High Performance Go

QConSF 2017 🏴

13 November 2017

Dave Cheney

Introduction

Hello, I'm David. 👋

I'm a software programming from Sydney, Australia. 🐨

I'm a bit of a fan of Go. I help organise GopherCon each year, have been involved with the Sydney Go meetup for years, and travel a lot to talk about and teach Go.

Agenda

Today we're going to take a look at techniques for writing high performance Go applications.

We're going to focus on three areas in this presentation:

- Benchmarking
- Performance measurement and profiling
- Memory management and GC

The goal is to give you, the audience, the tools you need to measure and improve the performance of your Go applications.

Benchmarking

Benchmarking

Before you can begin to tune your application, you need to establish a reliable baseline to measure the impact of your change to know if you're making things better, or worse.

In other words, *"Don't guess, measure"*

This section focuses on how to construct useful benchmarks using the Go testing framework, and gives practical tips for avoiding the pitfalls.

Benchmarking is closely related to profiling, which we'll touch on during this section, then cover in more detail in the next.

Benchmarking ground rules

Before you benchmark, 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 it, 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.

Using the testing package for benchmarking

The `testing` package has built in support for writing benchmarks.

```
// Fib computes the n'th number in the Fibonacci series.
func Fib(n int) int {
    switch n {
    case 0:
        return 0
    case 1:
        return 1
    default:
        return Fib(n-1) + Fib(n-2)
    }
}
```

fib.go

```
func BenchmarkFib20(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(20) // run the Fib function b.N times
    }
}
```

fib_test.go

DEMO: `go test -bench=. ./examples/fib`

How benchmarks work

Each benchmark is run `b.N` times until it takes longer than 1 second.

`b.N` starts at 1, if the benchmark completes in under 1 second `b.N` is increased and the benchmark run again.

`b.N` increases in the approximate sequence; 1, 2, 3, 5, 10, 20, 30, 50, 100, ...

```
% go test -bench=. ./examples/fib
BenchmarkFib20-4           30000              46408 ns/op
PASS
ok      _/Users/dfc/devel/high-performance-go-workshop/examples/fib     1.910s
```

Beware: below the µs mark you will start to see the relativistic effects of instruction reordering and code alignment.

Run benchmarks longer to get more accuracy; `go test -benchtime=10s`

Tip: If this is required, codify it in a `Makefile` so everyone is comparing apples to apples.

Comparing benchmarks

For repeatable results, you should run benchmarks multiple times.

You can do this manually, or use the `-count` flag.

```
% go test -bench=. -count=10 | tee old.txt
goos: darwin
goarch: amd64
BenchmarkFib20-4            30000              52201 ns/op
BenchmarkFib20-4            30000              53010 ns/op
BenchmarkFib20-4            30000              50739 ns/op
BenchmarkFib20-4            30000              51434 ns/op
BenchmarkFib20-4            30000              51697 ns/op
BenchmarkFib20-4            30000              52610 ns/op
BenchmarkFib20-4            30000              53618 ns/op
BenchmarkFib20-4            30000              51615 ns/op
BenchmarkFib20-4            30000              52105 ns/op
BenchmarkFib20-4            30000              52031 ns/op
PASS
ok      _/Users/dfc/devel/qconsf-2017/examples/fib      21.035s
```

Comparing benchmarks (cont.)

Determining the performance delta between two sets of benchmarks can be tedious and error prone.

```
% go test -bench=. -count=10 | tee old.txt
```

DEMO: Improve `Fib`

```
% go test -bench=. -count=10 | tee new.txt
```

Tools like rsc.io/benchstat (https://godoc.org/rsc.io/benchstat) are useful for comparing results.

```
% go get -u rsc.io/benchstat
% benchstat {old,new}.txt

name   old time/op  new time/op  delta
Fib-4  59.4µs ±11%  37.1µs ± 9%  -37.56%  (p=0.000 n=20+20)
```

Tip: p values above 0.05 are suspect, increase `-count` to add more samples.

Avoid benchmarking start up costs

Sometimes your benchmark has a once per run setup cost. `b.ResetTimer()` will can be used to ignore the time accrued in setup.

```
func BenchmarkExpensive(b *testing.B) {
    boringAndExpensiveSetup()
    b.ResetTimer()
    for n := 0; n < b.N; n++ {
        // function under test
    }
}
```

If you have some expensive setup logic *per loop iteration*, use `b.StopTimer()` and `b.StartTimer()` to pause the benchmark timer.

```
func BenchmarkComplicated(b *testing.B) {
    for n := 0; n < b.N; n++ {
        b.StopTimer()
        complicatedSetup()
        b.StartTimer()
        // function under test
    }
}
```

Benchmarking allocations

Allocation count and size is strongly correlated with benchmark time.

