ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Review Article

Open Access

Improve the Latency of Go Applications while using GOGC

Pallavi Priya Patharlagadda

United States of America

ABSTRACT

Go is a concurrent, garbage-collected, statically typed programming language that was developed at Google in 2009. Because of its straightforward, effective, and low learning curve, it is a well-liked option for developing online applications, command-line tools, and scalable network services. Garbage collection, which takes care of memory management automatically for you, is another key component of Go. As a result, there is no longer a requirement for manual memory management, which lowers the possibility of memory leaks and other issues. But it does come with a cost, as the Garbage collector takes CPU and Memory. In this paper, we will run some performance analysis with different GOGC options.

***Corresponding author**

Pallavi Priya Patharlagadda, United States of America.

Received: November 15, 2022; **Accepted:** November 22, 2022, **Published:** November 29, 2022

Introduction

Any programming language stores its values in physical memory. Since physical memory is limited, it must be properly managed and recycled to prevent out-of-memory situations. If an allocated memory space is no longer needed, then that memory needs to be deallocated so that further allocation can be done on the same space. Garbage Collection is the term used to describe this memory reuse process. So, anything created in memory that is useless or no longer needed is referred to as garbage. Usually, it is the developer's responsibility to clean up the data in memory once the action is performed.

Automatic garbage collection is the term used for collecting garbage (unused or no longer needed data) that is carried out mechanically without the need for human participation. A system called the Garbage Collector was created explicitly to trace that memory and release dynamically generated memory. There is always a cost associated with automatic garbage collection that exceeds the program's efficiency. Go provides support for Automatic Garbage collection.

Problem Statement

Every Go application comes with a runtime library that has Garbage collector in it. The frequency at which the GC should be run can be configured using either the GOGC environment variable (which all Go programs recognize) or through the SetGCPercent API in the runtime/debug package.

Go Garbage collector uses two important system resources, like CPU time and memory. In this paper, we do some performance analysis and provide the factors that need to be considered for setting the GOGC frequency.

Variables Storage in go

Go stores its local variables and functions in a LIFO data structure called stack. A new stack frame containing all the function's local variables is allocated each time it is called. The stack frame of the function is deallocated when it has completed running, freeing up memory for further usage. The stack has a limited size and local scope, but it is quick and offers automatic memory management. In general, all the static data gets stored in stacks. In terms of the stack, Go employs a method known as split stacks, or stack segmenting. Go begins with tiny stacks that could dynamically grow and shrink, in contrast to certain languages where the stack size needs to be specified at thread creation. Every goroutine begins with a small stack of around 2 KB that expands, and contracts as needed.

For storing the dynamic variables, heap is used. heap is a section of memory that is not inherently ordered or structured like the stack. The memory blocks can be allocated and deallocated whenever needed. Slower access times and manual memory management are the price of this flexibility. Data that must survive the lifetime of the function is allocated on the heap.

If the size of the variable is dynamically determined or Go compiler cannot determine a variable's lifetime, then the variables escape to the heap. For example, consider the backing array of a slice whose initial size is not fixed but rather varies. Note that escape to the heap needs to be transitive as well. This means that if a Go value is referenced by another Go value that has previously been found to escape, the other value needs to escape as well. The context in which a Go value is used and the escape analysis process of the Go compiler determine whether it escapes to heap or not. Trying to pinpoint exactly when values escape would be risky and challenging because the mechanism is complex and varies with each release of Go.

Garbage Collection in go

Go uses a concurrent, tricolor, mark-and-sweep algorithm. Because of this design, Go's GC can ensure effective memory management without interfering with the application's performance. Let's discuss this in detail.

Concurrent

Go Garbage Collector runs in parallel with the application. The word "concurrent" refers to the fact that the garbage collection procedure does not halt the application's execution. To inspect and recover memory, traditional garbage collectors frequently include a "stop-the-world" phase in which program execution stops completely. But this method can cause performance impacts to halt noticeably and have a negative impact on high-throughput or real-time systems.

