## Profiling Code

We can use the go tooling to inspect and profile our programs. Profiling is more of a journey and detective work. It requires some understanding about the application and expectations. The profiling data in and of itself is just raw numbers. We have to give it meaning and understanding.

## The Basics of Profiling

"Those who can make you believe absurdities can make you commit atrocities" - Voltaire

### How does a profiler work?

A profiler runs your program and configures the operating system to interrupt it at regular intervals. This is done by sending SIGPROF to the program being profiled, which suspends and transfers execution to the profiler. The profiler then grabs the program counter for each executing thread and restarts the program.

### Profiling do's and don't's

Before you profile, you must have a stable environment to get repeatable results.

* The machine must be idle—don't profile on shared hardware, don't browse the web while waiting for a long benchmark to run.
* Watch out for power saving and thermal scaling.
* Avoid virtual machines and shared cloud hosting; they are too noisy for consistent measurements.

If you can afford it, buy dedicated performance test hardware. Rack them, disable all the power management and thermal scaling and never update the software on those machines.

For everyone else, have a before and after sample and run them multiple times to get consistent results.

### Types of Profiling

**CPU profiling**  
CPU profiling is the most common type of profile. When CPU profiling is enabled, the runtime will interrupt itself every 10ms and record the stack trace of the currently running goroutines. Once the profile is saved to disk, we can analyse it to determine the hottest code paths. The more times a function appears in the profile, the more time that code path is taking as a percentage of the total runtime.

**Memory profiling**  
Memory profiling records the stack trace when a heap allocation is made. Memory profiling, like CPU profiling is sample based. By default memory profiling samples 1 in every 1000 allocations. This rate can be changed. Stack allocations are assumed to be free and are not tracked in the memory profile. Because of memory profiling is sample based and because it tracks allocations not use, using memory profiling to determine your application's overall memory usage is difficult.

**Block profiling**  
Block profiling is quite unique. A block profile is similar to a CPU profile, but it records the amount of time a goroutine spent waiting for a shared resource. This can be useful for determining concurrency bottlenecks in your application. Block profiling can show you when a large number of goroutines could make progress, but were blocked.

Blocking includes:

* Sending or receiving on an unbuffered channel.
* Sending to a full channel, receiving from an empty one.
* Trying to Lock a sync.Mutex that is locked by another goroutine.
* Block profiling is a very specialised tool, it should not be used until you believe you have eliminated all your CPU and memory usage bottlenecks.

**One profile at at time**  
Profiling is not free. Profiling has a moderate, but measurable impact on program performance—especially if you increase the memory profile sample rate. Most tools will not stop you from enabling multiple profiles at once. If you enable multiple profiles at the same time, they will observe their own interactions and skew your results.

**Do not enable more than one kind of profile at a time.**

### Hints to interpret what you see in the profile

If you see lots of time spent in runtime.mallocgc function, the program potentially makes excessive amount of small memory allocations. The profile will tell you where the allocations are coming from. See the memory profiler section for suggestions on how to optimize this case.

If lots of time is spent in channel operations, sync.Mutex code and other synchronization primitives or System component, the program probably suffers from contention. Consider to restructure program to eliminate frequently accessed shared resources. Common techniques for this include sharding/partitioning, local buffering/batching and copy-on-write technique.

If lots of time is spent in syscall.Read/Write, the program potentially makes excessive amount of small reads and writes. Bufio wrappers around os.File or net.Conn can help in this case.

If lots of time is spent in GC component, the program either allocates too many transient objects or heap size is very small so garbage collections happen too frequently.

* Large objects affect memory consumption and GC time, while large number of tiny allocations affects execution speed.
* Combine values into larger values. This will reduce number of memory allocations (faster) and also reduce pressure on garbage collector (faster garbage collections).
* Values that do not contain any pointers are not scanned by garbage collector. Removing pointers from actively used value can positively impact garbage collection time.

## Rules of Performance

1. Never guess about performance.
2. Measurements must be relevant.
3. Profile before you decide something is performance critical.
4. Test to know you are correct.

## Installing Tools

**hey**  
hey is a modern HTTP benchmarking tool capable of generating the load you need to run tests. It's built using the Go language and leverages goroutines for behind the scenes async IO and concurrency.

go get -u github.com/rakyll/hey

## Dave Cheney's Profiling Presentation:

Much of what I have learned comes from Dave and working on solving problems. This slide deck is a great place to start. Much of this material can be found in the material below.

[Seven ways to profile a Go program](http://go-talks.appspot.com/github.com/davecheney/presentations/seven.slide#1)

## Profiling, Debugging and Optimization Reading

Here is more reading and videos to also help get you started.

[The past and future of Microprocessor performance](https://github.com/davecheney/gophercon2018-performance-tuning-workshop/blob/master/1-welcome/introduction.md) - Dave Cheney

