**Understanding Allocations in Go**

Anyone who’s run a benchmark with the -benchmem flag will have seen the allocs/op stat in output like the below. In this post we’ll look at what counts as an alloc and what we can do to influence this number.

BenchmarkFunc-8 67836464 16.0 ns/op 8 B/op 1 allocs/op

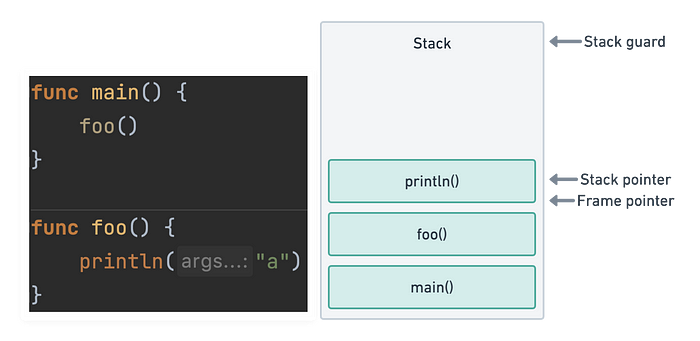
# The stack and heap we know and love

To discuss the allocs/op stat in Go, we’re going to be interested in two areas of memory in our Go programs: the stack and the heap.

In many popular programming environments the stack usually refers to the call stack of a thread. A call stack is a LIFO stack data structure that stores arguments, local variables, and other data tracked as a thread executes functions. Each function call adds (pushes) a new frame to the stack, and each returning function removes (pops) from the stack.

We must be able to safely free the memory of the most recent stack frame when it’s popped. We therefore can’t store anything on the stack that later needs to be referenced elsewhere.

Since threads are managed by the OS, the amount of memory available to a thread stack is typically fixed, e.g. a default of 8MB in many Linux environments. This means we also need to be mindful of how much data ends up on the stack, particularly in the case of deeply-nested recursive functions. If the stack pointer in the diagram above passes the stack guard, the program will crash with a stack overflow error.



The heap is a more complex area of memory that has no relation to the data structure of the same name.

We can use the heap on demand to store data needed in our program.

Memory allocated here can’t simply be freed when a function returns, and needs to be carefully managed to avoid leaks and fragmentation.

The heap will generally grow many times larger than any thread stack, and the bulk of any optimization efforts will be spent investigating heap use.

# The Go stack and heap

# Threads managed by the OS are completely abstracted away from us by the Go runtime, and we instead work with a new abstraction: goroutines.

# Goroutines are conceptually very similar to threads, but they exist within user space.

# This means the runtime, and not the OS, sets the rules of how stacks behave.

# 

# Rather than having hard limits set by the OS, goroutine stacks start with a small amount of memory (currently 2KB).

# Before each function call is executed, a check within the function prologue is executed to verify that a stack overflow won’t occur.

# In the below diagram, the convert() function can be executed within the limits of the current stack size (without SP overshooting stackguard0).

# If this wasn’t the case, the runtime would copy the current stack to a new larger space of contiguous memory before executing convert().

# This means that stacks in Go are dynamically sized, and can typically keep growing as long as there’s enough memory available to feed them.

# The Go heap is again conceptually similar to the threaded model described above.

# All goroutines share a common heap and anything that can’t be stored on the stack will end up there.

# When a heap allocation occurs in a function being benchmarked, we’ll see the allocs/ops stat go up by one.

# It’s the job of the garbage collector to later free heap variables that are no longer referenced.

*Go compilers will allocate variables that are local to a function in that function’s stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.*

*If a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack.*

# Since compiler implementations change over time, ****there’s no way of knowing which variables will be allocated to the heap simply by reading Go code****.

# It is, however, possible to view the results of the escape analysis mentioned above in output from the compiler.

# This can be achieved with the gcflags argument passed to go build. A full list of options can be viewed via go tool compile -help.