In Contrast, Go's GC is made to function in tandem with the application. This implies that the Go scheduler manages both the application and garbage collector scheduling when a Go program executes. Go Scheduler works in a similar way as it would if it had a standard application with numerous goroutines. Most of the garbage collector's job is completed concurrently with the application's execution in the background. This results in shorter stop-the-world pauses and enhances the latency profile of Go programs.

Tricolor

The "tricolor" used by Go's GC marking algorithm considers objects (or blocks of memory) in three different states: white, grey, and black.

- White items are ones that haven't been processed by the garbage collector. These are the objects directly accessible by the program, such as global variables or the local variables of the function that is presently executing, which may or may not be reachable from the roots.

- The garbage collector has identified certain things as "grey" if they can be reached from the roots; however, the objects they relate to, which are their descendants, have not yet undergone processing. If the scan finds a specific object has one or more pointers to a white object, it puts that white object in the grey set.

- Black objects are ones that have been fully processed by the garbage collector; both the object and all its offspring have been located and determined to be reachable.

At first, every object is colored white. The garbage collector marks the roots as grey, starting at the roots. Next, it processes every grey item, looking for references to other objects within it. The garbage collector makes a referred object grey if it is white. An object is colored black by the garbage collector after it has been processed.

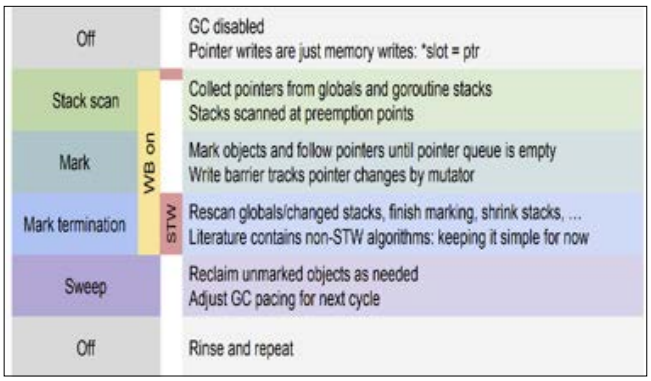
This tri-color technique helps the garbage collector locate and effectively recover unreachable memory by guaranteeing a clear separation of objects according to their reachability status. Since the GOGC runs in parallel with the application, how can we make sure that we are not entering a race condition where the garbage collector sweeps an object that is in use by the program? Write Barrier helps in preventing this race condition. This is a mechanism that makes sure the properties of the tri-color abstraction are maintained while the algorithm is in progress. When a pointer that references a white (not yet processed) object is written to a black (already processed) object, the garbage collector ensures the white object is marked as gray, preventing it from being prematurely collected.

Mark and Sweep

The Mark and Sweep terms specify the two-phase approach to reclaiming the unused or no longer needed memory by Go's GC.

- **Mark phase:** The Garbage collector goes across the object graph in this phase, beginning at the roots. It finds every reachable object by applying the tri-color marking procedure, as previously mentioned. The mark phase runs alongside the program, dividing marking tasks into intervals between Goroutine executions.
- **Sweep phase:** The sweep phase starts when all things that are within reach are tagged (in black). The Garbage collector recovers the memory that white (unreachable) objects have taken up during this stage. This stage, which cleans up RAM bit by bit, also occurs in tandem with the execution of goroutines.

It's not possible to release memory to be allocated until all memory has been traced, because there may still be an unscanned pointer keeping an object alive. Hence, sweeping must be separated from the process of marking. During the marking phase, the garbage collector marks data actively used by the application as a live heap. Then, during the sweeping phase, the GC traverses all the memory not marked as live and reuses it.



Advantages of Concurrent Go GC

- The application is not totally stopped while Garbage collector is running. So, applications get very low pause times.
- **Disadvantages of concurrent Go GC:**
- GC throughput:
- The amount of time required to remove the garbage increases with increase in the size of heap. This isn't false unless your program doesn't use any parallelism and you can keep sending cores to the GC indefinitely.
- **Compaction:**
- Since there isn't any compaction, the program's heap may eventually fragment.
- **Application throughput:**
- The application itself gets slowed down since the GC must work hard on every cycle, using up the CPU time.
- **Pause Distribution:**
- Since GOGC runs in parallel with the application, sometimes application creates garbage faster than the GC routines that can clean up. The only option left for the runtime in this situation is to completely halt your program and wait for the GC cycle to finish. Therefore, Go's assertion that GC pauses are extremely rare can only be verified if the GC has enough CPU time and headroom to outpace the main program.