You can tell the `testing` framework to record the number of allocations made by code under test.

```
package q

func BenchmarkRead(b *testing.B) {
    b.ReportAllocs()
    for n := 0; n < b.N; n++ {
        // function under test
    }
}
```

Avoid string concatenation

Go strings are immutable. Concatenating two strings generates a third. Which of the following is fastest?

```
s := request.ID
s += " " + client.Addr().String()
s += " " + time.Now().String()
r = s
```

```
var b bytes.Buffer
fmt.Fprintf(&b, "%s %v %v", request.ID, client.Addr(), time.Now())
r = b.String()
```

```
r = fmt.Sprintf("%s %v %v", request.ID, client.Addr(), time.Now())
```

```
b := make([]byte, 0, 40)
b = append(b, request.ID...)
b = append(b, ' ')
b = append(b, client.Addr().String()...)
b = append(b, ' ')
b = time.Now().AppendFormat(b, "2006-01-02 15:04:05.999999999 -0700 MST")
r = string(b)
```

DEMO: `go test -bench=. ./examples/concat`

Watch out for compiler optimisations

This example comes from issue 14813 (https://github.com/golang/go/issues/14813#issue-140603392).

```
const m1 = 0x5555555555555555
const m2 = 0x3333333333333333
const m4 = 0x0f0f0f0f0f0f0f0f
const h01 = 0x0101010101010101

func popcnt(x uint64) uint64 {
    x -= (x >> 1) & m1
    x = (x & m2) + ((x >> 2) & m2)
    x = (x + (x >> 4)) & m4
    return (x * h01) >> 56
}

func BenchmarkPopcnt(b *testing.B) {
    for n := 0; n < b.N; n++ {
        popcnt(uint64(n))
    }
}
```

How fast will this function benchmark?

```
% go test -bench=. ./examples/popcnt
```

What happened?

popcnt is a leaf function, so the compiler can inline it.

Because the function is inlined, the compiler can see it has no side effects, so the call is eliminated. This is what the compiler sees:

```
func BenchmarkPopcnt(b *testing.B) {
    for n := 0; n < b.N; n++ {
        // optimised away
    }
}
```

The same optimisations that make real code fast, by removing unnecessary computation, are the same ones that remove benchmarks that have no observable side effects.

This is only going to get more common as the Go compiler improves.

DEMO: show how to fix popcnt

Performance measurement and profiling

Performance measurement and profiling

`testing.B` is useful for *microbenchmarks*.

Microbenchmarks are useful for tuning the performance of a hot piece of code, but it's impractical (and unreliable) to write a `testing.B` benchmark for entire programs–you'd get more reliable results with `time(1)`.

In this section we'll explore the profiling tools built into Go to investigate the operation of the program from the inside.

pprof

The first tool we're going to be talking about today is *pprof*.

pprof (https://github.com/google/pprof) descends from the Google Perf Tools (https://github.com/gperftools/gperftools) suite.

`pprof` profiling is built into the Go runtime.

It consists of two parts:

- `runtime/pprof` package built into every Go program
- `go tool pprof` for investigating profiles.

CPU profiling

CPU profiling is the most common type of profile, and the most obvious.

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 complete 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.

Stack allocations are assumed to be free and are *not tracked* in the memory profile.

Memory profiling, like CPU profiling is sample based, by default memory profiling samples 1 in every 1000 allocations. This rate can be changed.

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.

Other supported profiles

Block profiling is similar to a CPU profile, but it records the amount of time a goroutine spent waiting for a shared resource.

Block profiling can show you when a large number of goroutines *could* make progress, but were *blocked*. This can be useful for determining *concurrency* bottlenecks in your application. Blocking includes:

- Sending or receiving on a unbuffered channel.
- Sending to a full channel, receiving from an empty one.
- Trying to `Lock` a `sync.Mutex` that is locked by another goroutine.

Mutex profiling records the stack traces of the *holder* of a contended mutex.

Thread creation profiling records the stack traces that led to the creation of new OS threads.