For escape analysis results, the -m option (print optimization decisions) can be used. Let’s test this with a simple program that creates two stack frames for functions main1 and stackIt.

func main1() {  
 \_ = stackIt()  
}  
//go:noinline  
func stackIt() int {  
 y := 2  
 return y \* 2  
}

Since we can can’t discuss stack behaviour if the compiler removes our function calls, the noinline [pragma](https://dave.cheney.net/2018/01/08/gos-hidden-pragmas) is used to prevent inlining when compiling the code.

Let’s take a look at what the compiler has to say about its optimization decisions.

The -l option is used to omit inlining decisions.

$ go build -gcflags '-m -l'  
# github.com/Jimeux/go-samples/allocations

# Here we see that no decisions were made regarding escape analysis. In other words, variable y remained on the stack, and didn’t trigger any heap allocations. We can verify this with a benchmark.

# With less Slice size=1000 it is allocated on stack only

package main

func main() {

    //result := calci.Sum(12, 10)

    //fmt.Println("Sum = ", result)

    \_ = stackIt()

}

//go:noinline

func stackIt1() int {

    y := 2

    s := make([]int, 0, 1000)

    s = append(s, y)

    return y \* 2

}

//OUTPUT

PS D:\Golang\practice> go build -gcflags '-m -l'

# practice.com

.\main.go:12:11: make([]int, 0, 1000) does not escape

# Here, with more slice=1000000 size it got allocated in heap

package main

func main() {

    //result := calci.Sum(12, 10)

    //fmt.Println("Sum = ", result)

    \_ = stackIt()

}

//go:noinline

func stackIt1() int {

    y := 2

    s := make([]int, 0, 1000000)

    s = append(s, y)

    return y \* 2

}

//OUTPUT

PS D:\Golang\practice> go build -gcflags '-m -l'

# practice.com

.\main.go:12:11: make([]int, 0, 1000000) escapes to heap

$ go test -bench . -benchmem  
BenchmarkStackIt-8 680439016 1.52 ns/op 0 B/op **0 allocs/op**

As expected, the allocs/op stat is 0. An important observation we can make from this result is that **copying variables can allow us to keep them on the stack** and avoid allocation to the heap. Let’s verify this by modifying the program to avoid copying with use of a pointer.

package main

func main() {

    //result := calci.Sum(12, 10)

    //fmt.Println("Sum = ", result)

    \_ = stackIt2()

}

//go:noinline

func stackIt2() \*int {

    y := 2

    res := y \* 2

    return &res

}

//OUTPUT

PS D:\Golang\practice> go build -gcflags '-m -l'

# practice.com

.\main.go:12:2: moved to heap: res

# The compiler tells us it moved the pointer res to the heap, which triggers a heap allocation as verified in the benchmark below

$ go test -bench . -benchmem  
BenchmarkStackIt2-8 70922517 16.0 ns/op 8 B/op **1 allocs/**

# So does this mean pointers are guaranteed to create allocations?

# Let’s modify the program again to this time pass a pointer down the stack.

package main

func main() {

    //result := calci.Sum(12, 10)

    //fmt.Println("Sum = ", result)

    y := 10

    \_ = stackIt3(&y)

}

//go:noinline

func stackIt3(y \*int) int {

    res := \*y \* 2

    return res

}

//OUTPUT

**PS D:\Golang\practice> go build -gcflags '-m -l'**

**# practice.com**

**.\main.go:11:14: y does not escape**

Yet running the benchmark shows nothing was allocated to the heap.

$ go test -bench . -benchmem  
BenchmarkStackIt3-8 705347884 1.62 ns/op 0 B/op **0 allocs/op**

# Why do we get this seeming inconsistency?

# stackIt2 passes the address of y up the stack to main, where y will be referenced after the stack frame of stackIt2 has already been freed. The compiler is therefore able to judge that y must be moved to the heap to remain alive. If it doesn’t do this, we’ll get a nil pointer in main when attempted to reference y.

# stackIt3, on the other hand, passes y down the stack, and y isn’t referenced anywhere outside main3. The compiler is therefore able to judge that y can exist within the stack alone, and doesn’t need to be allocated to the heap. We won’t be able to produce a nil pointer in any circumstances by referencing y.

