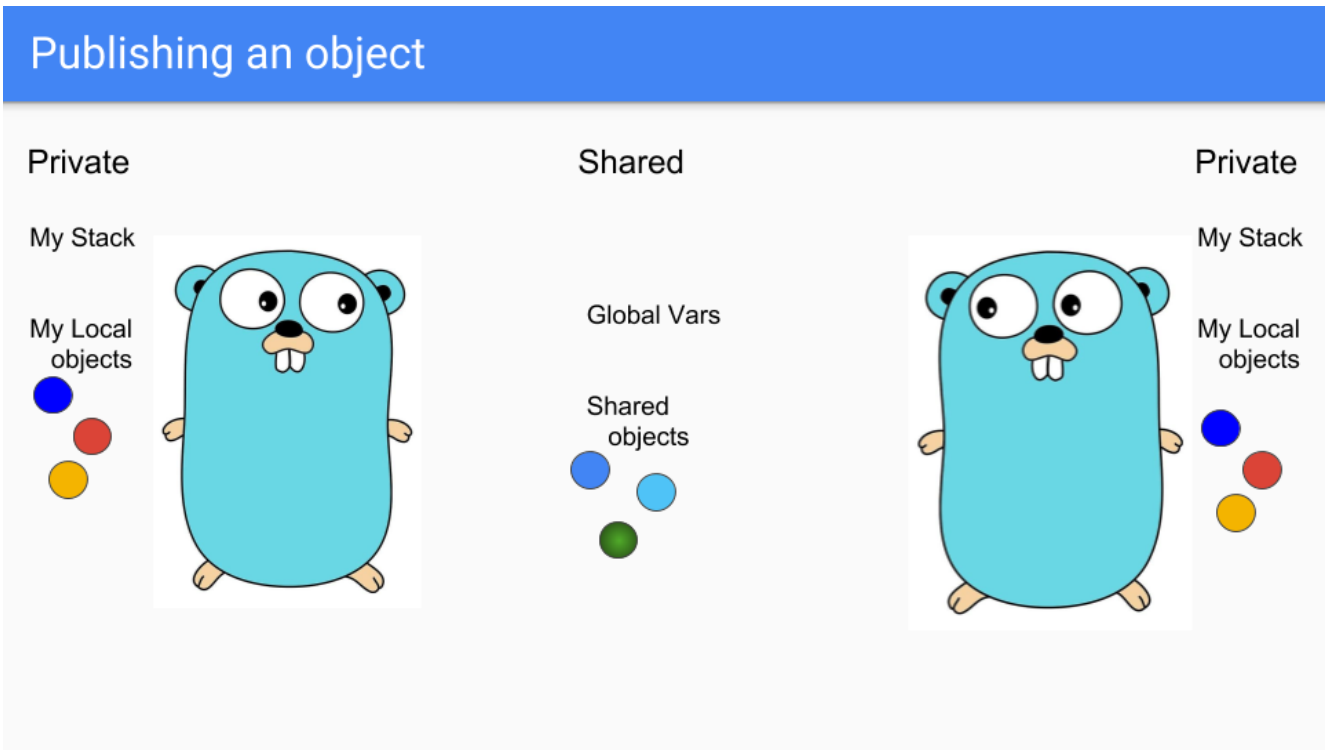


Our first attempt was to do something called the request oriented collector or ROC. The hypothesis can be seen here.

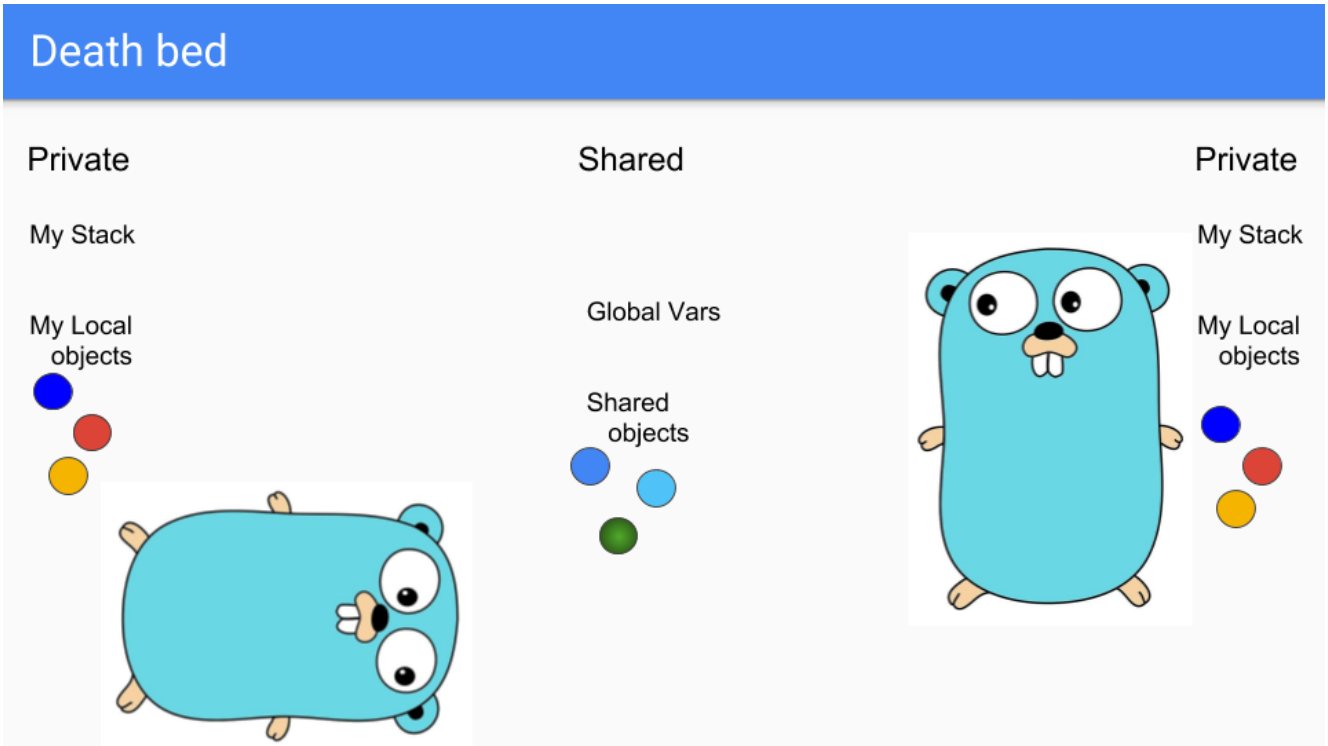


So what does this mean?

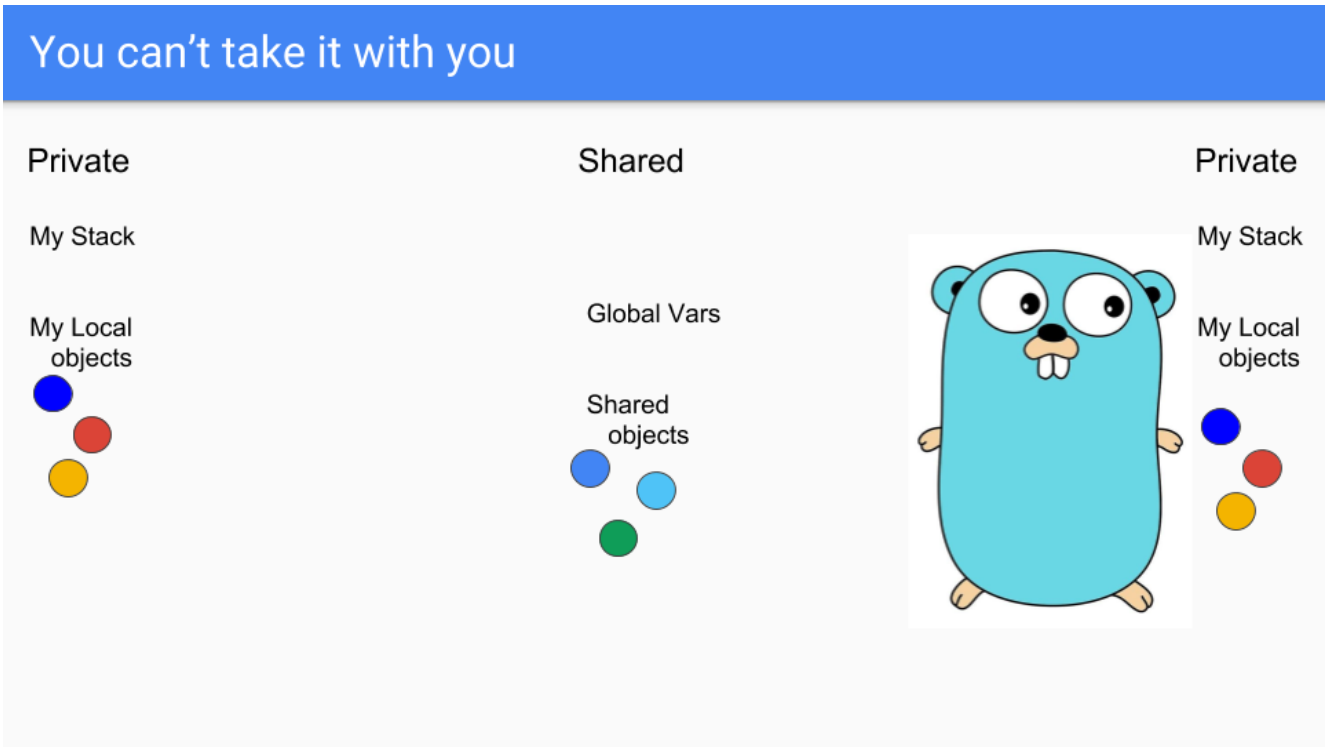
Goroutines are lightweight threads that look like Gophers, so here we have two Goroutines. They share some stuff such as the two blue objects there in the middle. They have their own private stacks and their own selection of private objects. Say the guy on the left wants to share the green object.



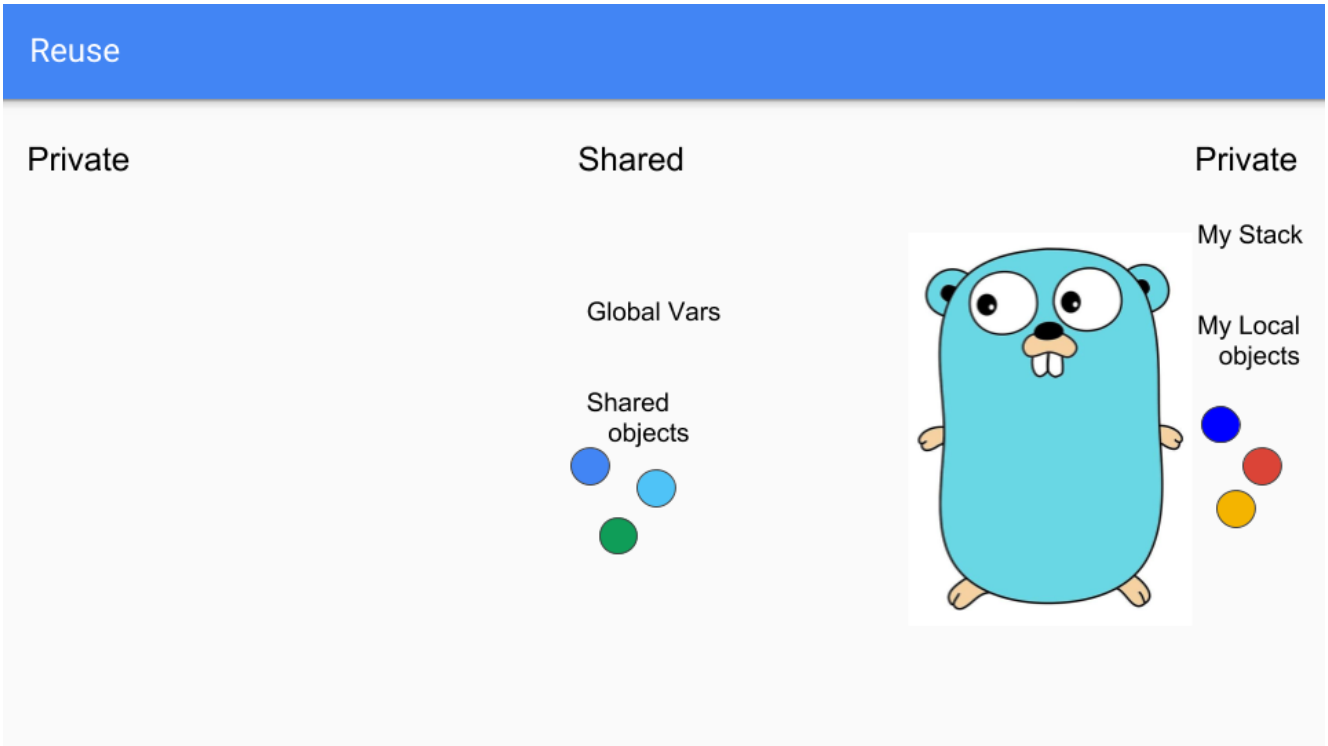
The goroutine puts it in the shared area so the other Goroutine can access it. They can hook it to something in the shared heap or assign it to a global variable and the other Goroutine can see it.



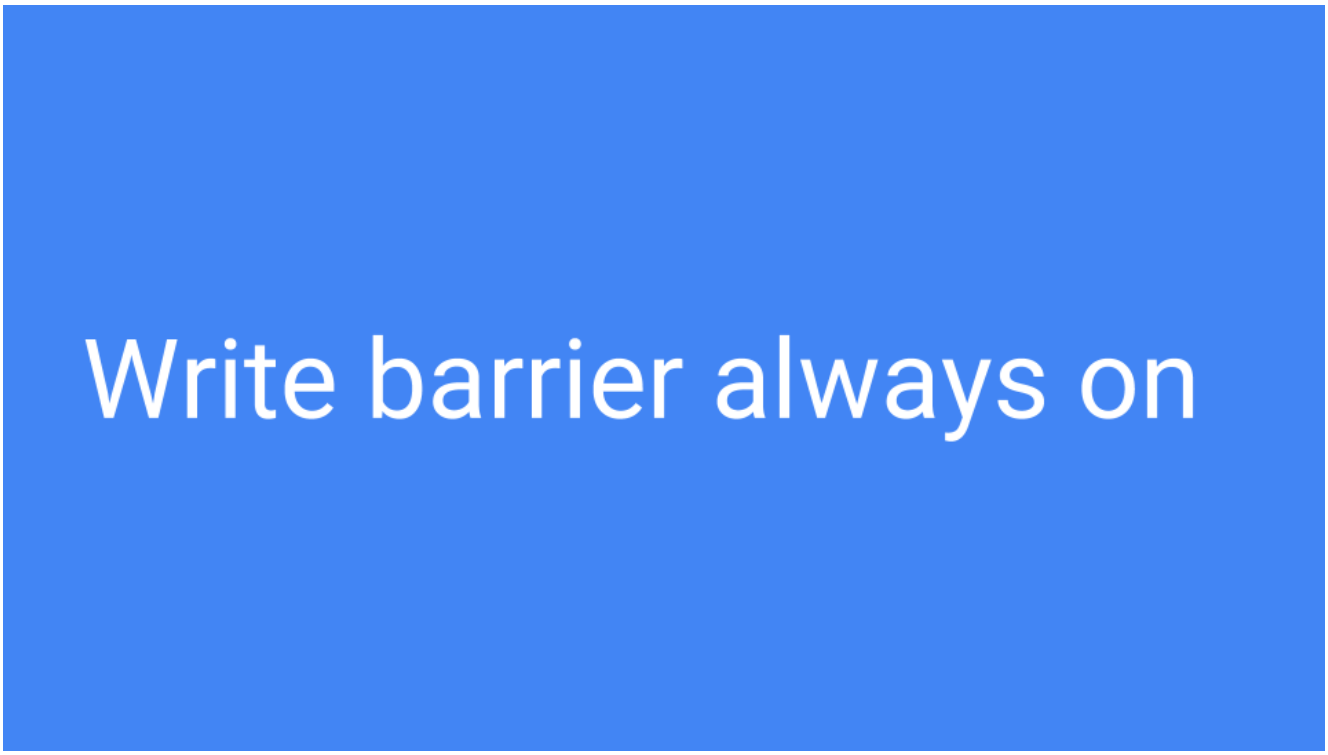
Finally the Goroutine on the left goes to its death bed, it's about to die, sad.



