High Performance Go

Tokyo
4 December 2016

Dave Cheney

License and Materials

The materials for this presentation are available on GitHub:

github.com/davecheney/high-performance-go-workshop (https://github.com/davecheney/high-performance-go-workshop)

You are encouraged to remix, transform, or build upon the material, providing you give appropriate credit and distribute your contributions under the same license.

If you have suggestions or corrections to this presentation, please raise an issue on the GitHub project (https://github.com/davecheney/high-performance-go-workshop/issues).

Agenda

This workshop is aimed at development teams who are building production Go applications intended for high scale deployment.

Today we are going to cover five areas:

- What does performance mean, what is possible?
- Benchmarking
- Performance measurement and profiling
- Memory management and GC tuning
- Concurrency

After each section we'll have time for questions.

One more thing ...

This isn't a lecture, it's a conversation.

If you don't understand something, or think what you're hearing is incorrect, please ask.

What does performance mean, what is possible?

What does performance mean?

Before we talk about writing high performance code, we need to talk about the hardware that will execute this code.

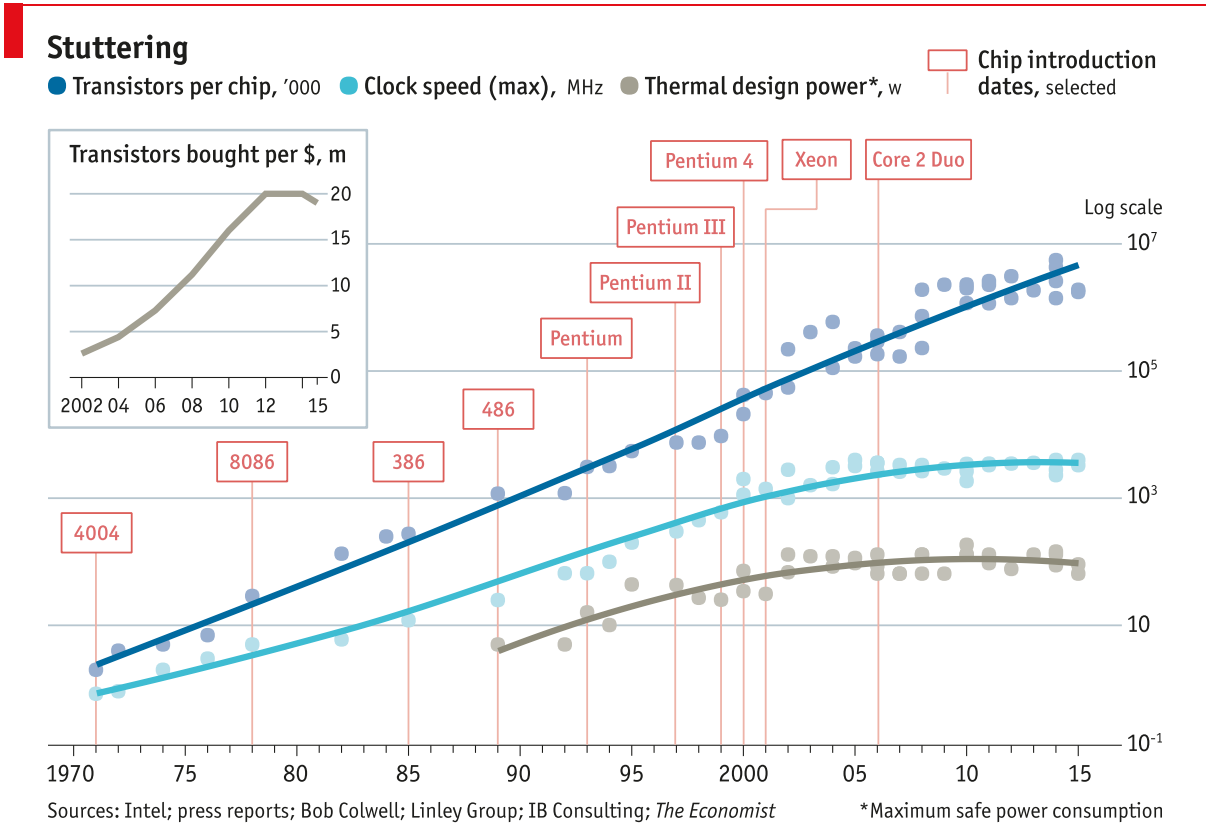
What are its properties and how have they changed over time?