[Language Mechanics On Escape Analysis](https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-escape-analysis.html) - William Kennedy  
[Profiling Go Programs](http://golang.org/blog/profiling-go-programs) - Go Team  
[Profiling & Optimizing in Go](https://www.youtube.com/watch?v=xxDZuPEgbBU) - Brad Fitzpatrick  
[Go Dynamic Tools](https://www.youtube.com/watch?v=a9xrxRsIbSU) - Dmitry Vyukov  
[How NOT to Measure Latency](https://www.youtube.com/watch?v=lJ8ydIuPFeU&feature=youtu.be) - Gil Tene  
[Go Performance Tales](http://jmoiron.net/blog/go-performance-tales) - Jason Moiron  
[Debugging performance issues in Go programs](https://software.intel.com/en-us/blogs/2014/05/10/debugging-performance-issues-in-go-programs) - Dmitry Vyukov  
[Reduce allocation in Go code](https://methane.github.io/2015/02/reduce-allocation-in-go-code) - Python Bytes  
[Write High Performance Go](http://go-talks.appspot.com/github.com/davecheney/presentations/writing-high-performance-go.slide) - Dave Cheney  
[Static analysis features of godoc](https://golang.org/lib/godoc/analysis/help.html) - Go Team  
[Seven ways to profile a Go program](https://www.bigmarker.com/remote-meetup-go/Seven-ways-to-profile-a-Go-program) - Dave Cheney  
[runtime: goroutine execution stalled during GC](https://github.com/golang/go/issues/16293) - Caleb Spare  
[Go's execution tracer](http://www.thedotpost.com/2016/10/rhys-hiltner-go-execution-tracer) - Rhys Hiltner  
[Using Instruments to profile Go programs](https://rakyll.org/instruments) - JBD  
[Fighting latency: the CPU profiler is not your ally](https://www.youtube.com/watch?v=nsM_m4hZ-bA&t=973s) - Filippo Valsorda  
[go tool trace](https://making.pusher.com/go-tool-trace/) - Will Sewell

## Go and OS Tooling

### time

The **time** command provide information that can help you get a sense how your program is performing.

Use the **time** command to see data about building the program. $ cd $GOPATH/src/github.com/ardanlabs/gotraining/topics/go/profiling/project $ /usr/bin/time -lp go build -- Mac OS X $ /usr/bin/time -v go build -- Linux

### perf

If you're a linux user, then perf(1) is a great tool for profiling applications. Now we have frame pointers, perf can profile Go applications.

$ go build -toolexec="perf stat" cmd/compile/internal/gc

# cmd/compile/internal/gc

Performance counter stats for '/home/dfc/go/pkg/tool/linux\_amd64/compile -o $WORK/cmd/compile/internal/gc.a -trimpath $WORK -p cmd/compile/internal/gc -complete -buildid 87cd803267511b4d9e753d68b5b66a70e2f878c4 -D \_/home/dfc/go/src/cmd/compile/internal/gc -I $WORK -pack ./alg.go ./align.go ./bexport.go ./bimport.go ./builtin.go ./bv.go ./cgen.go ./closure.go ./const.go ./cplx.go ./dcl.go ./esc.go ./export.go ./fmt.go ./gen.go ./go.go ./gsubr.go ./init.go ./inl.go ./lex.go ./magic.go ./main.go ./mpfloat.go ./mpint.go ./obj.go ./opnames.go ./order.go ./parser.go ./pgen.go ./plive.go ./popt.go ./racewalk.go ./range.go ./reflect.go ./reg.go ./select.go ./sinit.go ./sparselocatephifunctions.go ./ssa.go ./subr.go ./swt.go ./syntax.go ./type.go ./typecheck.go ./universe.go ./unsafe.go ./util.go ./walk.go':

7026.140760 task-clock (msec) # 1.283 CPUs utilized

1,665 context-switches # 0.237 K/sec

39 cpu-migrations # 0.006 K/sec

77,362 page-faults # 0.011 M/sec

21,769,537,949 cycles # 3.098 GHz [83.41%]

11,671,235,864 stalled-cycles-frontend # 53.61% frontend cycles idle [83.31%]

6,839,727,058 stalled-cycles-backend # 31.42% backend cycles idle [66.65%]

27,157,950,447 instructions # 1.25 insns per cycle

# 0.43 stalled cycles per insn [83.25%]

5,351,057,260 branches # 761.593 M/sec [83.49%]

118,150,150 branch-misses # 2.21% of all branches [83.15%]

5.476816754 seconds time elapsed

**Stack Traces and Core Dumps**

Having some basic skills in debugging Go programs can save any programmer a good amount of time trying to identify problems. I believe in logging as much information as you can, but sometimes a panic occurs and what you logged is not enough. Understanding the information in a stack trace can sometimes mean the difference between finding the bug now or needing to add more logging and waiting for it to happen again. We can also stop a running program and get Core Dump which also generates a stack trace.

**Notes**

* Stack traces are an important tool in debugging an application.
* The runtime should never panic so the trace is everything.
* You can see every goroutine and the call stack for each routine.
* You can see every value passed into each function on the stack.
* You can generate core dumps and use these same techniques.

**Stack Traces**

These two programs called the built-in function panic to produce a stack trace. Stack traces show not only the call stack from the line of code that caused the panic. They also show the values that were passed into each function.

**Example 1**

Build and run the program.

$ go build

$ ./example1

Review the stack trace.

// Stack Trace

goroutine 1 [running]:

main.example(0xc000042748, 0x2, 0x4, 0x106abae, 0x5, 0xa)

stack\_trace/example1/example1.go:13 +0x39

main.main()

stack\_trace/example1/example1.go:8 +0x72

// Declaration

main.example(slice []string, str string, i int)

// Call

make([]string, 2, 4), "hello", 10

// Values (0xc000042748, 0x2, 0x4, 0x106abae, 0x5, 0xa)

Slice Value: 0xc000042748, 0x2, 0x4

String Value: 0x106abae, 0x5

Integer Value: 0xa

Use go build -gcflags -S to map the PC offset values, +0x39 and +0x72 for each function call.

**Example 2**

Build and run the program.

$ go build

$ ./example2

Review the stack trace.

// Stack Trace

goroutine 1 [running]:

main.example(0xc019010001)

stack\_trace/example2/example2.go:13 +0x39

main.main()

stack\_trace/example2/example2.go:8 +0x29

// Declaration

main.example(b1, b2, b3 bool, i uint8)

// Call

main.example(true, false, true, 25)

// Word value (0xc019010001)

Bits Binary Hex Value

00-07 0000 0001 01 true

08-15 0000 0000 00 false

16-23 0000 0001 01 true

24-31 0001 1001 19 25