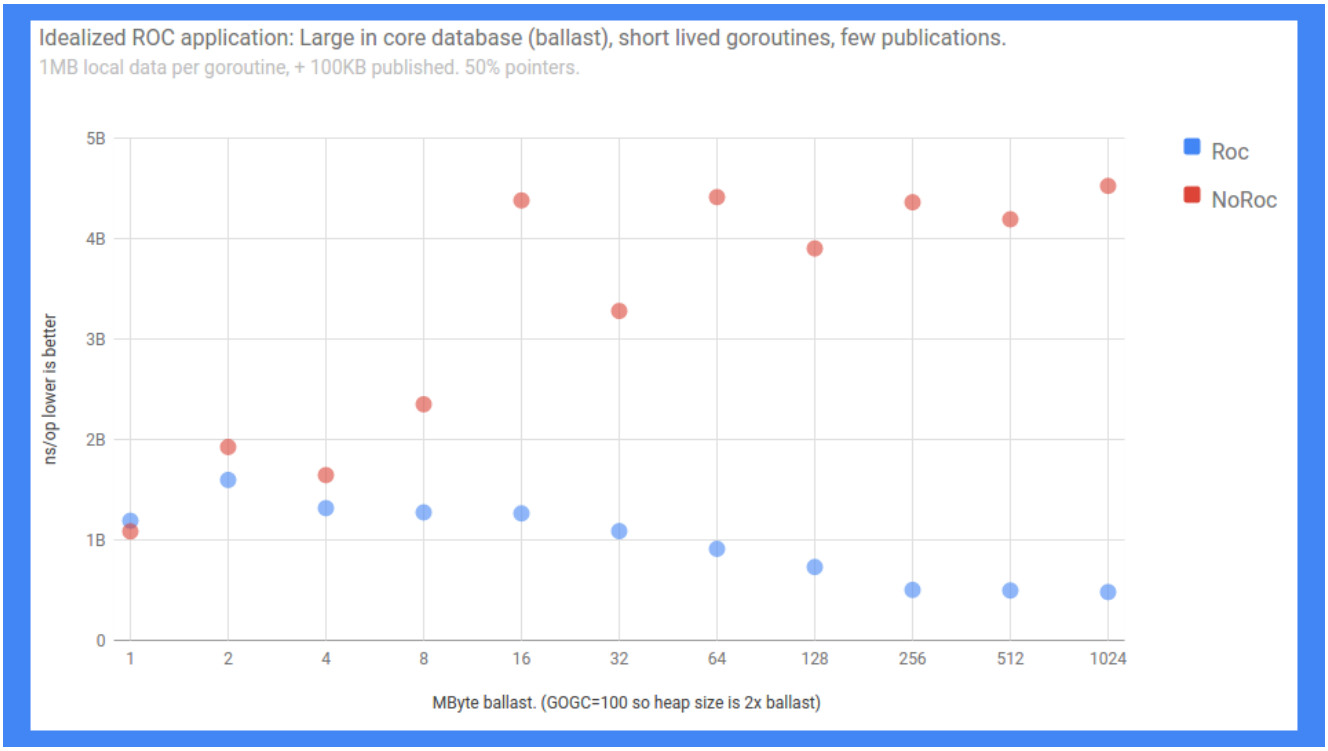
As you know you can't take your objects with you when you die. You can't take your stack either. The stack is actually empty at this time and the objects are unreachable so you can simply reclaim them.



The important thing here is that all actions were local and did not require any global synchronization. This is fundamentally different than approaches like a generational GC, and the hope was that the scaling we would get from not having to do that synchronization would be sufficient for us to have a win.



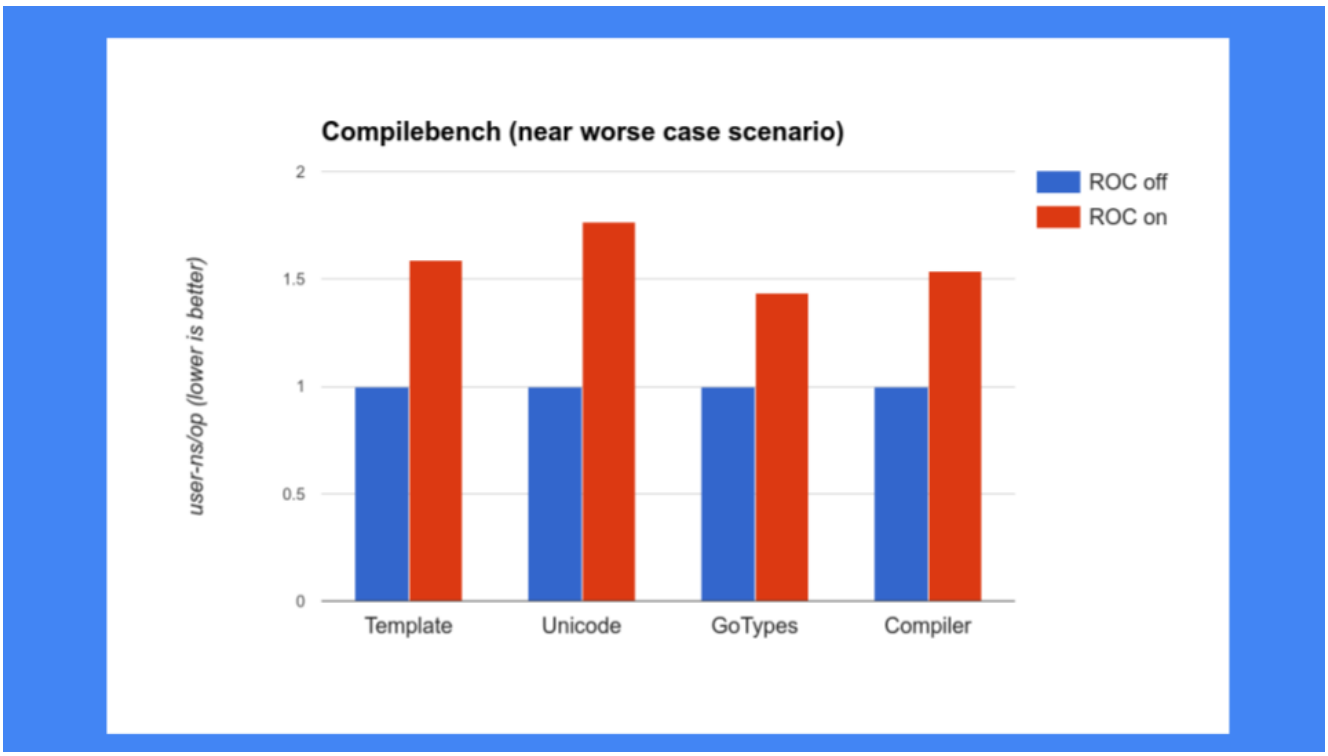
The other issue that was going on with this system was that the write barrier was always on. Whenever there was a write, we would have to see if it was writing a pointer to a private object into a public object. If so, we would have to make the referent object public and then do a transitive walk of reachable objects making sure they were also public. That was a pretty expensive write barrier that could cause many cache misses.



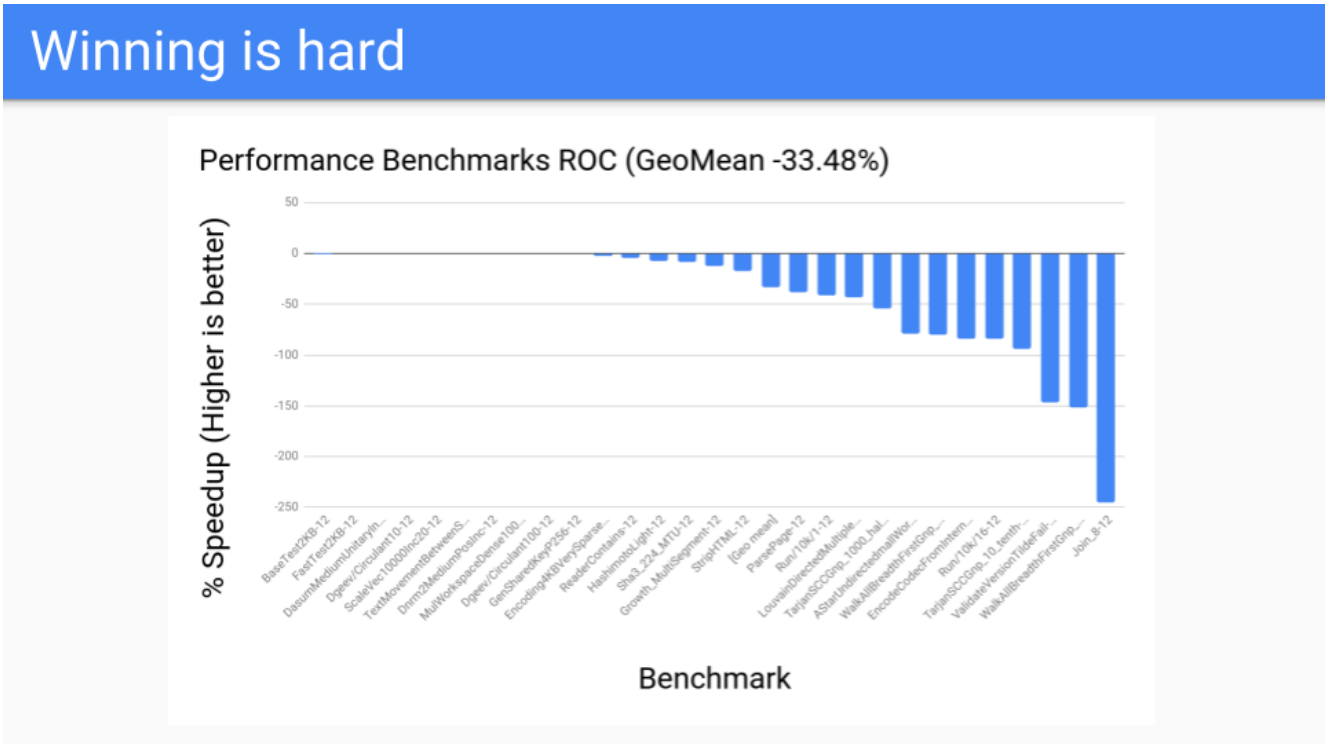
That said, wow, we had some pretty good successes.

This is an end-to-end RPC benchmark. The mislabeled Y axis goes from 0 to 5 milliseconds (lower is better), anyway that is just what it is. The X axis is basically the ballast or how big the in-core database is.

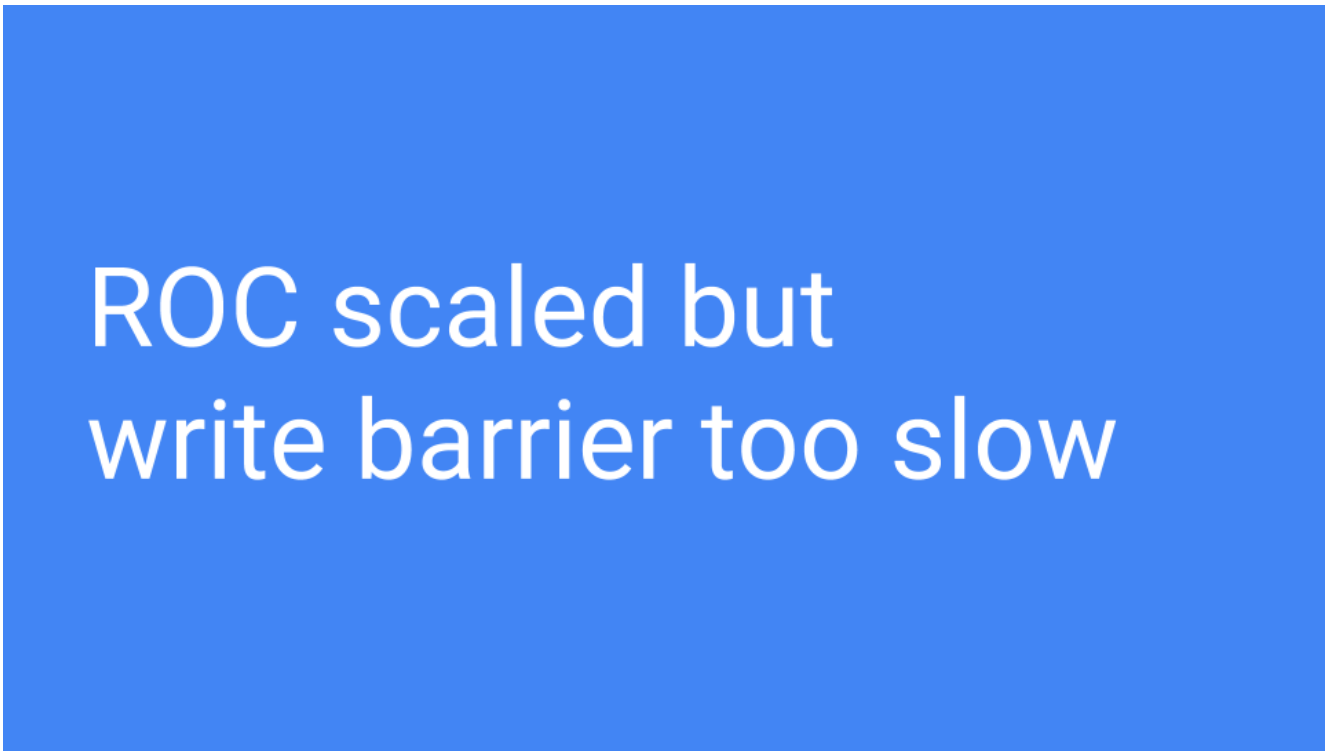
As you can see if you have ROC on and not a lot of sharing, things actually scale quite nicely. If you don't have ROC on it wasn't nearly as good.