# ****A general rule we can infer from this is that sharing pointers up the stack results in allocations, whereas sharing points down the stack doesn’t.**** However, this is not guaranteed, so you’ll still need to verify with gcflags or benchmarks to be sure. What we can say for sure is that any attempt to reduce allocs/op will involve hunting out wayward pointers.

package main

type BigStruct struct {

    A, B, C int

    D, E, F string

    G, H, I bool

}

func main() {

}

//go:noinline

func CreateCopy() BigStruct {

    return BigStruct{

        A: 123, B: 456, C: 789,

        D: "ABC", E: "DEF", F: "HIJ",

        G: true, H: true, I: true,

    }

}

//go:noinline

func CreatePointer() \*BigStruct {

    return &BigStruct{

        A: 123, B: 456, C: 789,

        D: "ABC", E: "DEF", F: "HIJ",

        G: true, H: true, I: true,

    }

}

//OUTPUT

PS D:\Golang\practice> go build -gcflags '-m -l'

# practice.com

.\main.go:23:9: &BigStruct{...} escapes to heap

PS D:\Golang\practice> go test -bench . -benchmem

goos: windows

goarch: amd64

pkg: practice.com

cpu: Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz

BenchmarkTestCopyIt-6           1000000000               0.08272 ns/op         0 B/op          0 allocs/op

BenchmarkTestPointerIt-6               1        1018472600 ns/op        1600003280 B/op 20000036 allocs/op

PASS

ok      practice.com    3.852s

PS D:\Golang\practice>

Next we’ll save the trace output for CreateCopy to file copy\_trace.out, and open it with the trace tool in the browser.

$ go test -run TestCopyIt -trace=copy\_trace.out  
PASS  
ok github.com/Jimeux/go-samples/allocations 0.281s$ go tool trace copy\_trace.out  
Parsing trace...  
Splitting trace...  
Opening browser. Trace viewer is listening on http://127.0.0.1:57530

Choosing View trace from the menu shows us the below, which is almost as unremarkable as our flame chart for the stackIt function. Only two of eight potential logical cores (Procs) are utilised, and goroutine G19 spends just about the entire time running our test loop — which is what we want.

# 

# Let’s generate the trace data for the CreatePointer code.

$ go test -run TestPointerIt -trace=pointer\_trace.out  
PASS  
ok github.com/Jimeux/go-samples/allocations 2.224sgo tool trace pointer\_trace.out  
Parsing trace...  
Splitting trace...  
Opening browser. Trace viewer is listening on [http://127.0.0.1:57784](http://127.0.0.1:57784/)

# You may have already noticed the test took 2.224s compared to 0.281s for CreateCopy, and selecting View trace displays something much more colourful and busy this time. All logical cores were utilised and there appears to be a lot more heap action, threads, and goroutines than last time.

# A screenshot of a computer Description automatically generated

# If we zoom in to a millisecond or so span of the trace, we see many goroutines performing operations related to [garbage collection](https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html). The quote earlier from the FAQ used the phrase garbage-collected heap because it’s the job of the garbage collector to clean up anything on the heap that is no longer being referenced.

# 

# Although Go’s garbage collector is increasingly efficient, the process doesn’t come for free. We can verify visually that the test code stopped completely at times in the above trace output. This wasn’t the case for CreateCopy, since all of our BigStruct instances remained on the stack, and the GC had very little to do.

# Comparing the goroutine analysis from the two sets of trace data offers more insight into this.

# CreatePointer (bottom) spent over 15% of its execution time sweeping or pausing (GC) and scheduling goroutines.

# Top-level goroutine analysis for CreateCopy

# 

# Top-level goroutine analysis for CreatePointer

# 

A look at some of the stats available elsewhere in the trace data further illustrates the cost of heap allocation, with a stark difference in the number of goroutines generated, and almost 400 STW (stop the world) events for the CreatePointer test.

+------------+------+---------+  
| | Copy | Pointer |  
+------------+------+---------+  
| Goroutines | 41 | 406965 |  
| Heap | 10 | 197549 |  
| Threads | 15 | 12943 |  
| bgsweep | 0 | 193094 |  
| STW | 0 | 397 |  
+------------+------+---------+