These are very specialised tools and should not be used until you believe you have eliminated all your CPU and memory usage bottlenecks.
In the interests of time, I'm only going to talk about CPU profiling.

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 profile's at the same time, they will observe their own interactions and throw off your results.

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

Profiling applications

The Go runtime's profiling interface is in the `runtime/pprof` package.

`runtime/pprof` is a very low level tool, and for historic reasons the interfaces to the different kinds of profile are not uniform.

A few years ago I wrote a small package, github.com/pkg/profile (https://github.com/pkg/profile), to make it easier to profile an application.

```
import "github.com/pkg/profile"

func main() {
      defer profile.Start().Stop()
      ...
}
```

Tip: `pkg/profile` will prevent you from enabling more than one profile at once.

Profiling godoc

DEMO: add CPU profiling to godoc.

1. edit `$GOPATH/src/golang.org/x/tools/cmd/godoc/main.go`, add

```
defer profile.Start(profile.CPUProfile).Stop()
```

2. `go install -v golang.org/x/tools/cmd/godoc`
3. `godoc -http=:8000`
4. `go tool pprof $PROFILE`

Using pprof

Now that I've talked about what pprof can measure, I will talk about how to use pprof to analyse a profile.

Starting with Go 1.9, `pprof` only requires one argument.

```
go tool pprof $PROFILE
```

Tip: If you're using Go 1.8, `pprof` takes two arguments and will produce the wrong result if you give it just one. Upgrade to Go 1.9.

Using pprof (cont.)

This is a sample CPU profile:

```
% go tool pprof /tmp/c.p
Entering interactive mode (type "help" for commands)
(pprof) top
Showing top 15 nodes out of 63 (cum >= 4.85s)
      flat  flat%   sum%        cum   cum%
    21.89s  9.84%  9.84%    128.32s 57.71%  net.(*netFD).Read
    17.58s  7.91% 17.75%     40.28s 18.11%  runtime.exitsyscall
    15.79s  7.10% 24.85%     15.79s  7.10%  runtime.newdefer
    12.96s  5.83% 30.68%    151.41s 68.09%  test_frame/connection.(*ServerConn).readBytes
    11.27s  5.07% 35.75%     23.35s 10.50%  runtime.reentersyscall
    10.45s  4.70% 40.45%     82.77s 37.22%  syscall.Syscall
     9.38s  4.22% 44.67%      9.38s  4.22%  runtime.deferproc_m
     9.17s  4.12% 48.79%     12.73s  5.72%  exitsyscallfast
     8.03s  3.61% 52.40%     11.86s  5.33%  runtime.casgstatus
     7.66s  3.44% 55.85%      7.66s  3.44%  runtime.cas
     7.59s  3.41% 59.26%      7.59s  3.41%  runtime.onM
     6.42s  2.89% 62.15%    134.74s 60.60%  net.(*conn).Read
     6.31s  2.84% 64.98%      6.31s  2.84%  runtime.writebarrierptr
     6.26s  2.82% 67.80%     32.09s 14.43%  runtime.entersyscall
```

Often this output is hard to understand.

Using pprof (cont.)

A better way to understand your profile is to visualise it.

```
% go tool pprof /tmp/c.p
Entering interactive mode (type "help" for commands)
(pprof) web
```

Opens a web page with a graphical display of the profile.

images/profile.svg (images/profile.svg)

Note: visualisation requires graphviz.

I find this method to be superior to the text mode, I strongly recommend you try it.

pprof also supports these modes in a non interactive form with flags like `-svg`, `-pdf`, etc. See `go tool pprof -help` for more details.

Further reading: Profiling Go programs (http://blog.golang.org/profiling-go-programs)

Further reading: Debugging performance issues in Go programs (https://software.intel.com/en-us/blogs/2014/05/10/debugging-performance-issues-in-go-programs)

Profiling benchmarks

The `testing` package has built in support for generating CPU, memory, and block profiles.

- `-cpuprofile=$FILE` writes a CPU profile to `$FILE`.
- `-memprofile=$FILE`, writes a memory profile to `$FILE`, `-memprofilerate=N` adjusts the profile rate to `1/N`.
- `-blockprofile=$FILE`, writes a block profile to `$FILE`.

Example: Running benchmarks of the `bytes` package.

```
% go test -run=XXX -bench=. -cpuprofile=c.p bytes
% go tool pprof c.p
```

Note: use `-run=XXX` to disable tests, you only want to profile benchmarks. You can also use `-run=^$` to accomplish the same thing.