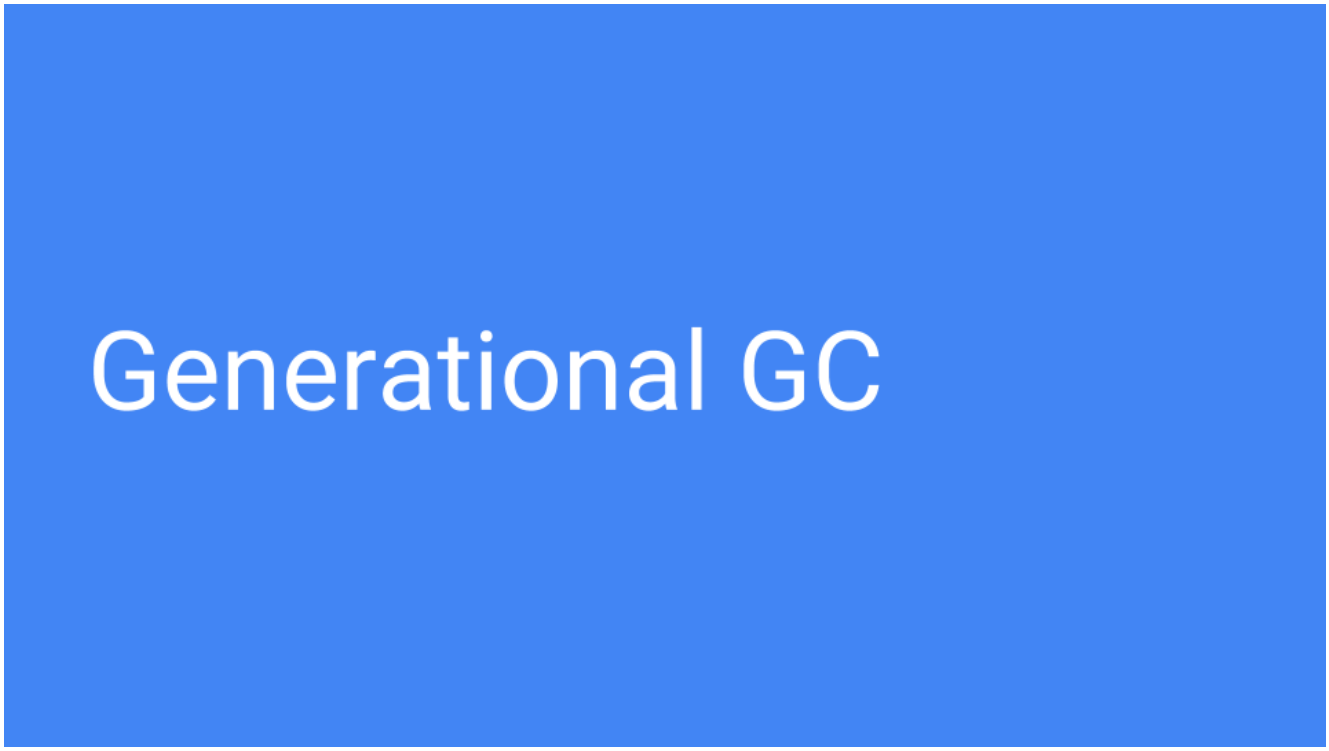
But that wasn't good enough, we also had to make sure that ROC didn't slow down other pieces of the system. At that point there was a lot of concern about our compiler and we could not slow down our compilers. Unfortunately the compilers were exactly the programs that ROC did not do well at. We were seeing 30, 40, 50% and more slowdowns and that was unacceptable. Go is proud of how fast its compiler is so we couldn't slow the compiler down, certainly not this much.



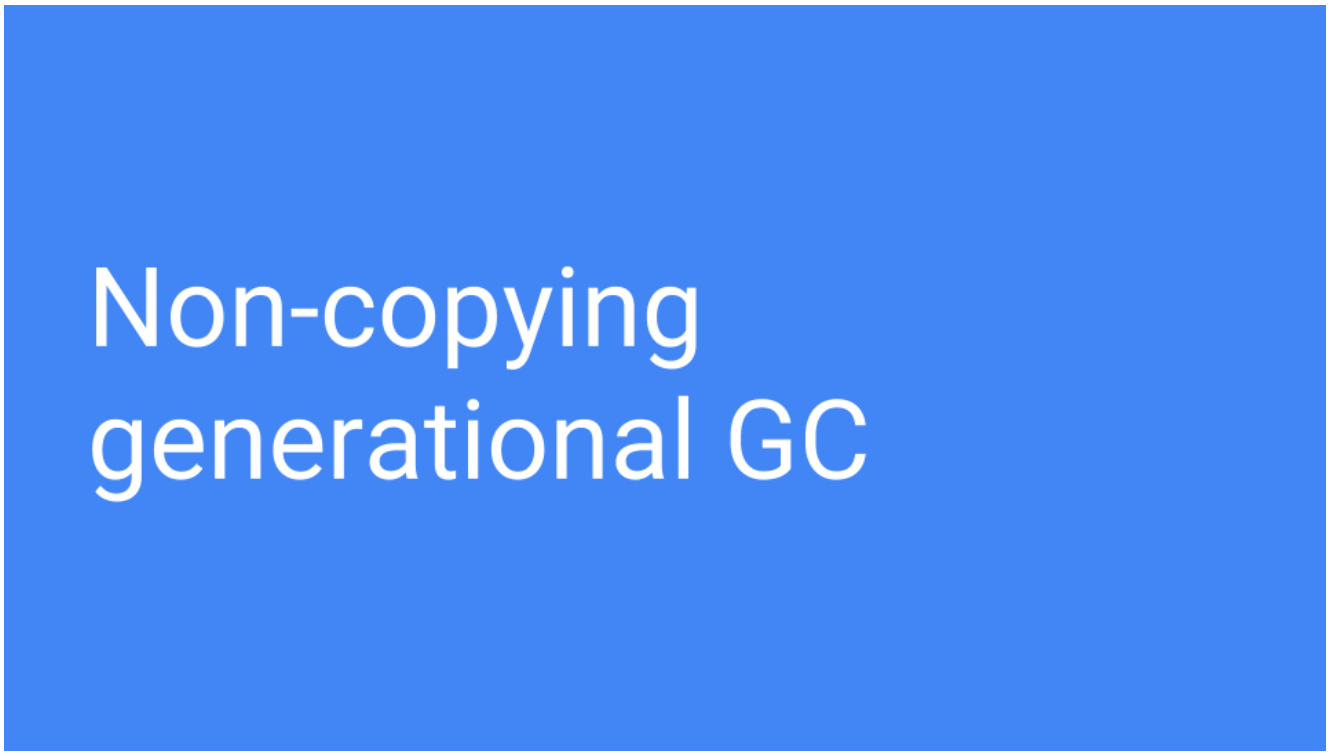
We then went and looked at some other programs. These are our performance benchmarks. We have a corpus of 200 or 300 benchmarks and these were the ones the compiler folks had decided were important for them to work on and improve. These weren't selected by the GC folks at all. The numbers were uniformly bad and ROC wasn't going to become a winner.



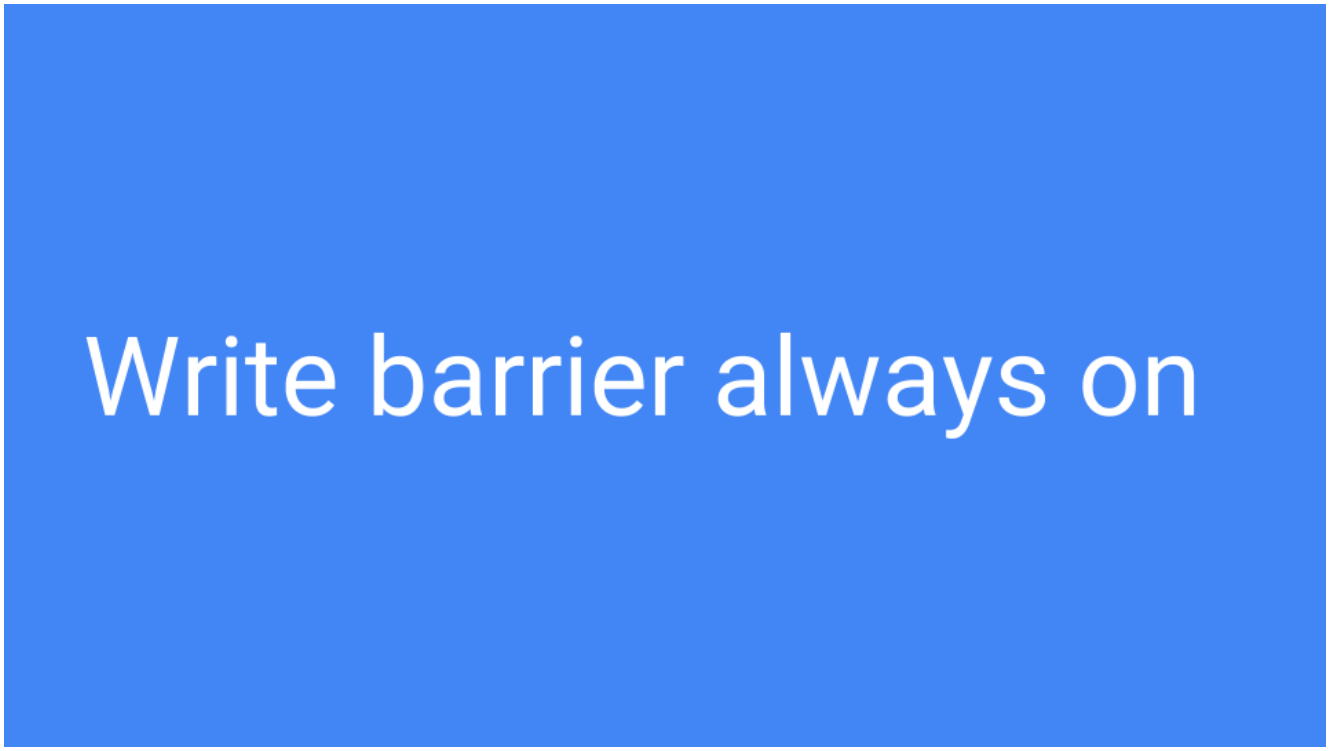
It's true we scaled but we only had 4 to 12 hardware thread system so we couldn't overcome the write barrier tax. Perhaps in the future when we have 128 core systems and Go is taking advantage of them, the scaling properties of ROC might be a win. When that happens we might come back and revisit this, but for now ROC was a losing proposition.



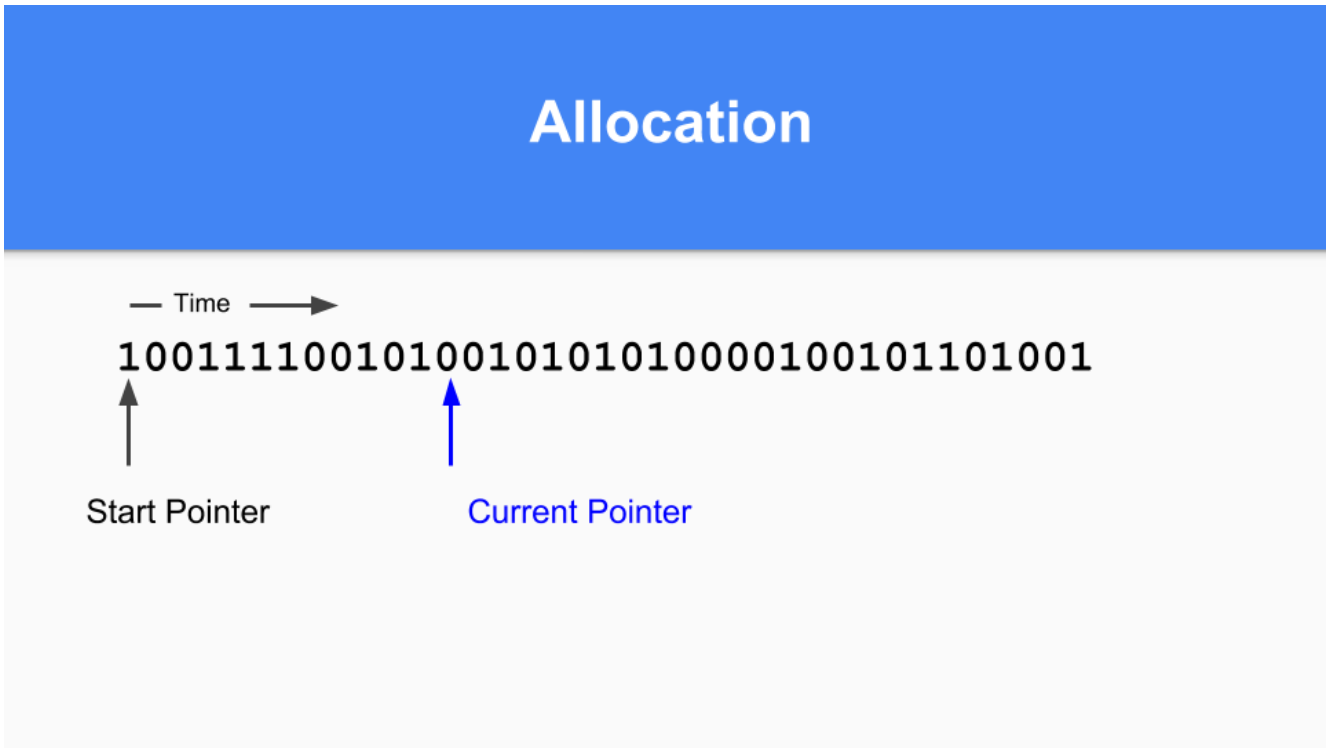
So what were we going to do next? Let's try the generational GC. It's an oldie but a goodie. ROC didn't work so let's go back to stuff we have a lot more experience with.