GOGC

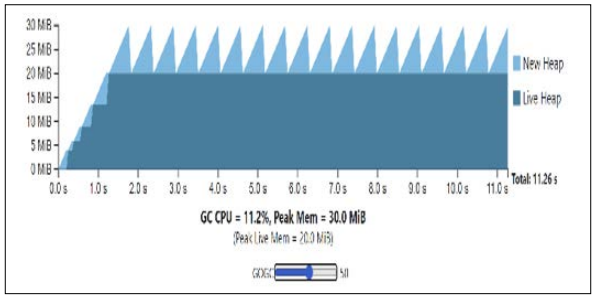
One of the oldest environment variables supported by the Go runtime is GOGC. The garbage collector's level of aggression is managed by GOGC. This number is assumed to be 100 by default, meaning that garbage collection won't start until the heap has expanded by 100% since the last collection. The default setting of GOGC=100 effectively instructs the garbage collector to start every time the live heap doubles.

If the GOGC value is set to a higher value, let's say GOGC = 200, the garbage collection cycle won't begin until the live heap has expanded to 200% of its former size. Setting the value lower-for example, GOGC = 20—will result in more frequent garbage collector triggers. The key takeaway is that doubling GOGC will double heap memory overheads and roughly halve GC CPU costs. Garbage collection will be completely disabled if GOGC is turned off. The frequency of the garbage collector can be configured through the GOGC environment variable or through the SetGCPercent API of the runtime/debug package.

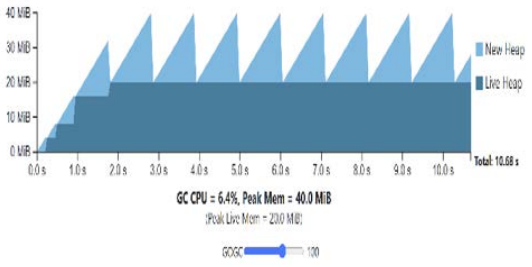
By including a runtime.GC statement in Go code, one can manually start garbage collection in Go. But note that runtime. GC may stop the caller and may even cause the application to crash, particularly if you are running a heavily loaded Go program. This mostly occurs because it is impossible to collect garbage when everything else is changing quickly. If you do, the garbage collector won't be able to distinguish between the members of the white, black, and gray sets. This garbage collection status is also called garbage collection safe point.

The applications whose non-GC work requires 10 seconds of CPU time to finish are shown in the graph below. Before reaching a stable state, it goes through a few setup stages (increasing its live heap) in the first second. The application runs within a container with a capacity of just over 60 MB. The application consumes 20 MB of live memory in steady state. It is assumed that the program utilizes no additional memory and that the live heap contains all pertinent GC operations that must be completed.

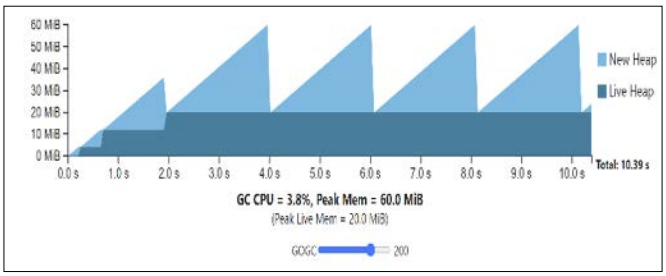
As the new heap approaches zero, each GC cycle comes to an end. The total time for cycle N's mark phase and cycle N+1's sweep phase is the amount of time it takes for the new heap to decline to zero. Please note that all the visualizations in this graph assume that the application is paused while the GC runs. As a result, the time it takes for new heap memory to drop to zero completely represents the GC CPU costs. The same understanding still holds true; this is just to make visualization easier. X-axis represents the CPU time. Observe that the total period is longer due to the GC using more CPU time. Below graphs represent application CPU time and live memory when GOGC is set to different values



