**ISSN:** 2583-9888
**DOI:** doi.org/10.51219/JAIMLD/pallavi-priya-patharlagadda/213

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science
https://urfpublishers.com/journal/artificial-intelligence

Vol: 2 & Iss: 2

*Research Article*

# Optimize Memory Usage of Go Applications by Setting the Memory Limit

**Pallavi Priya Patharlagadda**

**Citation:** Patharlagadda PP. Optimize Memory Usage of Go Applications by Setting the Memory Limit. *J Artif Intell Mach Learn & Data Sci 2024*, 2(2), 881-886. DOI: doi.org/10.51219/JAIMLD/pallavi-priya-patharlagadda/213

**Received:** 03 June, 2024; **Accepted:** 28 June, 2024; **Published:** 30 June, 2024

***Corresponding author:** Pallavi Priya Patharlagadda, Engineering, USA

**Copyright:** © 2024 Sahni BPS., This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

## ABSTRACT

Go is one of the most efficient programming languages, and it mainly contains two components: the Go compiler, which transforms the code into an executable binary, and the runtime, which is piece of software that performs functions like the garbage collector, the scheduler, goroutines, maps, channels, etc.

Golang runtime is quite lean on compute resource usage. This is perfect on its own, as it allows the majority of developers to stay concentrated on coding tasks. But when the Golang application is deployed as a containerized service, It needs to share available compute resources with many other containers. Go binaries are not inherently container-aware; specifically, they do not account for memory and CPU limits. In the high-load scenario, When the Golang application needs more memory, the application would request the memory from the Operating System. When the Operating system (Linux) runs out of the available physical memory, the OOM Killer will intervene and kill the process that is taking more memory. To prevent this from happening, Go provided the SetMemoryLimit option in Golang 1.19. This helps in making the application aware of the maximum memory limit. When the application reaches its memory limit, the Go Runtime will trigger the Garbage collector to sweep up the unused memory so the application can start reusing it instead of requesting the Operating system for more memory. This Paper describes the usage and the performance impact of setting this memory limit.

## 1. Introduction

Go is a statically typed, concurrent, and garbage-collected programming language created at Google in 2009. It is designed to be simple, efficient, and easy to learn, making it a popular choice for building scalable network services, web applications, and command-line tools.

Go is known for its support for concurrency, which is the ability to run multiple tasks simultaneously. Concurrency is achieved in Go through the use of Goroutines and Channels, which allow you to write code that can run multiple operations at the same time. This makes Go an ideal choice for building high-performance and scalable network services, as well as for solving complex computational problems.

Another important feature of Go is its garbage collection, which automatically manages memory for you. This eliminates the need for manual memory management, reducing the likelihood of memory leaks and other bugs that can arise from manual memory management.

## 2. Problem Statement

When the golang application gets more requests from clients, It uses the allocated application memory and then starts requesting more and more memory from the Operating System. Initially, the Operating System would allocate memory to the application. But when the host memory is close to exhaustion, the Linux operating system would respond with an Out Of Memory (OOM) exception and kill the application so that other

applications on the system could run seamlessly. I have seen this situation happening in production, and it has a significant impact on the users as the application needs to start again.

## 3. Memory Management and Garbage Collection

Any application contains variables, objects, and instructions to be executed on the variables and objects. These are usually stored in two main memory stores: the stack and the heap. Typically, the stack stores data whose size and usage time can be predicted by the Go compiler. This includes local function variables, function arguments, return values, etc.

The stack is managed automatically and follows the Last-In-First-Out (LIFO) principle. When a function is called, all associated data is placed on top of the stack, and when the function finishes, this data is removed from the stack. The stack does not require a complex garbage collection mechanism and incurs minimal overhead for memory management. Retrieving and storing data in the stack happens very quickly.

Data that changes dynamically during execution or requires access beyond the scope of a function cannot be placed on the stack because the compiler cannot predict its usage. Such data is stored on the heap. Retrieving data from the heap and managing it is a costly process. 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. This mechanism of reusing memory is called Garbage Collection. The term garbage essentially means unused or objects that are created in memory and are no longer needed. These are nothing more than garbage. Memory is a costly space and must be cleaned periodically to make space for other programs to execute (or for the same program to work efficiently). Hence, we need garbage collection to wipe out that memory. If this process of Garbage collection is done automatically, without any manual intervention, it is called Automatic garbage collection. The Garbage Collector is a system designed specifically to identify and free dynamically allocated memory. Automatic garbage collection is always an overhead over the efficiency of the executing program.