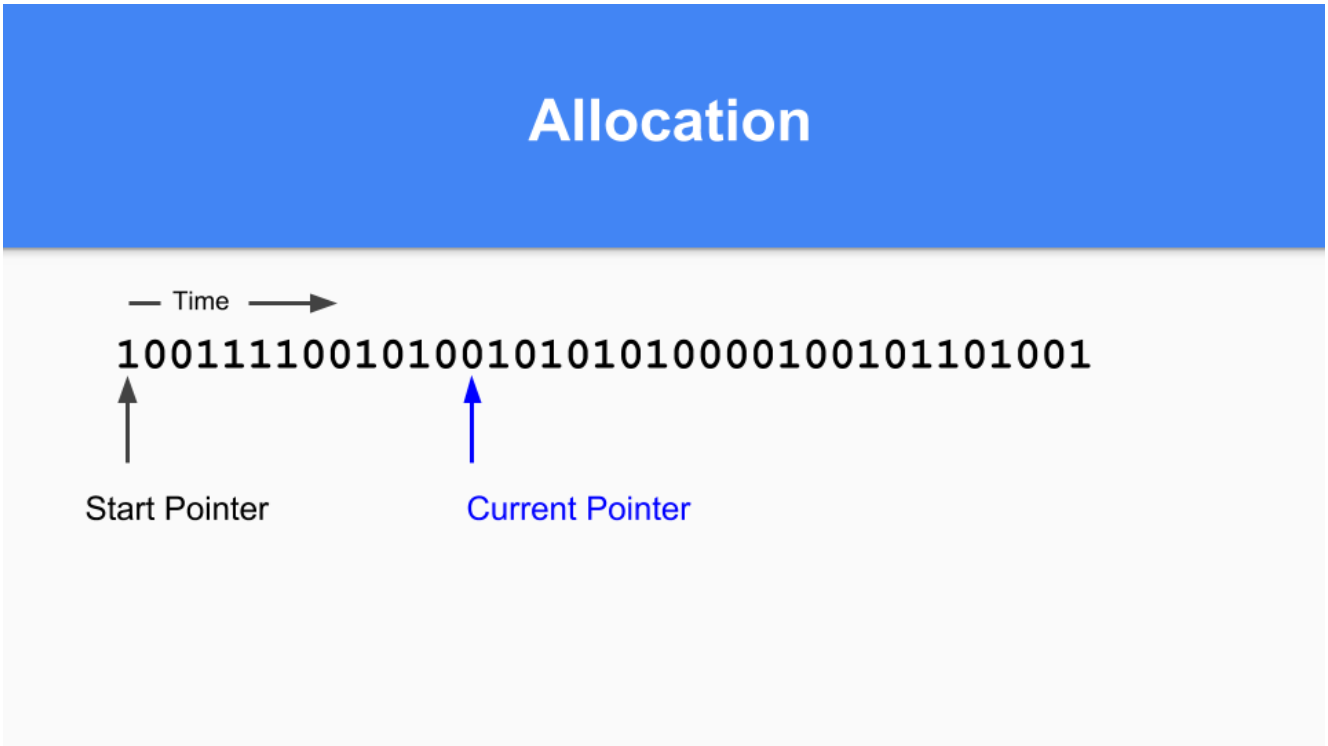
We weren't going to give up our latency, we weren't going to give up the fact that we were non-moving. So we needed a non-moving generational GC.



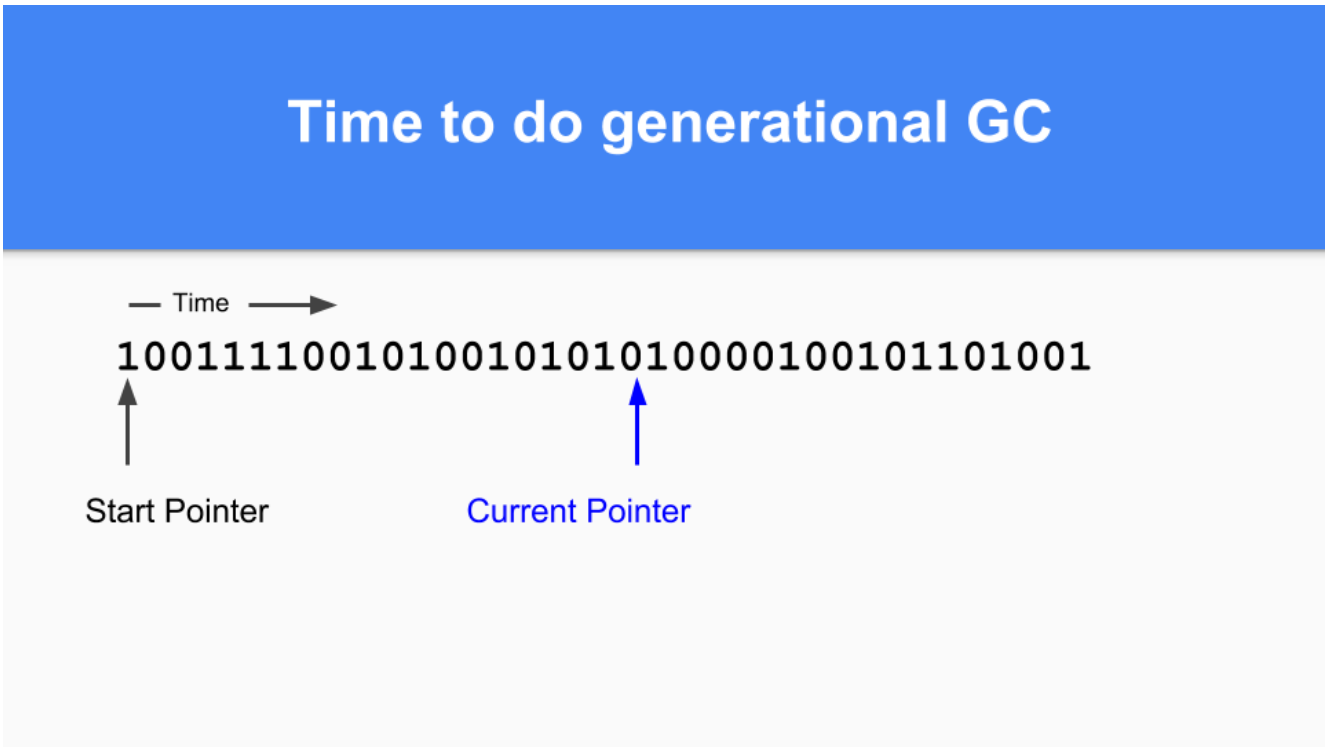
So could we do this? Yes, but with a generational GC, the write barrier is always on. When the GC cycle is running we use the same write barrier we use today, but when GC is off we use a fast GC write barrier that buffers the pointers and then flushes the buffer to a card mark table when it overflows.



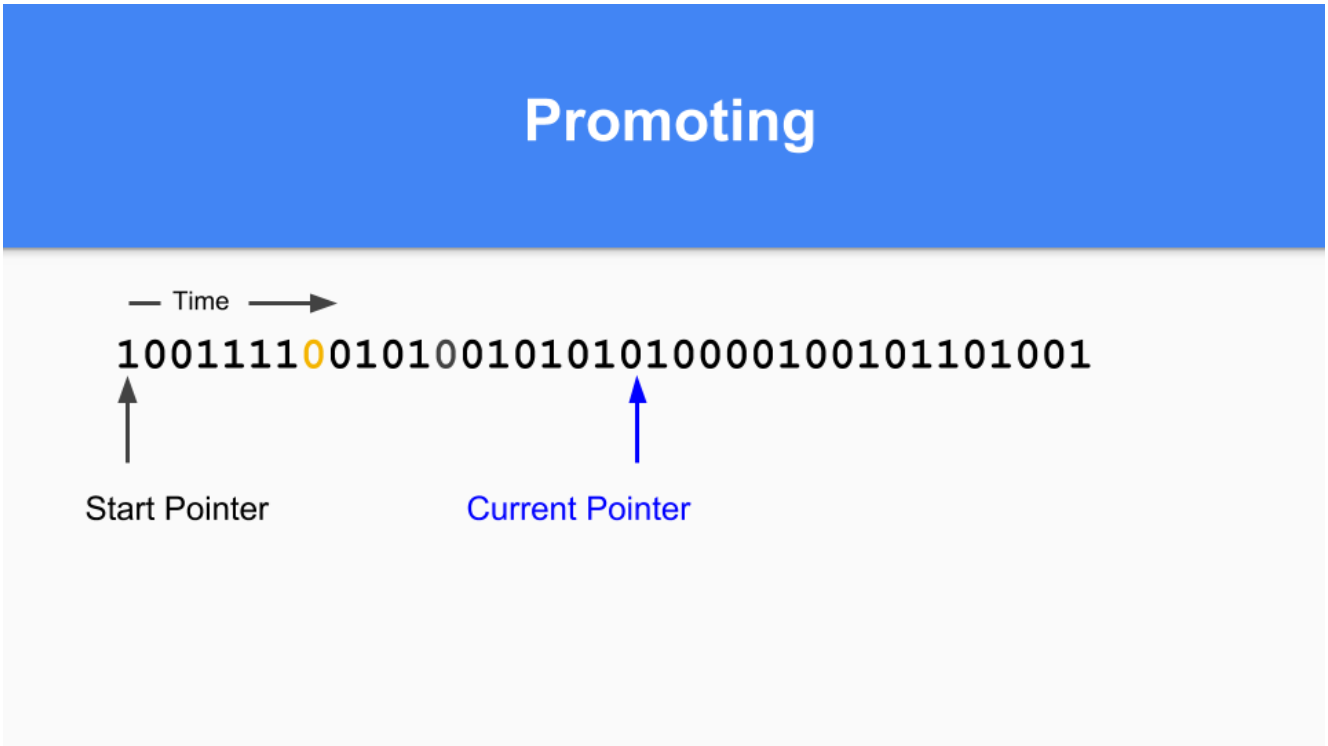
So how is this going to work in a non-moving situation? Here is the mark / allocation map. Basically you maintain a current pointer. When you are allocating you look for the next zero and when you find that zero you allocate an object in that space.



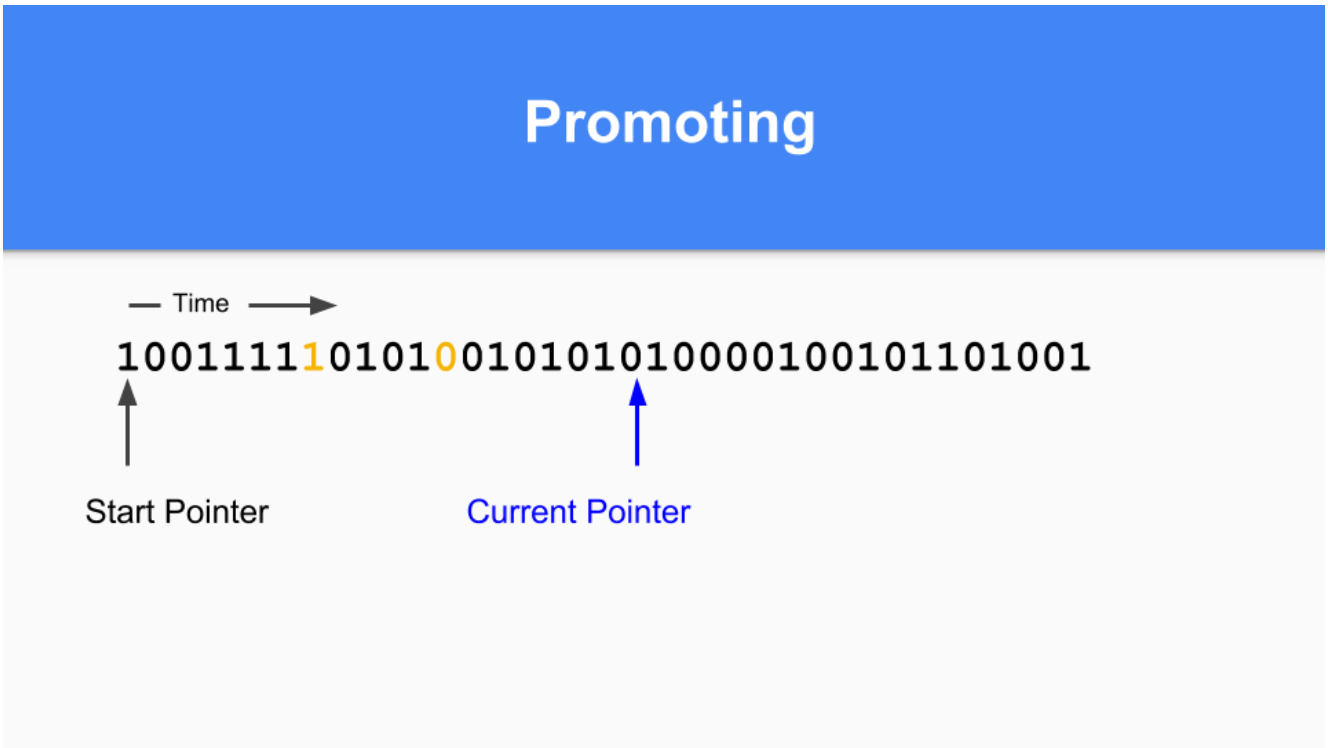
You then update the current pointer to the next 0.



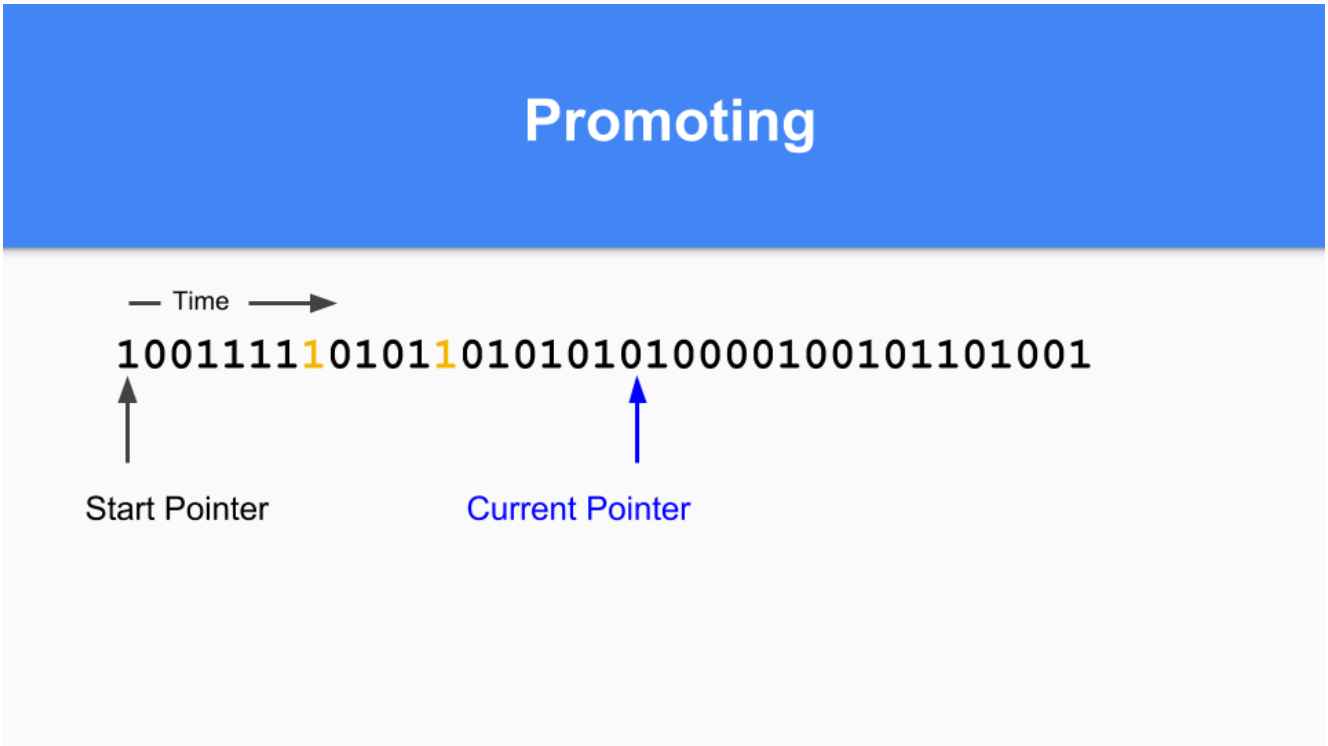
You continue until at some point it is time to do a generation GC. You will notice that if there is a one in the mark/allocation vector then that object was alive at the last GC so it is mature. If it is zero and you reach it then you know it is young.