Go Execution tracer

The execution tracer was developed by Dmitry Vyukov (https://github.com/dvyukov) for Go 1.5 and remained under documented, and under utilised, until last year.

Unlike sample based profiling, the execution tracer is integrated into the Go runtime, so it doesn't just know what a Go program is doing at a particular point in time, but *why*.

Captures with nanosecond precision:

- goroutine creation/start/end
- goroutine blocking/unblocking
- network blocking
- system calls
- GC events

go tool trace

Before we go on there are some things we should talk about the usage of the trace tool.

- The tool uses the javascript debugging support built into Chrome. Trace profiles can only be viewed in Chrome, they won't work in Firefox, Safari, IE/Edge. Sorry.
- Because this is a Google product, it supports keyboard shortcuts; use `WASD` to navigate, use ? to get a list.
- Viewing traces can take a **lot** of memory. Seriously, 4Gb won't cut it, 8Gb is probably the minimum, more is definitely better.

go tool trace (cont.)

DEMO: add an execution trace to godoc.

1. edit `$GOPATH/src/golang.org/x/tools/cmd/godoc/main.go`, add

```
defer profile.Start(profile.TraceProfile).Stop()
```

2. `go install -v golang.org/x/tools/cmd/godoc`
3. `godoc -http=:8000`
4. `go tool trace $PROFILE`

Tracing running applications

In the previous examples we ran the trace over the whole program.

As you saw, traces can be very large, even for small amounts of time, so collecting trace data continually would generate far too much data. Also, tracing can have an impact on the speed of your program, especially if there is a lot of activity.

What we want is a way to collect a short trace from a running program.

Fortuntately, the `net/http/pprof` package has just such a facility. Adding this line to your `main` package:

```
import _ "net/http/pprof"
```

It will register tracing and profiling routes with `http.DefaultServeMux`.

Mandelbrot microservice

It's 2017, generating Mandelbrots is pointless unless you can offer them on the internet as a microsevice.

Thus, I present to you, *Mandelweb*

```
% go run examples/mandelweb/mandelweb.go
2017/09/17 15:29:21 listening on http://127.0.0.1:8080/
```

127.0.0.1:8080/mandelbrot (http://127.0.0.1:8080/mandelbrot)

We can grab a five second trace from mandelweb with `curl` (or `wget`)

```
curl -o trace.out http://127.0.0.1:8080/debug/pprof/trace?seconds=5
```

Generating some load

The previous example was interesting, but an idle webserver has, by definition, no performance issues. We need to generate some load. For this I'm using hey by JBD (https://github.com/rakyll/hey).

```
go get -u github.com/rakyll/hey
```

Let's start with one request per second.

```
hey -c 1 -n 1000 -q 1  http://127.0.0.1:8080/mandelbrot
```

And with that running, in another window collect the trace

```
% curl -o trace.out http://127.0.0.1:8080/debug/pprof/trace?seconds=5
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 66169    0 66169    0     0  13233      0 --:--:--  0:00:05 --:--:-- 17390
% go tool trace trace.out
2017/09/17 16:09:30 Parsing trace...
2017/09/17 16:09:30 Serializing trace...
2017/09/17 16:09:30 Splitting trace...
2017/09/17 16:09:30 Opening browser. Trace viewer is listening on http://127.0.0.1:60301
```

Further reading

- Rhys Hiltner, Go's execution tracer (https://www.youtube.com/watch?v=mmqDlbWk_XA) (dotGo 2016)
- Rhys Hiltner, An Introduction to "go tool trace" (https://www.youtube.com/watch?v=V74JnrGTwKA) (GopherCon 2017)
- Dave Cheney, Seven ways to profile Go programs (https://www.youtube.com/watch?v=2h_NFBFrciI) (GolangUK 2016)
- Dave Cheney, High performance Go workshop (https://dave.cheney.net/training#high-performance-go)
- Francesc Campoy, just for func #22: using the Go execution tracer (https://www.youtube.com/watch?v=ySy3sR1LFCQ)

Memory management and GC

Memory management and GC

Go is a garbage collected language. This is a design principle, it will not change.

As a garbage collected language, the performance of Go programs is often determined by their interaction with the garbage collector.

Next to your choice of algorithms, memory consumption is the most important factor that determines the performance and scalability of your application.