## 4. Garbage Collection in GO

The garbage collection is expensive as it consumes two important system resources: CPU time and physical memory. The memory in the garbage collector consists of the following:

- Live heap memory (memory marked as “live” in the previous garbage collection cycle)
- New heap memory (heap memory not yet analyzed by the garbage collector)
- Memory is used to store some metadata, which is usually insignificant compared to the first two entities.

Go uses a garbage collection algorithm based on tracing and the Mark and Sweep algorithm. 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. As a result, the act of sweeping must be entirely separated from the act 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.

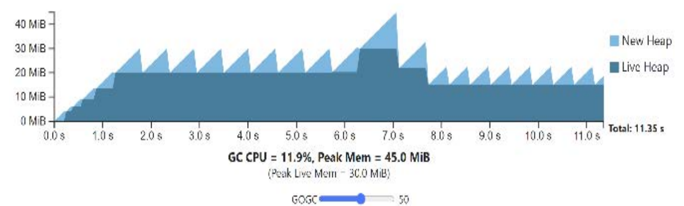
### 4.1. GOGC

GOGC is one of the oldest environment variables supported by the Go runtime. At a high level, GOGC determines the trade-off between GC CPU and memory. It controls the aggressiveness of the garbage collector. By default, this value is assumed to be 100, which means garbage collection will not be triggered until the heap has grown by 100% since the previous collection. Effectively, GOGC=100 (the default) means the garbage collector will run each time the live heap doubles.

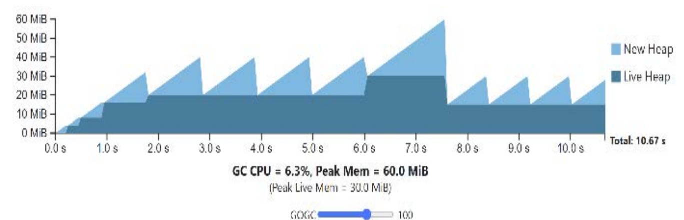
Setting this value higher, say GOGC=200, will delay the start of a garbage collection cycle until the live heap has grown to 200% of the previous size. Setting the value lower, say GOGC=20 will cause the garbage collector to be triggered more often as less new data can be allocated on the heap before triggering a collection. The key takeaway is that doubling GOGC will double heap memory overheads and roughly halve GC CPU cost Setting GOGC=off will disable garbage collection entirely.

The below graph depicts the execution of some programs whose non-GC work takes 10 seconds of CPU time to complete. In the first second, it performs some initialization steps (growing its live heap) before settling into a steady state. The application allocates runs in a container that has little over 60 MB available. At steady state, the application uses 20 MB live memory, but in transient spike, the application can use up to 40 MB of peak live memory. It assumes that the only relevant GC work to complete comes from the live heap and that the application uses no additional memory.

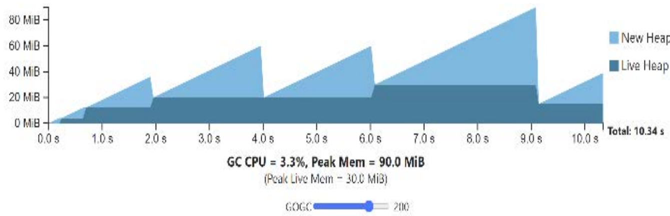
Each GC cycle ends while the new heap drops to zero. The time taken while the new heap drops to zero is the combined time for the mark phase for cycle N and the sweep phase for cycle N+1. Note that this visualization (and all the visualizations in this guide) assume the application is paused while the GC executes, so GC CPU costs are fully represented by the time it takes for new heap memory to drop to zero. This is only to make visualization simpler; the same intuition still applies. The X axis shows the full CPU-time duration of the program, and Y axis denotes the memory. Notice that additional CPU time used by the GC increases the overall duration. As GOGC increases, CPU overhead decreases, but peak memory increases proportionally to the live heap size. As GOGC decreases, the peak memory requirement decreases at the expense of additional CPU overhead. below graphs represent application behavior when GOGC is set to different values




GoGC = 50, GC CPU = 11.9%, Peak Memory = 45.0 MiB, Total CPU time = 11.35s




GoGC = 100, GC CPU = 6.3%, Peak Mem = 60.0 MiB, Total: 10.67s




GoGC = 200, GC CPU = 3.3%, Peak Mem = 90.4 MiB, Total: 10.34 s

Consider a situation where the application memory is close to the total memory. In that scenario, If the application requests more memory and the OS doesn't have that much memory to allocate then, the OS would respond with an OOM Exception and kill the application.