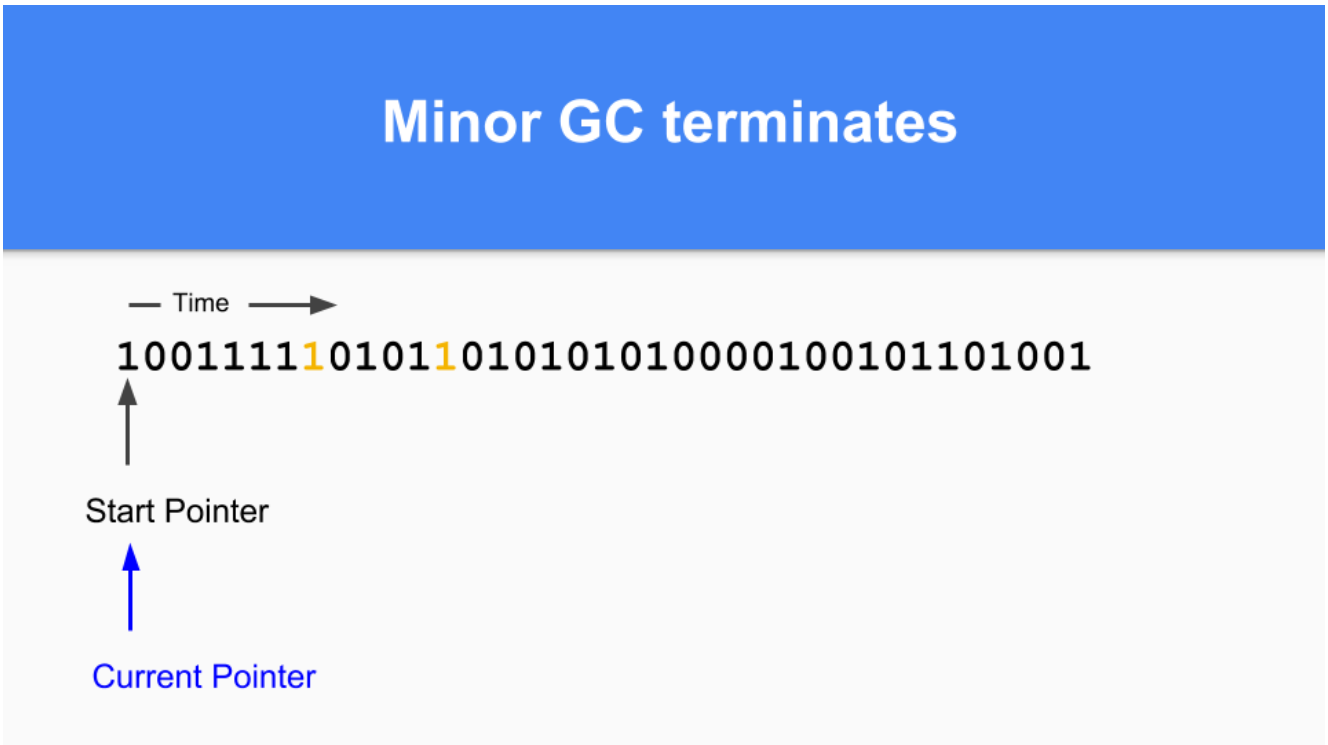
So how do you do promoting. If you find something marked with a 1 pointing to something marked with a 0 then you promote the referent simply by setting that zero to a one.



You have to do a transitive walk to make sure all reachable objects are promoted.

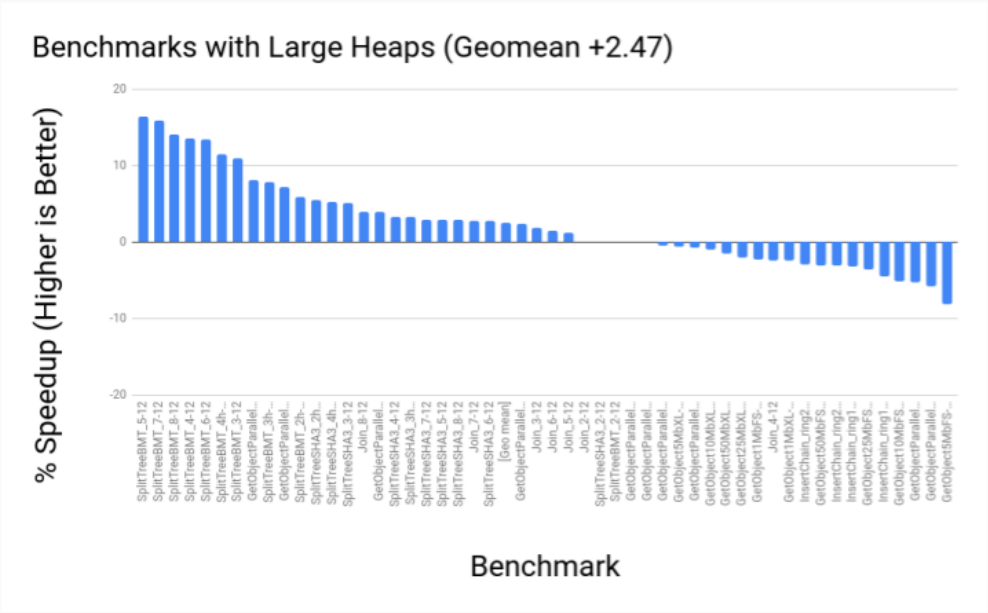


When all reachable objects have been promoted the minor GC terminates.



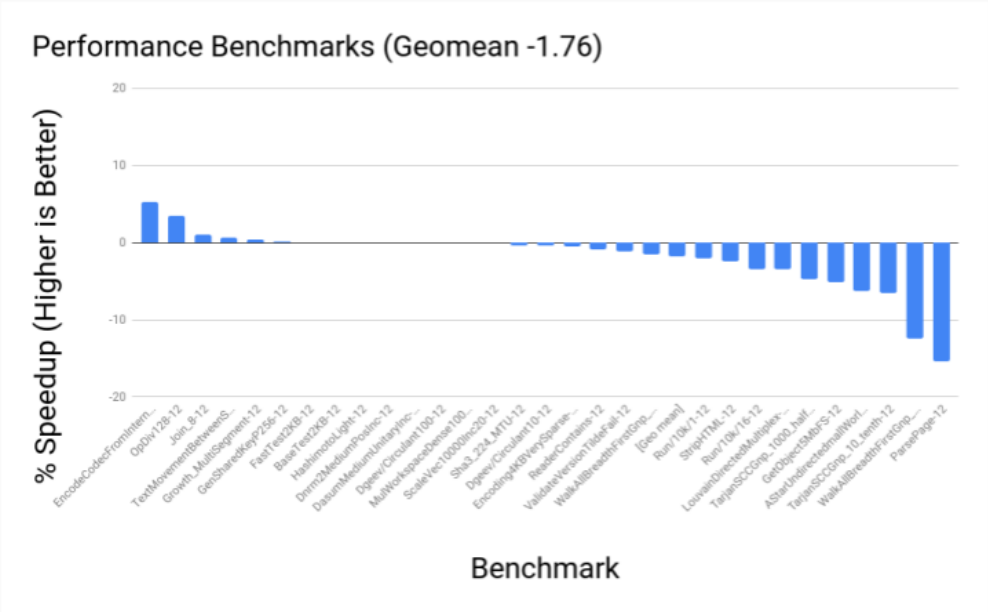
Finally, to finish your generational GC cycle you simply set the current pointer back to the start of the vector and you can continue. All the zeros weren't reached during that GC cycle so are free and can be reused. As many of you know this is called 'sticky bits' and was invented by Hans Boehm and his colleagues.

Winning is hard

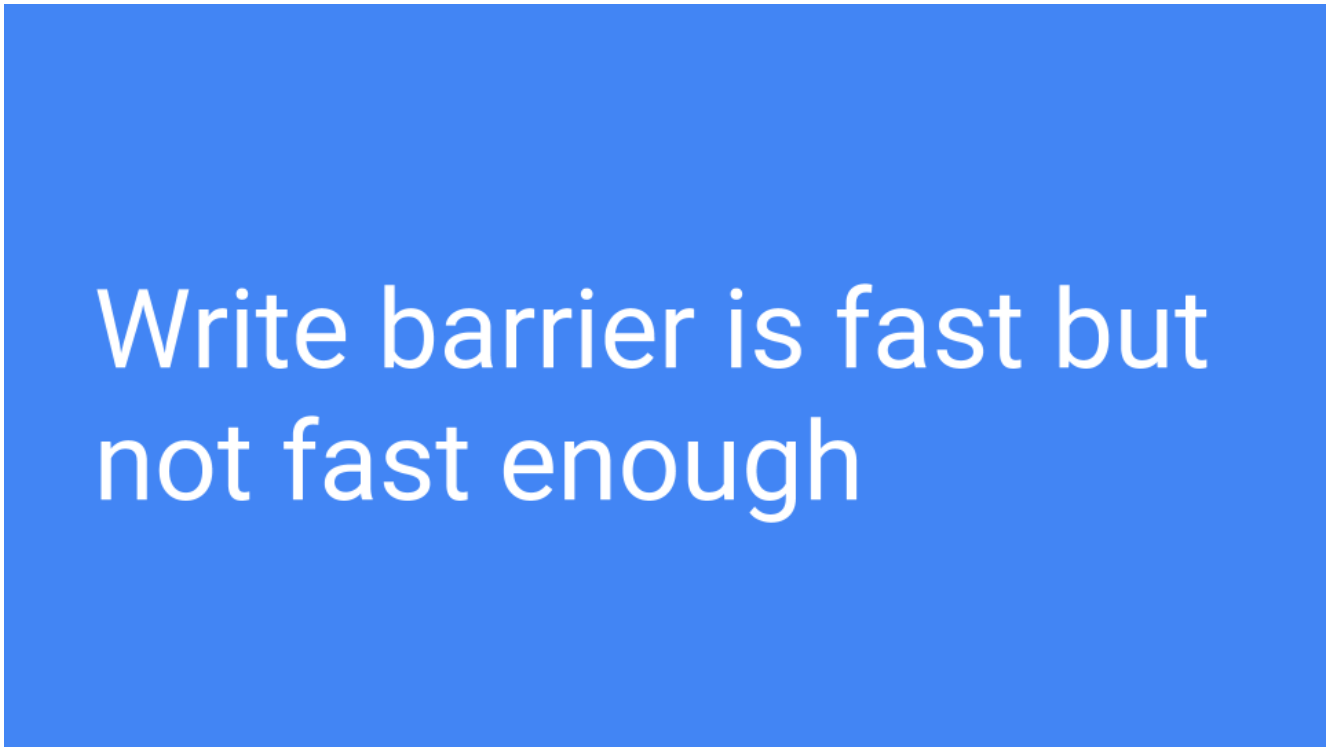


So what did the performance look like? It wasn't bad for the large heaps. These were the benchmarks that the GC should do well on. This was all good.

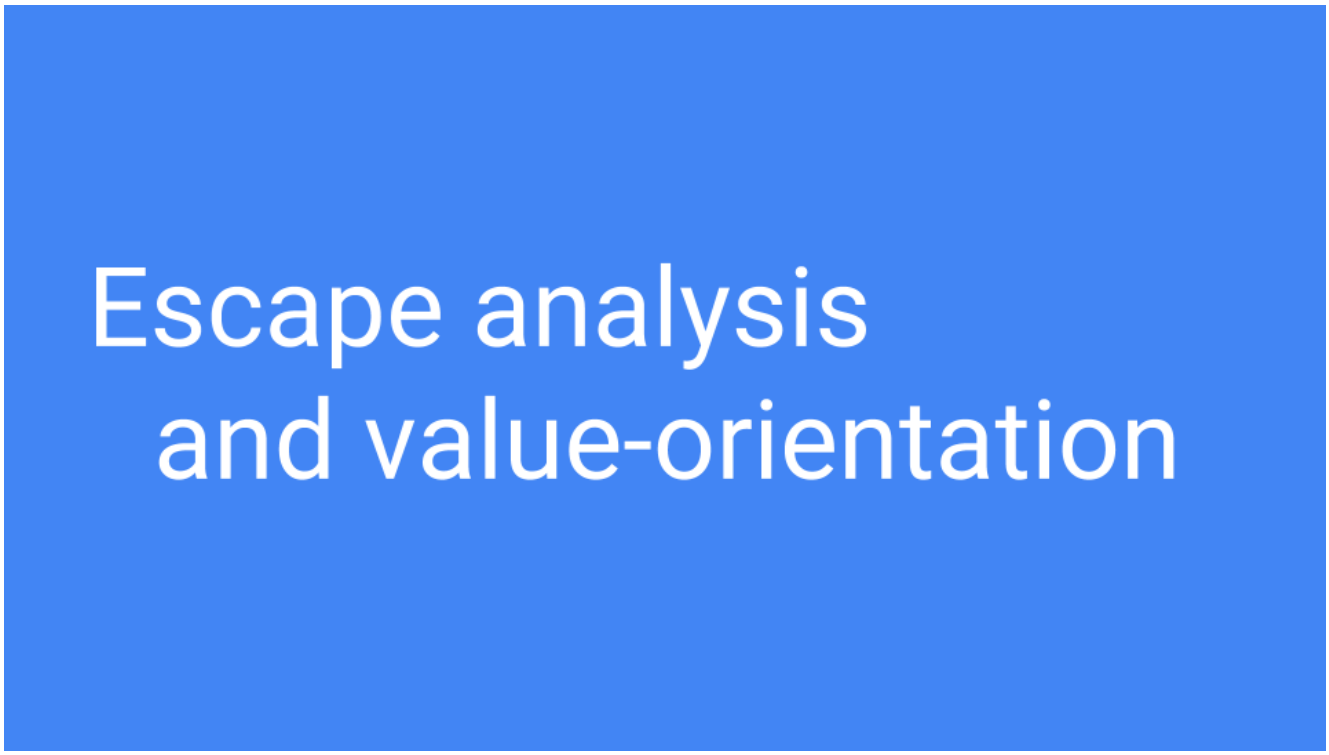
Winning can get harder



We then ran it on our performance benchmarks and things didn't go as well. So what was going on?