This section discusses the operation of the garbage collector and strategies for lowering memory usage if garbage collector performance is a bottleneck.

Go GC is focused on reducing latency

The purpose of a garbage collector is to present the illusion that there is an infinite amount of memory available to the program.

You may disagree with this statement, but this is the base assumption of how garbage collector designers think.

The Go GC is designed for low latency servers and interactive applications.

The Go GC favors *lower latency* over *maximum throughput*; it moves some of the allocation cost to the mutator to reduce the cost of cleanup later.

Garbage collector design

The design of the Go GC has changed over the years

- Go 1.0, stop the world mark sweep collector based heavily on tcmalloc.
- Go 1.3, fully precise collector, wouldn't mistake big numbers on the heap for pointers, thus leaking memory.
- Go 1.5, new GC design, focusing on *latency* over *throughput*.
- Go 1.6, GC improvements, handling larger heaps with lower latency.
- Go 1.7, small GC improvements, mainly refactoring.
- Go 1.8–1.9, further work to reduce STW times, now down to the 100 microsecond range.
- Go 1.10, ROC collector is an experiment to extend the idea of escape analysis per goroutine.

Garbage collector monitoring

A simple way to obtain a general idea of how hard the garbage collector is working is to enable the output of GC logging.

These stats are always collected, but normally suppressed, you can enable their display by setting the `GODEBUG` environment variable.

```
% env GODEBUG=gctrace=1 godoc -http=:8000
gc 1 @0.017s 8%: 0.021+3.2+0.10+0.15+0.86 ms clock, 0.043+3.2+0+2.2/0.002/0.009+1.7 ms cpu, 5->6->1 MB,
gc 2 @0.026s 12%: 0.11+4.9+0.12+1.6+0.54 ms clock, 0.23+4.9+0+3.0/0.50/0+1.0 ms cpu, 4->6->3 MB, 6 MB go
gc 3 @0.035s 14%: 0.031+3.3+0.76+0.17+0.28 ms clock, 0.093+3.3+0+2.7/0.012/0+0.84 ms cpu, 4->5->3 MB, 3
gc 4 @0.042s 17%: 0.067+5.1+0.15+0.29+0.95 ms clock, 0.20+5.1+0+3.0/0/0.070+2.8 ms cpu, 4->5->4 MB, 4 MB
gc 5 @0.051s 21%: 0.029+5.6+0.33+0.62+1.5 ms clock, 0.11+5.6+0+3.3/0.006/0.002+6.0 ms cpu, 5->6->4 MB, 5
gc 6 @0.061s 23%: 0.080+7.6+0.17+0.22+0.45 ms clock, 0.32+7.6+0+5.4/0.001/0.11+1.8 ms cpu, 6->6->5 MB, 7
gc 7 @0.071s 25%: 0.59+5.9+0.017+0.15+0.96 ms clock, 2.3+5.9+0+3.8/0.004/0.042+3.8 ms cpu, 6->8->6 MB, 8
```

The trace output gives a general measure of GC activity.

DEMO: Show godoc with `GODEBUG=gctrace=1` enabled

Recommendation: use this env var in production, it has no performance impact.

Garbage collector monitoring (cont.)

Using `GODEBUG=gctrace=1` is good when you *know* there is a problem, but for general telemetry on your Go application I recommend the `net/http/pprof` interface.

```
import _ "net/http/pprof"
```

Importing the `net/http/pprof` package will register a handler at `/debug/pprof` with various runtime metrics, including:

- A list of all the running goroutines, `/debug/pprof/heap?debug=1`.
- A report on the memory allocation statistics, `/debug/pprof/heap?debug=1`.

Warning: `net/http/pprof` will register itself with your default `http.ServeMux`.

Be careful as this will be visible if you use `http.ListenAndServe(address, nil)`.

DEMO: `godoc -http=:8080`, show `/debug/pprof`.

Garbage collector tuning

The Go runtime provides one environment variable to tune the GC, `GOGC`.

The formula for GOGC is as follows.

```
goal = reachable * (1 + GOGC/100)
```

For example, if we currently have a 256MB heap, and `GOGC=100` (the default), when the heap fills up it will grow to

```
512MB = 256MB * (1 + 100/100)
```

- Values of `GOGC` greater than 100 causes the heap to grow faster, reducing the pressure on the GC.
- Values of `GOGC` less than 100 cause the heap to grow slowly, increasing the pressure on the GC.