### 4.2. Gomemlimit

Until Go 1.19, GOGC was the sole parameter that could be used to modify the GC's behavior. But this doesn't take into account that available memory is finite. Consider a scenario where there's a transient spike in the live heap size. Because the GC will pick a total heap size proportional to that live heap size, the GOGC must be configured to match the peak live heap size, even if, in the usual case, a higher GOGC value provides a better trade-off.

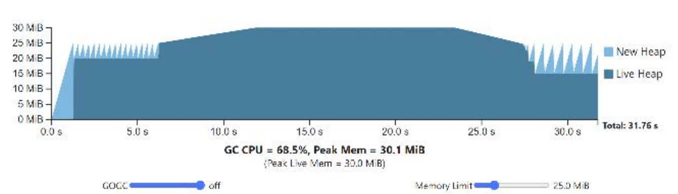
Consider an application running in a container with a bit over 60 MB of memory available; then the application cannot go beyond 60 MB. In some applications, these transient peaks can be rare and hard to predict, leading to occasional, unavoidable, and potentially costly out-of-memory conditions. So, in the 1.19 release, Go added support for setting a runtime memory limit. The memory limit may be configured either via the GOMEMLIMIT environment variable, which all Go programs recognize, or through the SetMemoryLimit function available in the runtime/debug package.

This memory limit sets a maximum on the total amount of memory that the Go runtime can use. The specific set of memory included is defined in terms of runtime. MemStats as the expression.

Notice that peak memory use stops at whatever the memory limit is, but the rest of the program's execution still obeys the total heap size rule set by GOGC. So, Even when GOGC is turned off, the memory limit is still respected. In fact, this particular configuration represents a maximization of resource economy because it sets the minimum GC frequency required to maintain some memory limit. In this case, all of the program's execution has the heap size rise to meet the memory limit. The use of a memory limit does come with a cost and certainly doesn't invalidate the utility of GOGC.

Consider what happens when the live heap grows large enough to bring total memory use close to the memory limit. With GOGC off and lower memory limit, we notice that the total time the application takes will start to grow in an unbounded manner as the GC is constantly executing to maintain an impossible memory limit.

This situation, where the program fails to make reasonable progress due to constant GC cycles, is called thrashing. It's particularly dangerous because it effectively stalls the program. In many cases, an indefinite stall is worse than an out-of-memory condition, which tends to result in a much faster failure.




GOGC = off, GC CPU = 68.5%, Peak Memory = 30.1 MiB, Total CPU time = 31.76s, Memory Limit = 25 MB

For this reason, the memory limit is defined as soft. The Go runtime makes no guarantees that it will maintain this memory limit under all circumstances; it only promises some reasonable amount of effort. This relaxation of the memory limit is critical to avoiding thrashing behavior because it gives the GC a way out: let memory use surpass the limit to avoid spending too much time in the GC.

How this works internally is that the GC sets an upper limit on the amount of CPU time it can use over some time window. This limit is currently set at roughly 50%, with a 2 * GOMAXPROCS CPU-second window. The consequence of limiting GC CPU time is that the GC's work is delayed, while the Go program may continue allocating new heap memory, even beyond the memory limit.

The intuition behind the 50% GC CPU limit is based on the worst-case impact on a program with ample available memory. In the case of a misconfiguration of the memory limit, where it is set too low mistakenly, the program will slow down at most by 2x because the GC can't take more than 50% of its CPU time away.

## 5. Set the Memory Limit for a Golang Application While Running the Docker Container

For testing purposes, I am using an 8-GB Linux-based VM. I have written a sample Golang program where I initialize a sample structure of data in a loop and print it on the console. The main purpose of this program is to take a huge heap size so that out-of-memory situation can be created. With the values mentioned in the program, I am able to see out-of-memory issue.