The write barrier was fast but it simply wasn't fast enough. Furthermore it was hard to optimize for. For example, write barrier elision can happen if there is an initializing write between when the object was allocated and the next safepoint. But we were having to move to a system where we have a GC safepoint at every instruction so there really wasn't any write barrier that we could elide going forward.

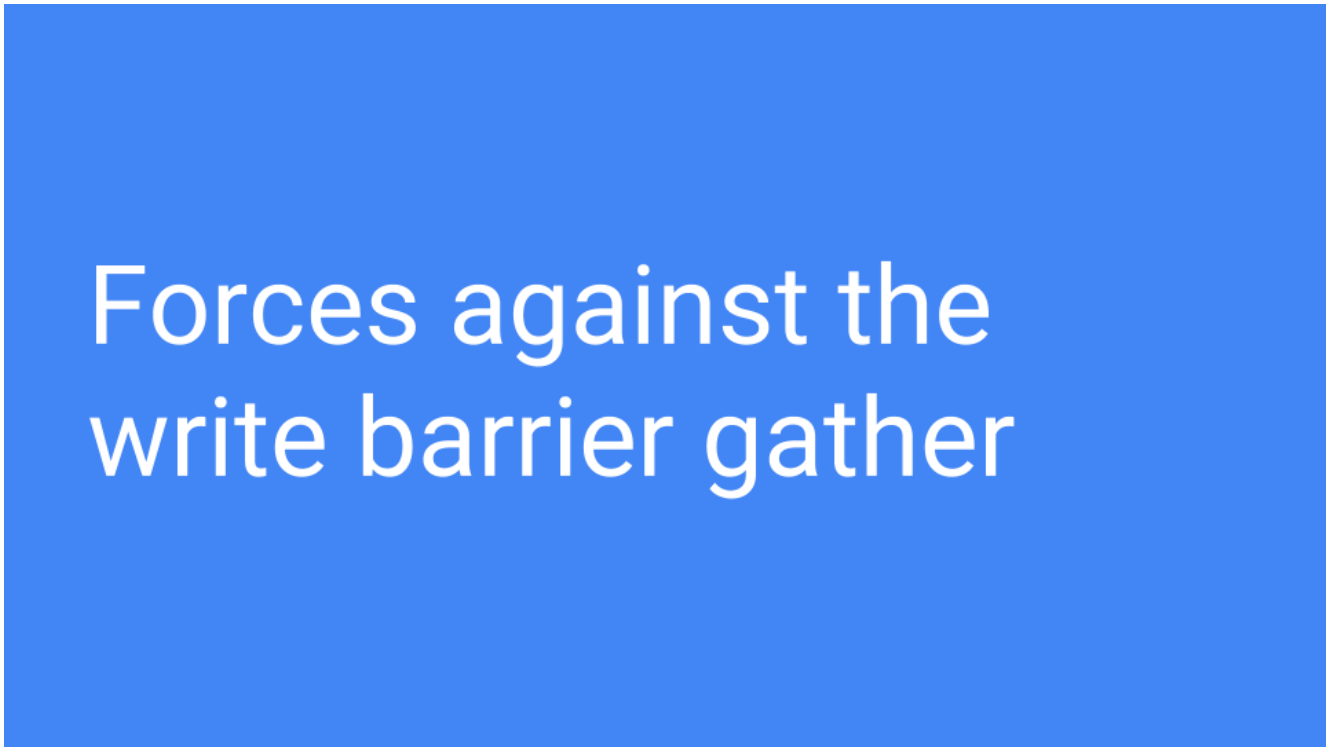


We also had escape analysis and it was getting better and better. Remember the value-oriented stuff we were talking about? Instead of passing a pointer to a function we would pass the actual value. Because we were passing a value, escape analysis would only have to do intraprocedural escape analysis and not interprocedural analysis.

Of course in the case where a pointer to the local object escapes, the object would be heap allocated.

It isn't that the generational hypothesis isn't true for Go, it's just that the young objects live and die young on the stack. The result is that generational collection is much less effective than you might find

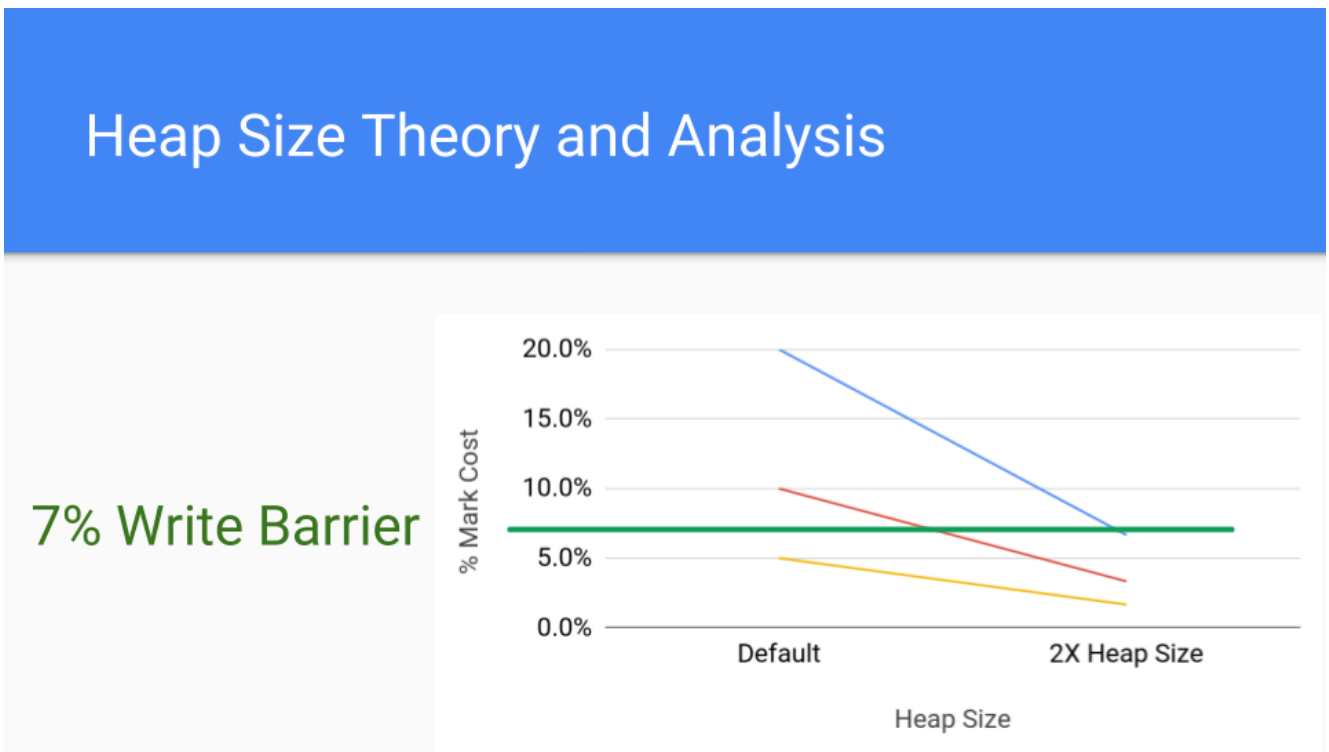
in other managed runtime languages.



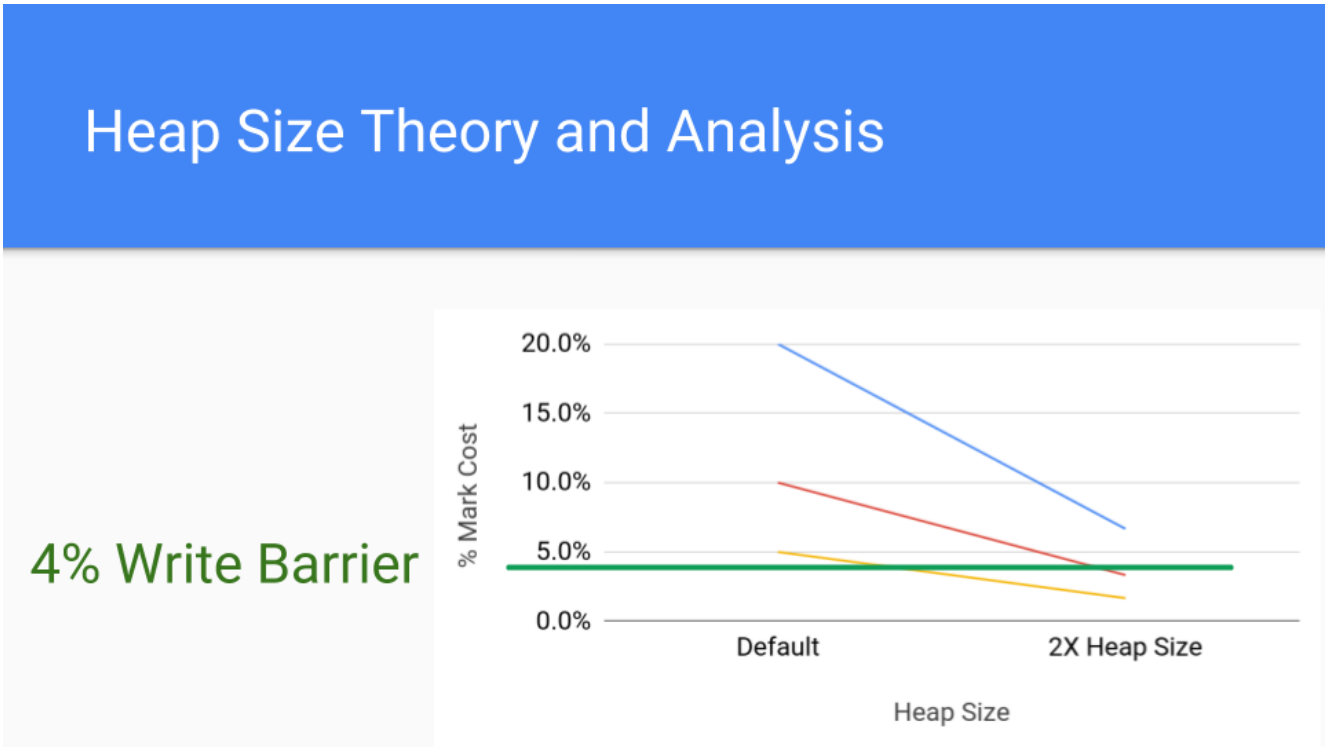
So these forces against the write barrier were starting to gather. Today, our compiler is much better than it was in 2014. Escape analysis is picking up a lot of those objects and sticking them on the stack-objects that the generational collector would have helped with. We started creating tools to help our users find objects that escaped and if it was minor they could make changes to the code and help the compiler allocate on the stack.

Users are getting more clever about embracing value-oriented approaches and the number of pointers is being reduced. Arrays and maps hold values and not pointers to structs. Everything is good.

But that's not the main compelling reason why write barriers in Go have an uphill fight going forward.



Let's look at this graph. It's just an analytical graph of mark costs. Each line represents a different application that might have a mark cost. Say your mark cost is 20%, which is pretty high but it's possible. The red line is 10%, which is still high. The lower line is 5% which is about what a write barrier costs these days. So what happens if you double the heap size? That's the point on the right. The cumulative cost of the mark phase drops considerably since GC cycles are less frequent. The write barrier costs are constant so the cost of increasing the heap size will drive that marking cost underneath the cost of the write barrier.



Here is a more common cost for a write barrier, which is 4%, and we see that even with that we can drive the cost of the mark barrier down below the cost of the write barrier by simply increasing the heap size.

The real value of generational GC is that, when looking at GC times, the write barrier costs are ignored since they are smeared across the mutator. This is generational GC's great advantage, it greatly reduces the long STW times of full GC cycles but it doesn't necessarily improve throughput. Go doesn't have this stop the world problem so it had to look more closely at the throughput problems and that is what we did.

That's a lot of failure and with such failure comes food and lunch. I'm doing my usual whining "Gee wouldn't this be great if it wasn't for the write barrier."

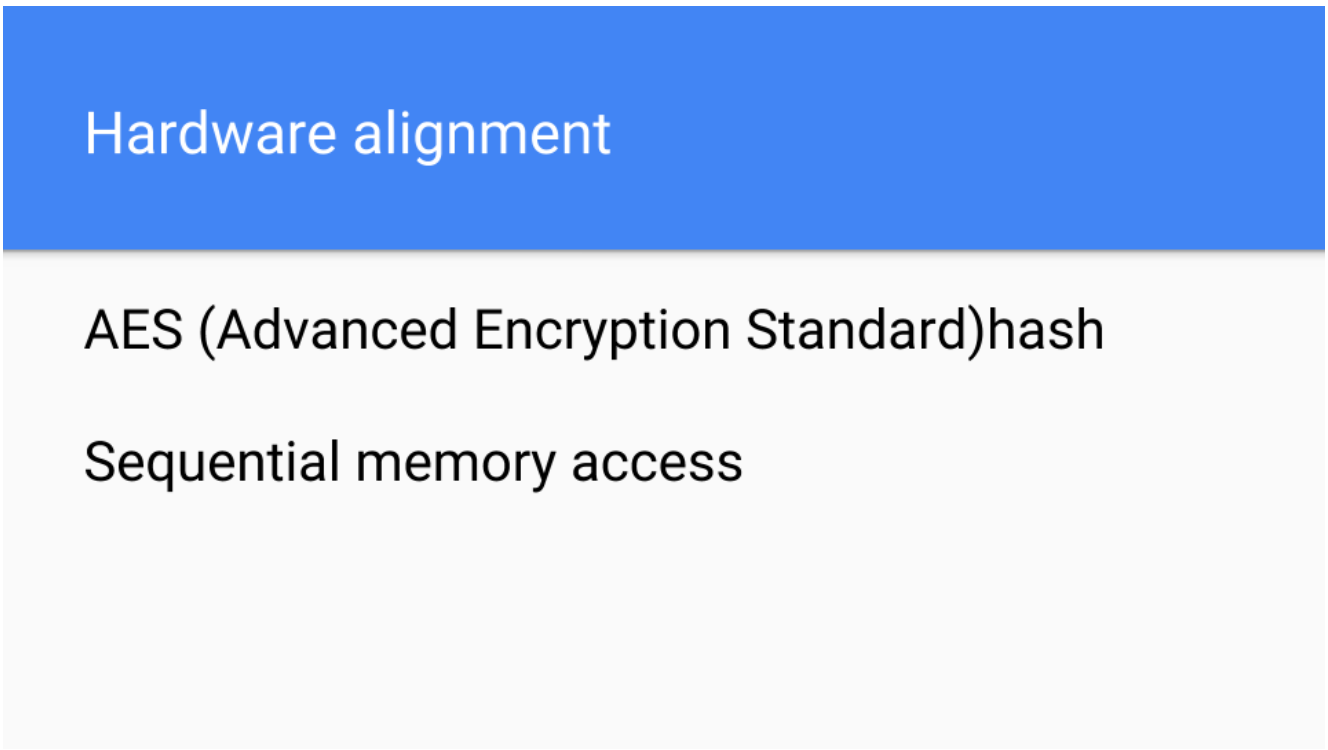
Meanwhile Austin has just spent an hour talking to some of the HW GC folks at Google and he was saying we should talk to them and try and figure out how to get HW GC support that might help. Then I started telling war stories about zero-fill cache lines, restartable atomic sequences, and other things that didn't fly when I was working for a large hardware company. Sure we got some stuff into a chip called the Itanium, but we couldn't get them into the more popular chips of today. So the moral of the story is simply to use the HW we have.