As software authors we have benefited from Moore's Law, the doubling of the number of available transistors on a chip every 18 months, for 50 years.

No other industry has experienced a *six order of magnitude* improvement in their tools in the space of a lifetime.

But this is all changing.

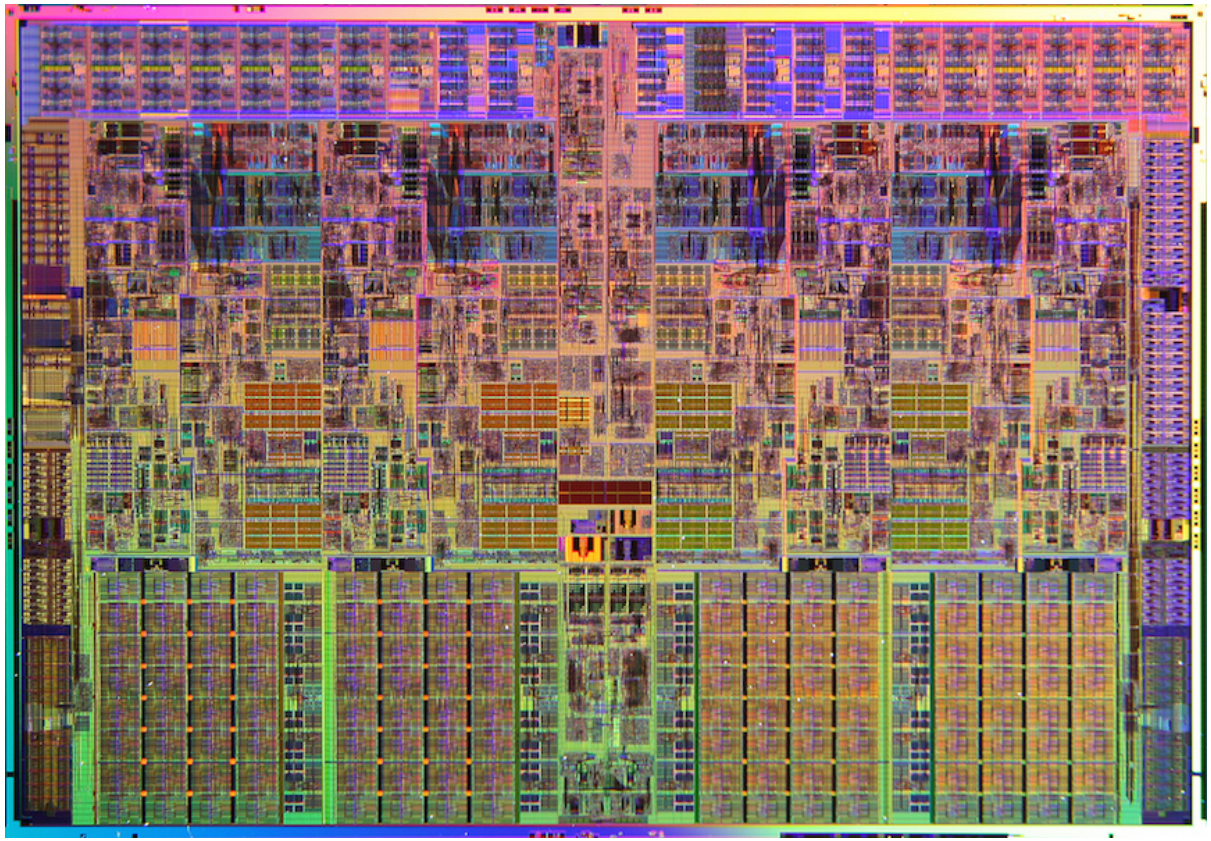
The CPU



The number of transistors per CPU die continues to increase.