GoGC = 50, GC CPU = 11.2%, Peak Memory = 30.0 MiB, Total CPU time = 11.26s



GoGC = 100, GC CPU = 6.4%, Peak Mem = 40.0 MiB, Total: 10.68s



GoGC = 200, GC CPU = 3.8%, Peak Mem = 60.0 MiB, Total: 10.39s

From the graph, we can see that CPU overhead falls as GOGC rises, while peak memory rises in direct proportion to the live heap size. Peak memory requirements drop as GOGC drops, but at the cost of increased CPU overhead.

Latency With GOGC

GOGC is not fully stop-the-world and does most of its work concurrently with the application. This is primarily to reduce application latencies. However, there are some pauses, which might affect the latency. Below are some of the scenarios.

- Short stop-the-world pauses occur when the GC switches between the mark and sweep phases
- Scheduling delays occur because the GC uses 25% of CPU resources during the mark phase
- Pointer writing necessitates extra work during the mark phase
- Running goroutines will be suspended when GC scans its roots.

Set the GOGC for a Golang Application while Running Thedieseler Container

For testing purposes, I am using an 8-GB Linux-based VM. I have written a Go program where I initialize a sample structure in a loop and print it on the console. The main purpose of this program is to consume huge heap.

Also, to measure the memory allocation strategies and related performance issues, I am using Go Runtime Memory Stats. Go runtime package exposes runtime. ReadMemStats(m *MemStats) that fills a MemStats object. There are a number of fields in that structure, but I am using the below fields.

- **Alloc:** the currently allocated number of bytes on the heap.
- **TotalAlloc:** cumulative maximum bytes allocated on the heap (will not decrease).
- **Sys:** total memory obtained from the OS,
- **NumGC:** number of completed GC cycles

Below is the main. go file

```
package main
import (
    "encoding/json"
    "fmt"
    "runtime"
    "strconv"
)
type Person struct {
    Name string `json:"name,omitempty"`
    Id   int    `json:"id"`
    Sal  int    `json:"sal"`
    Address string `json:"address"`
    AccNumber string `json:"accNumber"`
}
```

```
type Persons struct {
	People []string
}
func main() {
	var ps Persons
	printMemUsage()
	for i := 1; i <= 4000; i++ {
		for j := 1; j <= 4000; j++ {
			name := "kakha_" + strconv.Itoa(i) + strconv.Itoa(j)
			s := Person{Name: name, Id: i, Sal: j*10, Address:
"123 station Drive,california,12345", AccNumber:"ABC12345" }
			js, error := json.Marshal(s)
			if error == nil {
				ps.People = append(ps.People, string(js))
			}
		}
	}
	printMemUsage()
	fmt.Println(ps.People)
	printMemUsage()
}
func printMemUsage() {
	var mem runtime.MemStats
	runtime.ReadMemStats(&mem)
	fmt.Printf("Alloc = %v MiB", bToMb(mem.Alloc))
		fmt.Printf("\tTotalAlloc = %v MiB", bToMb(mem.
TotalAlloc))
	fmt.Printf("\tSys = %v MiB", bToMb(mem.Sys))
	fmt.Printf("\tNumGC = %v\n", mem.NumGC)
}
func bToMb(b uint64) uint64 {
	return b / 1024 / 1024
}
```

To verify if the variables are using heap or stack, try compiling the go file using below command. From the output, we could see the variable is stored in Heap memory.

go build -gcflags="-m" main.go

Command Output:

```
	# command-line-arguments
./main.go:35:13: inlining call to fmt.Println
./main.go:41:12: ... argument does not escape
./main.go:41:36: ~r0 escapes to heap
./main.go:42:43: ~r0 escapes to heap
./main.go:43:36: ~r0 escapes to heap
./main.go:44:32: m. NumGC escapes to heap
./main.go:25:39: "kakha_" + ~r0 + ~r0 escapes to heap
./main.go:28:26: s escapes to heap
./main.go:30:41: string(js) escapes to heap
./main.go:35:13: ... argument does not escape
./main.go:35:16: ps. People escape to heap
```

Once the build is successful, the application can be run using the command: ./main.exe

Caution: If you are running on a host machine with 8 GB of RAM, the system may hang, and you may lose control over your laptop for some time.