Anyway that got us talking, what about something crazy?

What about card marking without a write barrier? It turns out that Austin has these files and he writes into these files all of his crazy ideas that for some reason he doesn't tell me about. I figure it is some

sort of therapeutic thing. I used to do the same thing with Eliot. New ideas are easily smashed and one needs to protect them and make them stronger before you let them out into the world. Well anyway he pulls this idea out.

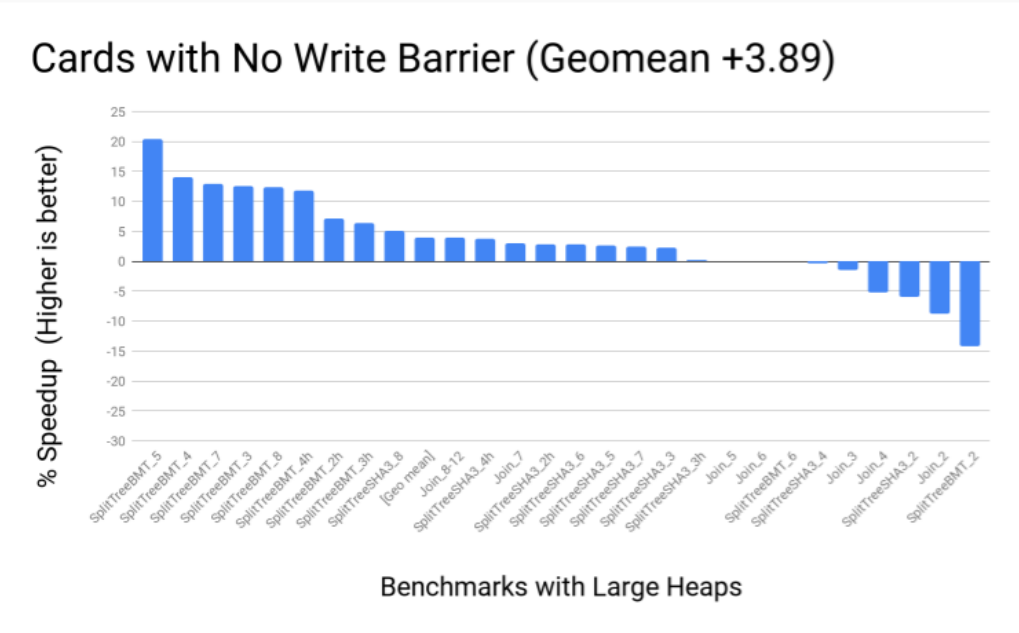
The idea is that you maintain a hash of mature pointers in each card. If pointers are written into a card, the hash will change and the card will be considered marked. This would trade the cost of write barrier off for cost of hashing.



But more importantly it's hardware aligned.

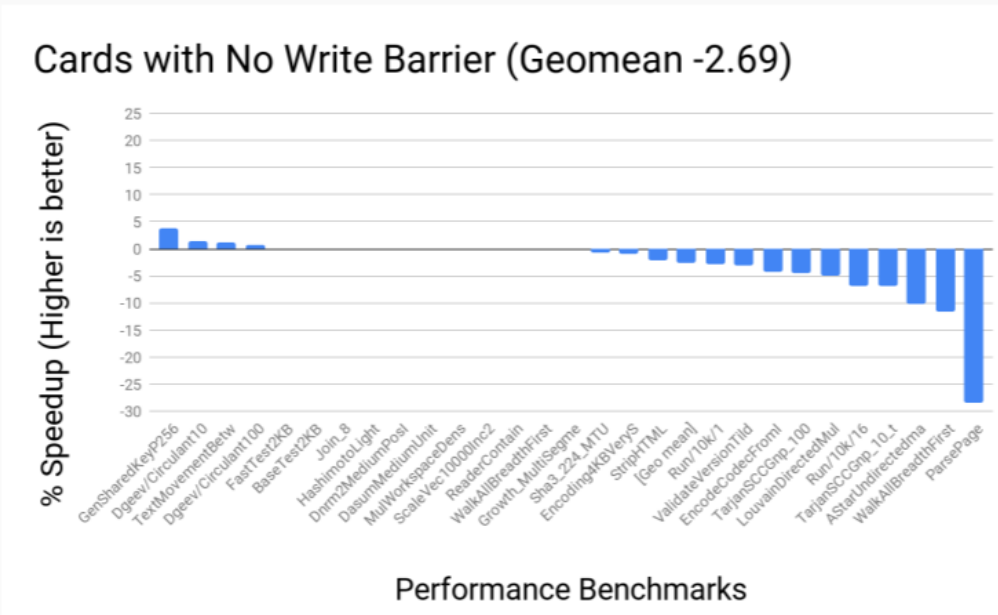
Today's modern architectures have AES (Advanced Encryption Standard) instructions. One of those instructions can do encryption-grade hashing and with encryption-grade hashing we don't have to worry about collisions if we also follow standard encryption policies. So hashing is not going to cost us much but we have to load up what we are going to hash. Fortunately we are walking through memory sequentially so we get really good memory and cache performance. If you have a DIMM and you hit sequential addresses, then it's a win because they will be faster than hitting random addresses. The hardware prefetchers will kick in and that will also help. Anyway we have 50 years, 60 years of designing hardware to run Fortran, to run C, and to run the SPECint benchmarks. It's no surprise that the result is hardware that runs this kind of stuff fast.

Promising Results



We took the measurement. This is pretty good. This is the benchmark suite for large heaps which should be good.

What Doesn't Kill You May Still Hurt

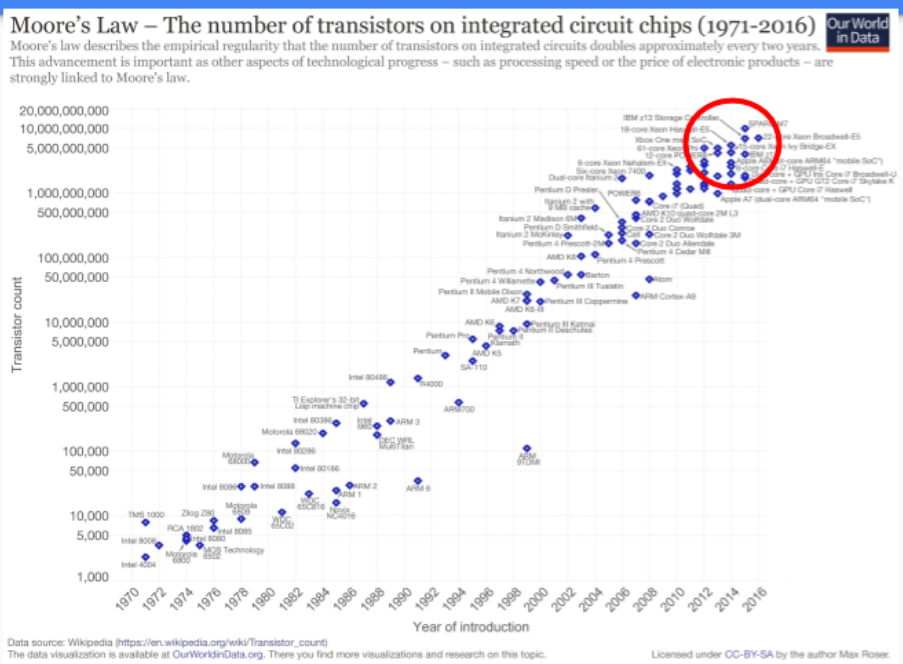


We then said what does it look like for the performance benchmark? Not so good, a couple of outliers. But now we have moved the write barrier from always being on in the mutator to running as part of the GC cycle. Now making a decision about whether we are going to do a generational GC is delayed until the start of the GC cycle. We have more control there since we have localized the card work. Now that we have the tools we can turn it over to the Pacer, and it could do a good job of dynamically cutting off programs that fall to the right and do not benefit from generational GC. But is this going to win going forward? We have to know or at least think about what hardware is going to look like going forward.

Memories of the future ...

What are the memories of the future?

Moore's Law Standard Graph



Let's take a look at this graph. This is your classic Moore's law graph. You have a log scale on the Y axis showing the number of transistors in a single chip. The X-axis is the years between 1971 and 2016. I will note that these are the years when someone somewhere predicted that Moore's law was dead.

Dennard scaling had ended frequency improvements ten years or so ago. New processes are taking longer to ramp. So instead of 2 years they are now 4 years or more. So it's pretty clear that we are entering an era of the slowing of Moore's law.

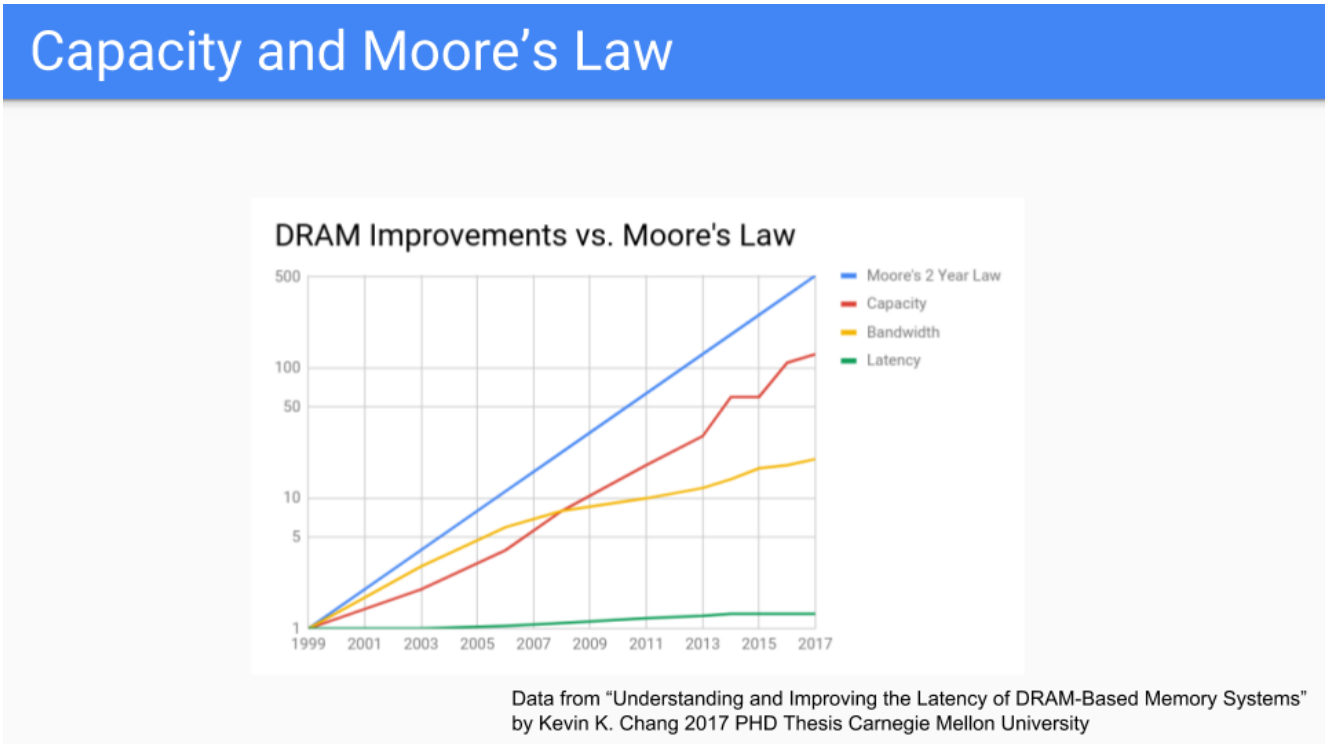
Let's just look at the chips in the red circle. These are the chips that are the best at sustaining Moore's law.

They are chips where the logic is increasingly simple and duplicated many times. Lots of identical cores, multiple memory controllers and caches, GPUs, TPUs, and so forth.

As we continue to simplify and increase duplication we asymptotically end up with a couple of wires, a transistor, and a capacitor. In other words a DRAM memory cell.

Put another way, we think that doubling memory is going to be a better value than doubling cores.

Original graph at www.kurzweilai.net/ask-ray-the-future-of-moores-law.