The default value of 100 is *just a guide*. you should choose your own value *after profiling your application with production loads*.

Reduce allocations

Make sure your APIs allow the caller to reduce the amount of garbage generated.

Consider these two Read methods

```
func (r *Reader) Read() ([]byte, error)
func (r *Reader) Read(buf []byte) (int, error)
```

The first Read method takes no arguments and returns some data as a `[]byte`. The second takes a `[]byte` buffer and returns the amount of bytes read.

The first Read method will *always* allocate a buffer, putting pressure on the GC. The second fills the buffer it was given.

strings and []bytes

In Go `string` values are immutable, `[]byte` are mutable.

Most programs prefer to work `string`, but most IO is done with `[]byte`.

Avoid `[]byte` to string conversions wherever possible, this normally means picking one representation, either a `string` or a `[]byte` for a value. Often this will be `[]byte` if you read the data from the network or disk.

The `bytes` (https://golang.org/pkg/bytes/) package contains many of the same operations— `Split`, `Compare`, `HasPrefix`, `Trim`, etc—as the `strings` (https://golang.org/pkg/strings/) package.

Under the hood `strings` uses same assembly primitives as the `bytes` package.

Preallocate slices if the length is known

Append is convenient, but wasteful.

Slices grow by doubling up to 1024 elements, then by approximately 25% after that. What is the capacity of b after we append one more item to it?

```
func main() {
    b := make([]int, 1024)
    b = append(b, 99)
    fmt.Println("len:", len(b), "cap:", cap(b))
}
```

Run

If you use the append pattern you could be copying a lot of data and creating a lot of garbage.

Preallocate slices if the length is known (cont.)

If know know the length of the slice beforehand, then pre-allocate the target to avoid copying and to make sure the target is exactly the right size.

Before:

```
var s []string
for _, v := range fn() {
	s = append(s, v)
}
return s
```

After:

```
vals := fn()
s := make([]string, len(vals))
for i, v := range vals {
	s[i] = v
}
return s
```

Using sync.Pool

The sync package comes with a `sync.Pool` type which is used to reuse common objects.

`sync.Pool` has no fixed size or maximum capacity. You add to it and take from it until a GC happens, then it is emptied unconditionally.

```
var pool = sync.Pool{New: func() interface{} { return make([]byte, 4096) }}

func fn() {
    buf := pool.Get().([]byte) // takes from pool or calls New
    // do work
    pool.Put(buf) // returns buf to the pool
}
```

Warning: `sync.Pool` is not a cache. It can and will be emptied *at any time*.

Do not place important items in a `sync.Pool`, they will be discarded.

Use streaming IO interfaces

Where-ever possible avoid reading data into a `[]byte` and passing it around.

Depending on the request you may end up reading megabytes (or more!) of data into memory. This places huge pressure on the GC, which will increase the average latency of your application.

Instead use `io.Reader` and `io.Writer` to construct processing pipelines to cap the amount of memory in use per request.

For efficiency, consider implementing `io.ReaderFrom` / `io.WriterTo` if you use a lot of `io.Copy`. These interface are more efficient and avoid copying memory into a temporary buffer.

Conclusion

Conclusion

Start with the simplest possible code.

Measure. Profile your code to identify the bottlenecks, *do not guess*.

If performance is good, *stop*. You don't need to optimise everything, only the hottest parts of your code.

As your application grows, or your traffic pattern evolves, the performance hot spots will change.

Don't leave complex code that is not performance critical, rewrite it with simpler operations if the bottleneck moves elsewhere.

Conclusion (cont.)

Always write the simplest code you can, the compiler is optimised for *normal* code.

Shorter code is faster code; Go is not C++, do not expect the compiler to unravel complicated abstractions.

Shorter code is *smaller* code; which is important for the CPU's cache.

Pay very close attention to allocations, avoid unnecessary allocation where possible.

Don't trade performance for reliability

"I can make things very fast if they don't have to be correct."

Russ Cox

"Readable means reliable"

Rob Pike

Performance and reliability are equally important.

I see little value in making a very fast server that panics, deadlocks or OOMs on a regular basis.

Don't trade performance for reliability

Thank you

Dave Cheney
dave@cheney.net (mailto:dave@cheney.net)
http://dave.cheney.net/ (http://dave.cheney.net/)
@davecheney (http://twitter.com/davecheney)