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 <= 7000; i++ {
		for j := 1; j <= 7000; 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:47:6: can inline bToMb
./main.go:41:36: inlining call to bToMb
./main.go:41:12: inlining call to fmt.Printf
./main.go:42:43: inlining call to bToMb
./main.go:42:12: inlining call to fmt.Printf
./main.go:43:36: inlining call to bToMb
./main.go:43:12: inlining call to fmt.Printf
./main.go:44:12: inlining call to fmt.Printf
./main.go:25:35: inlining call to strconv.Itoa
./main.go:25:53: inlining call to strconv.Itoa
./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:42:12: ... argument does not escape
./main.go:42:43: ~r0 escapes to heap
./main.go:43:12: ... argument does not escape
./main.go:43:36: ~r0 escapes to heap
./main.go:44:12: ... argument does not escape
./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 escapes 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.

## 6. 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:1.22
WORKDIR /src
COPY main.go .
RUN go build -o /test ./main.go
CMD ["/test"]
```

Command to Build Docker Image:

```
docker build -t setmemlimit .
```

## 7. Start the Container without Gomemlimit

Let's start the container without gomemlimit 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 without gomemlimit.

```
docker run --log-driver=journald -d setmemlimit:latest
```

This would start a new container without any memorylimit. The Go Application would run for some time, and later Linux OS would be unable to allocate more memory, thus killing the process. The below log is observed in the journal.

Application kernel: Out of memory: Killed process 570771 (test) total-vm:22604780kB, anon-rss:7505364kB, file-rss:0kB, shmem-rss:0kB, UID:0 pgtables:39212kB oom_score_adj:0

Also, the exit status of the Docker container is 137, indicating that container was immediately terminated by the operating system via SIGKILL signal. Below is the docker inspect command output.

```
"State": {
	"Status": "exited",
	"Running": false,
	"Paused": false,
	"Restarting": false,
	"OOMKilled": true,
	"Dead": false,
	"Pid": 0,
	"ExitCode": 137,
```

"Error": "",
},

## 8. Start the Container with Gomemlimit

Command to start the container with gomemlimit.

dockerrun-e"GOMEMLIMIT=6000MiB"--log-driver=journald -d setmemlimit:latest

This would start a new container with a memory limit specified as 6 MB. As soon as the Go Application memory usage reaches around 6 MB, the garbage collector will run and try to free the heap memory. So, the application didn't crash and exited with 0 status.