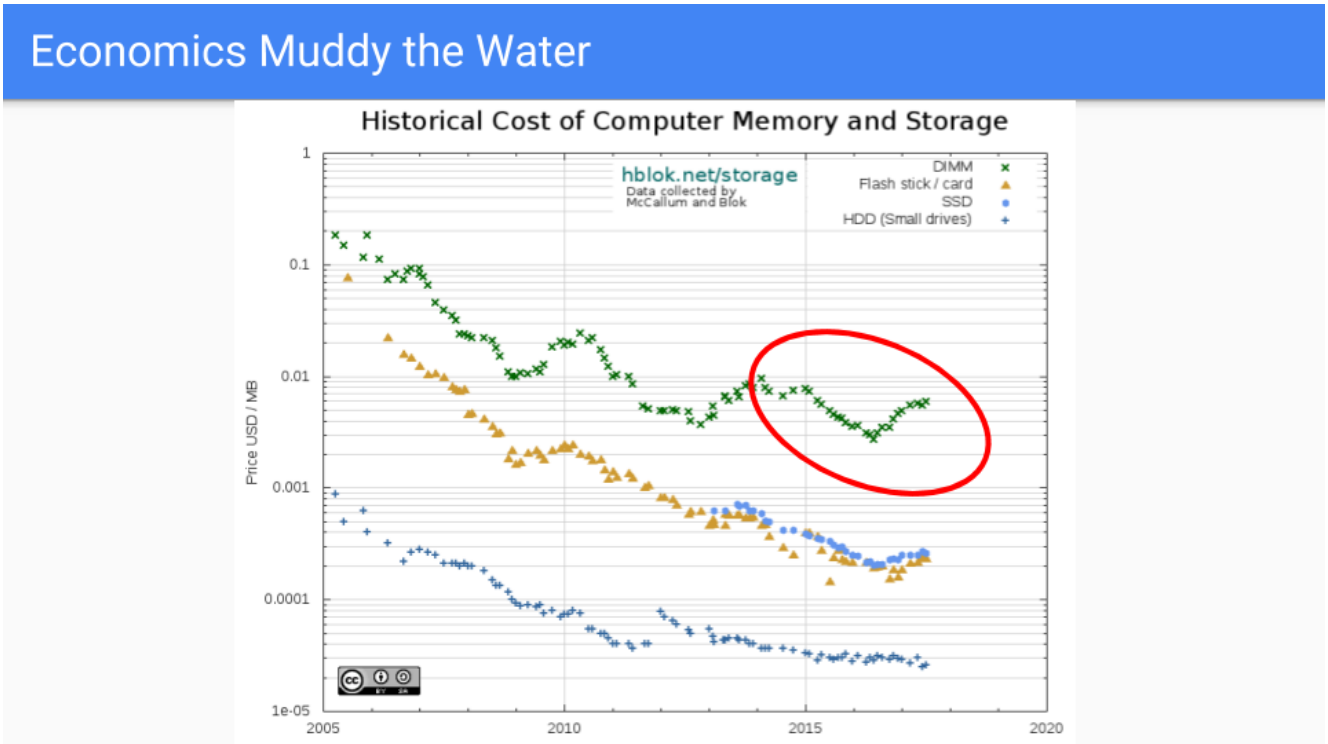


Let's look at another graph focused on DRAM. These are numbers from a recent PhD thesis from CMU. If we look at this we see that Moore's law is the blue line. The red line is capacity and it seems to be following Moore's law. Oddly enough I saw a graph that goes all the way back to 1939 when we were using drum memory and that capacity and Moore's law were chugging along together so this graph has been going on for a long time, certainly longer than probably anybody in this room has been alive.

If we compare this graph to CPU frequency or the various Moore's-law-is-dead graphs, we are led to the conclusion that memory, or at least chip capacity, will follow Moore's law longer than CPUs. Bandwidth, the yellow line, is related not only to the frequency of the memory but also to the number of pins one can get off of the chip so it's not keeping up as well but it's not doing badly.

Latency, the green line, is doing very poorly, though I will note that latency for sequential accesses does better than latency for random access.

(Data from "Understanding and Improving the Latency of DRAM-Based Memory Systems Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering Kevin K. Chang M.S., Electrical & Computer Engineering, Carnegie Mellon University B.S., Electrical & Computer Engineering, Carnegie Mellon University Carnegie Mellon University Pittsburgh, PA May, 2017". See Kevin K. Chang's thesis. The original graph in the introduction was not in a form that I could draw a Moore's law line on it easily so I changed the X-axis to be more uniform.)



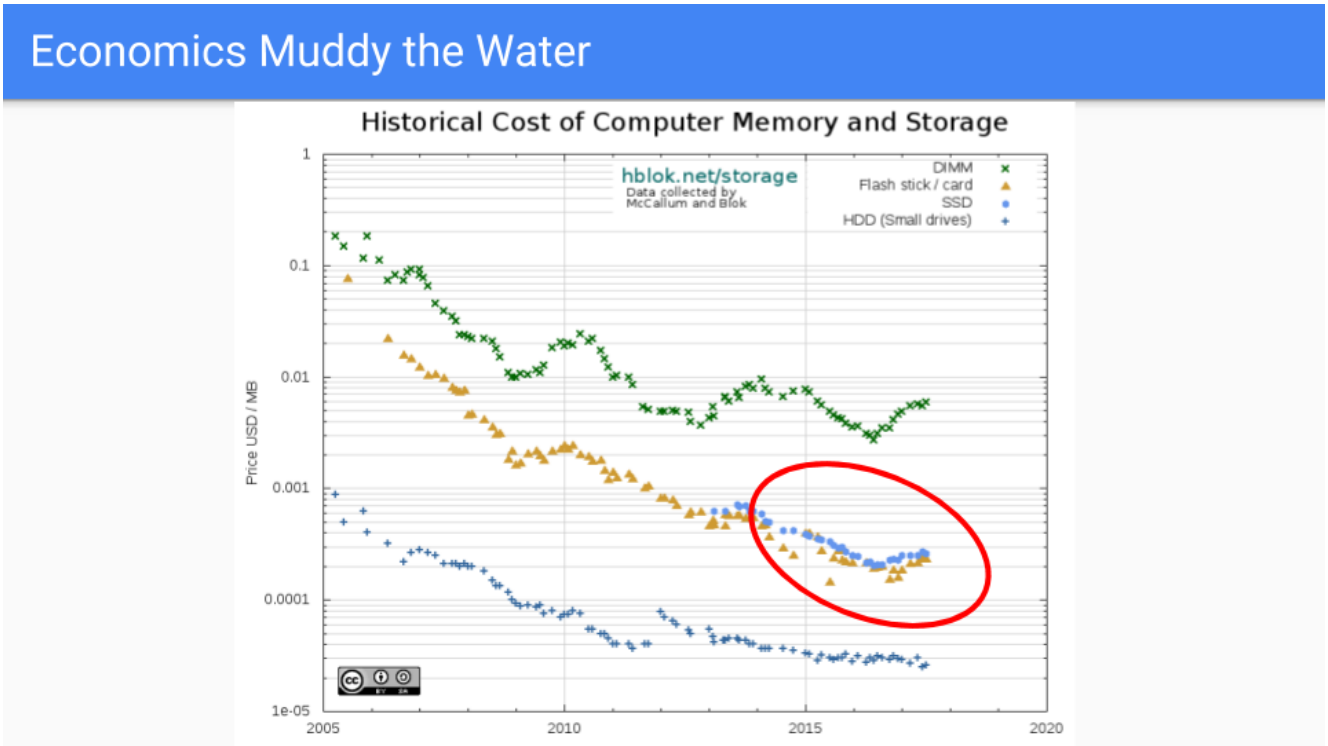
Let's go to where the rubber meets the road. This is actual DRAM pricing and it has generally declined from 2005 to 2016. I chose 2005 since that is around the time when Dennard scaling ended and along with it frequency improvements.

If you look at the red circle, which is basically the time our work to reduce Go's GC latency has been going on, we see that for the first couple of years prices did well. Lately, not so good, as demand has exceeded supply leading to price increases over the last two years. Of course, transistors haven't gotten bigger and in some cases chip capacity has increased so this is driven by market forces. RAMBUS and other chip manufacturers say that moving forward we will see our next process shrink in the 2019-2020 time frame.

I will refrain from speculating on global market forces in the memory industry beyond noting that pricing is cyclic and in the long term supply has a tendency to meet demand.

Long term, it is our belief that memory pricing will drop at a rate that is much faster than CPU pricing.

(Sources https://hblok.net/blog/ and https://hblok.net/storage_data/storage_memory_prices_2005-2017-12.png)



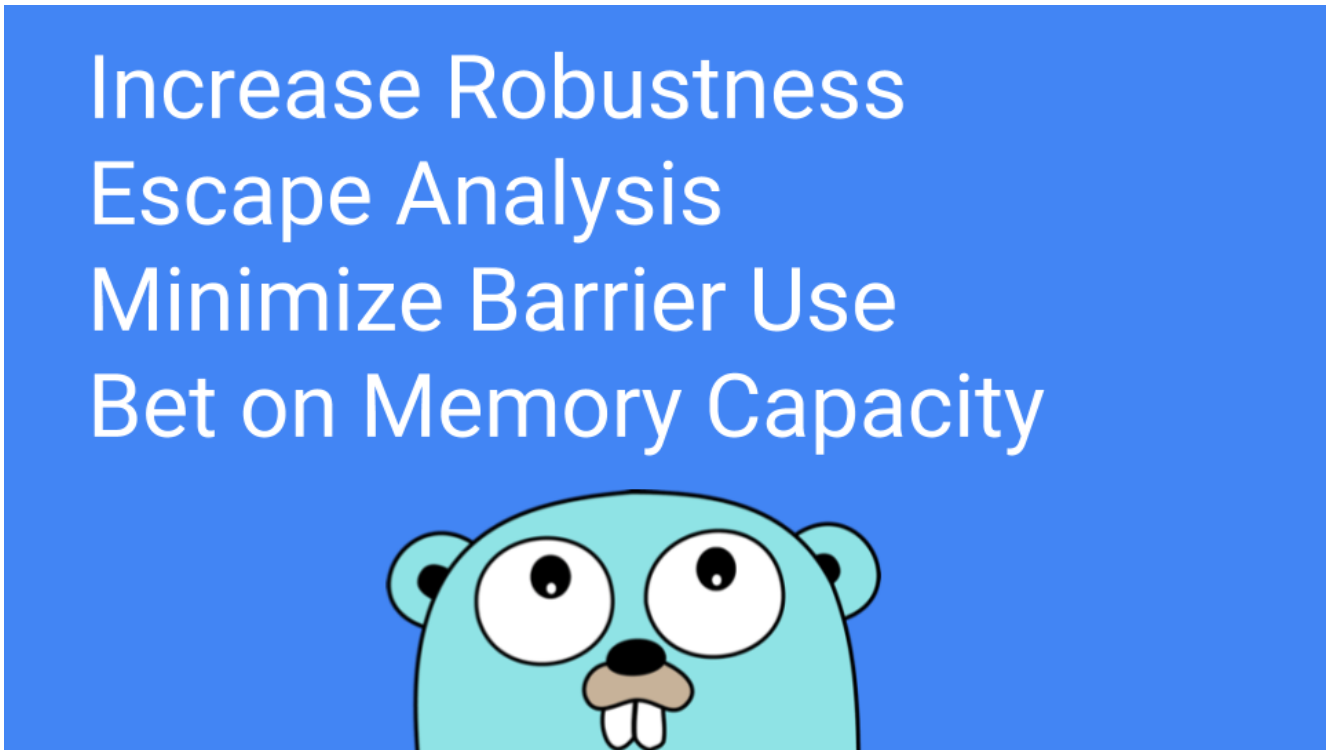
Let's look at this other line. Gee it would be nice if we were on this line. This is the SSD line. It is doing a better job of keeping prices low. The material physics of these chips is much more complicated that with DRAM. The logic is more complex, instead of a one transistor per cell there are half a dozen or so.

Going forward there is a line between DRAM and SSD where NVRAM such as Intel's 3D XPoint and Phase Change Memory (PCM) will live. Over the next decade increased availability of this type of memory is likely to become more mainstream and this will only reinforce the idea that adding memory is the cheap way to add value to our servers.

More importantly we can expect to see other competing alternatives to DRAM. I won't pretend to know which one will be favored in five or ten years but the competition will be fierce and heap memory will move closer to the highlighted blue SSD line here.

All of this reinforces our decision to avoid always-on barriers in favor of increasing memory.

So what does all this mean for Go going forward?



We intend to make the runtime more flexible and robust as we look at corner cases that come in from our users. The hope is to tighten the scheduler down and get better determinism and fairness but we don't want to sacrifice any of our performance.

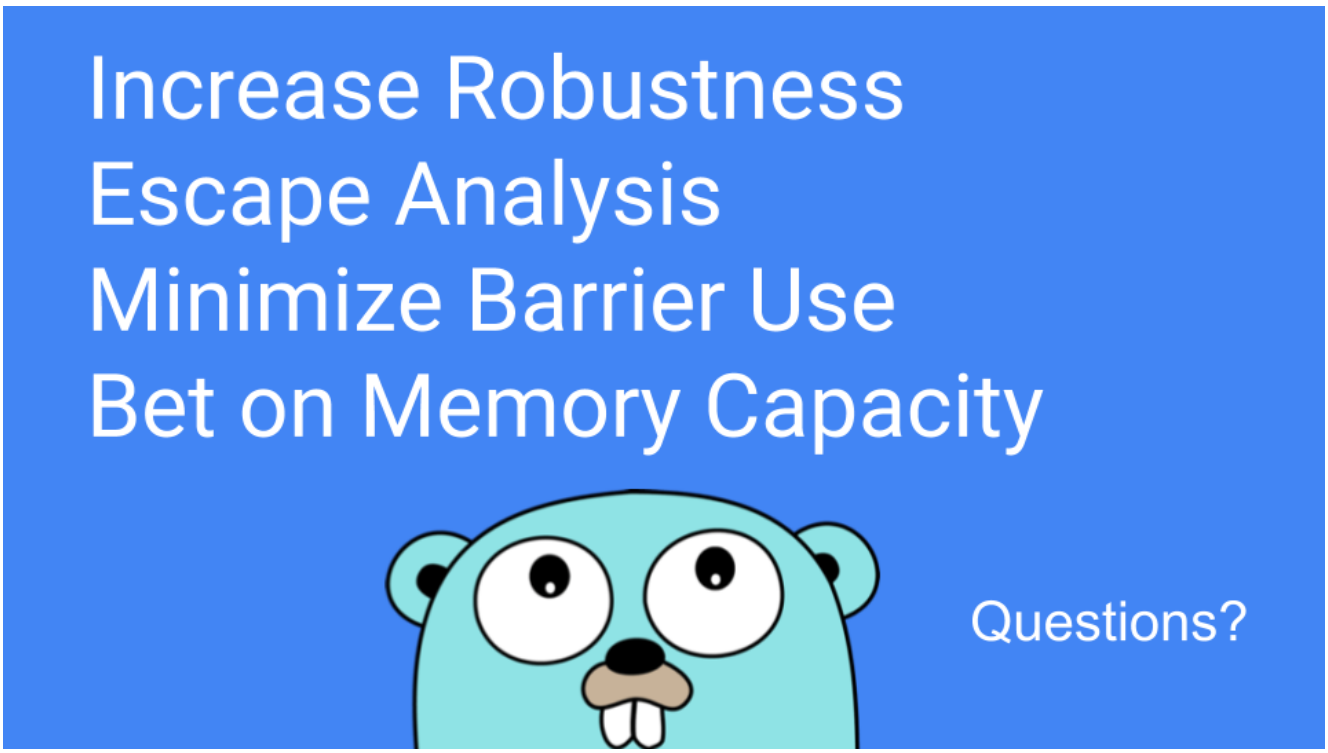
We also do not intend to increase the GC API surface. We've had almost a decade now and we have two knobs and that feels about right. There is not an application that is important enough for us to add a new flag.

We will also be looking into how to improve our already pretty good escape analysis and optimize for Go's value-oriented programming. Not only in the programming but in the tools we provide our users.

Algorithmically, we will focus on parts of the design space that minimize the use of barriers, particularly those that are turned on all the time.

Finally, and most importantly, we hope to ride Moore's law's tendency to favor RAM over CPU certainly for the next 5 years and hopefully for the next decade.

So that's it. Thank you.



P.S. The Go team is looking to hire engineers to help develop and maintain the Go runtime and compiler toolchain.

Interested? Have a look at our open positions.