Run the Golang Application as a Docker Container

To deploy the Golang application as a Docker container, below is the Dockerfile used.

```
# syntax=docker/dockerfile:1
FROM golang:1.19
WORKDIR /src
COPY main.go .
RUN go build -o /test ./main.go
CMD ["/test"]
```

Command to Build Docker Image

```
	docker build -t gogc_img .
```

After successful docker build, the gogc_img will be created. This can be used for launching docker containers.

Start the Container with Gogc Values

Let's start the container with different GOGC values and see the behavior. I am using journald as the log-driver so that my logs are stored in journald.

Command to start the container with GOGC value as 50.

```
		docker run -e "GOGC=50" --log-driver=journald
-d gogc_img:latest
```

This would start a new container, which would run GOGC whenever the heap grows 50% of the previous.

same test was executed with different GOGC values of 50, 100, 150, and 200 three times. below chart provides more information for different GOGC values.

	Alloc	Total Alloc	Sys Mem	CPU time	No of times GC ran
GOGC=50	6053	17396	10670	1min 14.856s	55
	6053	17396	10894	1min 9.430s	56
	6053	17396	9742	1min 33.079s	56
GOGC=100	7468	17396	10333	49.700s	29
	7468	17396	10105	47.476s	29
	7468	17396	9849	48.154s	28
GOGC=150	8599	17396	11573	43.433s	19
	8599	17396	11549	45.098s	20
	8599	17396	11187	44.808s	20
GOGC=200	7468	17396	11387	39.865s	15
	7468	17396	11141	38.651s	15
	7468	17396	12346	37.352s	14

As we can see from the chart, with increase in GOGC values, the number of times GC runs is reduced, and the execution time is also reduced.

Conclusion

Considering GOGC, below are some of the recommendations on how to improve application throughput and latency.

1. Reducing GC frequency can lead to latency improvements. The default setting may not be suitable for all applications. For example, If the application host has more memory, then GOGC values can be set at a higher value. Try to experiment with different GOGC values on your application at different traffic levels, and then decide on the GOGC configuration that best suits your application.
2. Try to make the application use the smallest heap possible. Less heap corresponds to less GC work.
3. Try to avoid pointers, as they would involve a reference, thereby increasing the scanning times. (For example, instead of having strings as keys that contain pointers, make a hash out of the string and store it.

Even after following the above recommendations, if the application is still facing latency issues, below ideas can be applied to understand the reasons behind them.

1. Understand if any of the variables are escaping to heap. "go build -gcflags=-m=3" provides more information on why the variables are escaping to heap. More heap memory corresponds to more GC work.

2. Use Memory profiling to find the hot spots in the heap allocation.
3. Use CPU Profiling to understand where the CPU is spending time during GC phases. You can try using GC tracing or Execution tracing for it (1-9).

References

1. A Guide to the Go Garbage Collector https://tip.golang.org/doc/gc-guide
2. Understanding Go's Garbage Collection https://bwoff.medium.com/understanding-gos-garbage-collection-415a19cc485c
3. The Go Garbage Collector (GC) https://www.mtsoukalos.eu/Go-Garbage-Collector/
4. https://go.dev/talks/2015/go-gc.pdf
5. Modern garbage collection https://blog.plan99.net/modern-garbage-collection-911ef4f8bd8e
6. https://www.komu.engineer/blogs/01/go-gc-maps
7. Go's garbage collector https://agrim123.github.io/posts/go-garbage-collector.html
8. Gopher Academy Blog https://blog.gopheracademy.com/advent-2018/avoid-gc-overhead-large-heaps/
9. A Guide to the Go Garbage Collector https://tip.golang.org/doc/gc-guide#GOGC