```
docker inspect 75cdf3703e46
    "State": {
        "Status": "exited",
        "Running": false,
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
        "Dead": false,
        "Pid": 0,
        "ExitCode": 0,
        "Error": "",
    },
```

So, setting the memory limit prevented the application from being terminated with out-of-memory issues.

## 9. Performance Evaluation

Let's check the Application performance under normal load without and with the Memory Limit and with default GOGC

Consider the same go file which I used earlier but this time taking loop value as 5k instead of 7k so that the heap usage of the application is reduced.

Without GOMEMLIMIT:

docker run --log-driver=journald -d lowmemory:latest

application 845d28b610b0[1034]: Alloc = 6789 MiB TotalAlloc = 27189 MiB Sys = 13318 MiB NumGC = 31

application systemd [1]: docker-845d28b610b04efbe91043d3dcdc44ab88bf2138c88d809062232435ce278167.scope: Consumed 1 minute, 25.753 seconds of CPU time.

After performing the same test multiple times, we could see the CPU time is always between 1 minute and 2 minutes.

With GOMEMLIMIT:

docker run -e "GOMEMLIMIT=6000MiB" --log-driver=journald -d lowmemory:latest

application 6c215ea93fb5 [1034]: Alloc = 6773 MiB TotalAlloc = 27189 MiB Sys = 15606 MiB; NumGC = 344.

application systemd[1]: docker-6c215ea93fb59447813c99ff4d951ba752c2c00bae608f4f3cf15106d01d69fc.scope: Consumed 6 minutes 24.129 seconds of CPU time.

After performing the same test multiple times, we could see the CPU time is always between 5 and 7 minutes and never below that. This is due to the garbage collector running multiple times. Also, we can see that the number of times the garbage collector ran was 344, compared to 31 without the memory limit.

So, setting the Memory Limit for a Go Application does have a performance impact. The below table summarizes the GC cycles and execution time with different memory limits. The application is run with the same memory limit at least three times, and the memory values and CPU execution times are noted. Less CPU time is observed without the memory limit, and with the Memory Limit of 6.5 GB, the application has less execution time compared to memory limits of 5.5 GB, 6 GB, and 7 GB.

| | Alloc | Total Alloc | Sys Mem | CPU time | No of times GC ran |
|---|---|---|---|---|---|
| Without Memory Limit | 6773 | 27189 | 13318 | 1min 25.753s | 31 |
| | 6773 | 27189 | 15299 | 1min 806ms | 31 |
| | 6773 | 27189 | 15254 | 1min 29.454s | 31 |
| | 6773 | 27189 | 14472 | 1min 49.487s | 32 |
| | | | | | |
| With Memory Limit 7 GB | 6773 | 27189 | 13947 | 6min 12.650s | 334 |
| | 6773 | 27189 | 13777 | 5min 40.280s | 267 |
| | 6773 | 27189 | 13999 | 6min 29.249s | 349 |
| | | | | | |
| With Memory Limit 6.5 GB | 6773 | 27189 | 13089 | 5min 32.510s | 293 |
| | 6773 | 27189 | 13987 | 4min 55.255s | 247 |
| | 6773 | 27189 | 14087 | 5min 44.714s | 265 |
| | | | | | |
| With Memory Limit 6 GB | 6773 | 27189 | 11490 | 7min 52.607s | 335 |
| | 6773 | 27189 | 13642 | 8min 1.108s | 413 |
| | 6773 | 27189 | 15606 | 7min 24.129s | 324 |
| | | | | | |
| With Memory Limit 5.5 GB | 6773 | 27189 | 13712 | 8min 10.115s | 451 |
| | 6773 | 27189 | 15317 | 8min 58.252s | 462 |
| | 6773 | 27189 | 15317 | 8min 15.240s | 453 |

10. Conclusion

Below are some of the recommendations on setting the memory limit.

1. Set the memory limit only when the execution environment of your Go program is entirely within your control and the Go program is the only program with access to some set of resources (i.e., some kind of memory reservation, like a container memory limit).

Example: Deployment of a web service into containers with a fixed amount of available memory. In this case, a good rule of thumb is to leave an additional 5–10% of headroom to account for memory sources the Go runtime is unaware of.

2. Adjust the memory limit in real time to adapt to changing conditions.

Example: cgo program, where C libraries temporarily need to use substantially more memory.

3. Don't set GOGC to off with a memory limit if the Go program is sharing some of its limited memory with other programs, and those programs are generally decoupled from the Go program. Instead, keep the memory limit as it helps to curb undesirable transient behavior, but set GOGC to some smaller, reasonable value for the average case.

Example: If the Go program calls some subprocess and blocks while its callee executes, the result will be less reliable, as inevitably both programs will need more memory. Letting

the Go program use less memory when it doesn't need it will generate a more reliable result overall. This advice also applies to overcommit situations, where the sum of memory limits of containers running on one machine may exceed the actual physical memory available to the machine.

1. Don't use the memory limit when deploying to an execution environment you don't control, especially when your program's memory use is proportional to its inputs.

**Example:** CLI tool or a desktop application. Baking a memory limit into the program when it's unclear what kind of inputs it might be fed or how much memory might be available on the system can lead to confusing crashes and poor performance. Plus, an advanced end-user can always set a memory limit if they wish.

2. Don't set a memory limit to avoid out-of-memory conditions when a program is already close to its environment's memory limits.

This replaces an out-of-memory risk with a risk of severe application slowdown, which is often not preferable, even with the efforts Go makes to mitigate thrashing. In such a case, it would be much more effective to either increase the environment's memory limits (and then potentially set a memory limit) or decrease GOGC (which provides a much cleaner trade-off than thrashing-mitigation does).

## 11. References

1. A Guide to the Go Garbage collector.
2. Fenyuk Y. Golang. Shaping web-service memory consumption. Medium 2024.
3. Cheney D. A whirlwind tour of Go's runtime environment variables. Dave.
4. https://www.geeksforgeeks.org/go-programming-language-introduction/
5. Debnath M. Understanding garbage colledction in GO. Developer.com 2022.