However clock speeds have not increased in a decade, and transistors per dollar has started to fall.

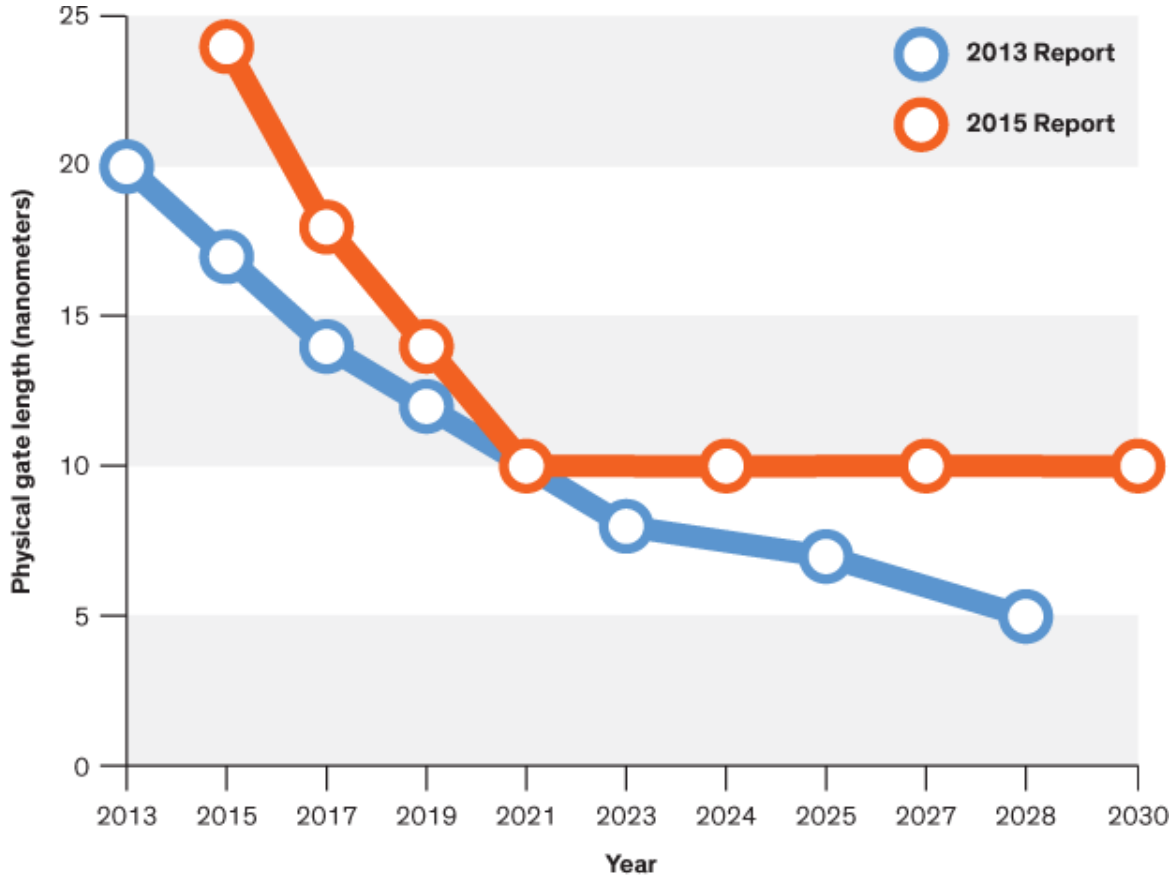
More cores



Over the last decade performance, especially on the server, has been dictated by adding more CPU cores.

CPU core count is dominated by heat dissipation and cost.

Transistor size reductions have run out of steam



Shrinking the size of a transistor, to fit more in the same sized die, is hitting a wall.

CPU dies can be made larger, but that increases latency, and cost per transistor.

Modern CPUs are optimised for bulk transfers

"Modern processors are a like nitro fueled funny cars, they excel at the quarter mile. Unfortunately modern programming languages are like Monte Carlo, they are full of twists and turns."

David Ungar, OOPSLA (year unknown)

Much of the improvement in performance in the last two decades has come from architectural improvements:

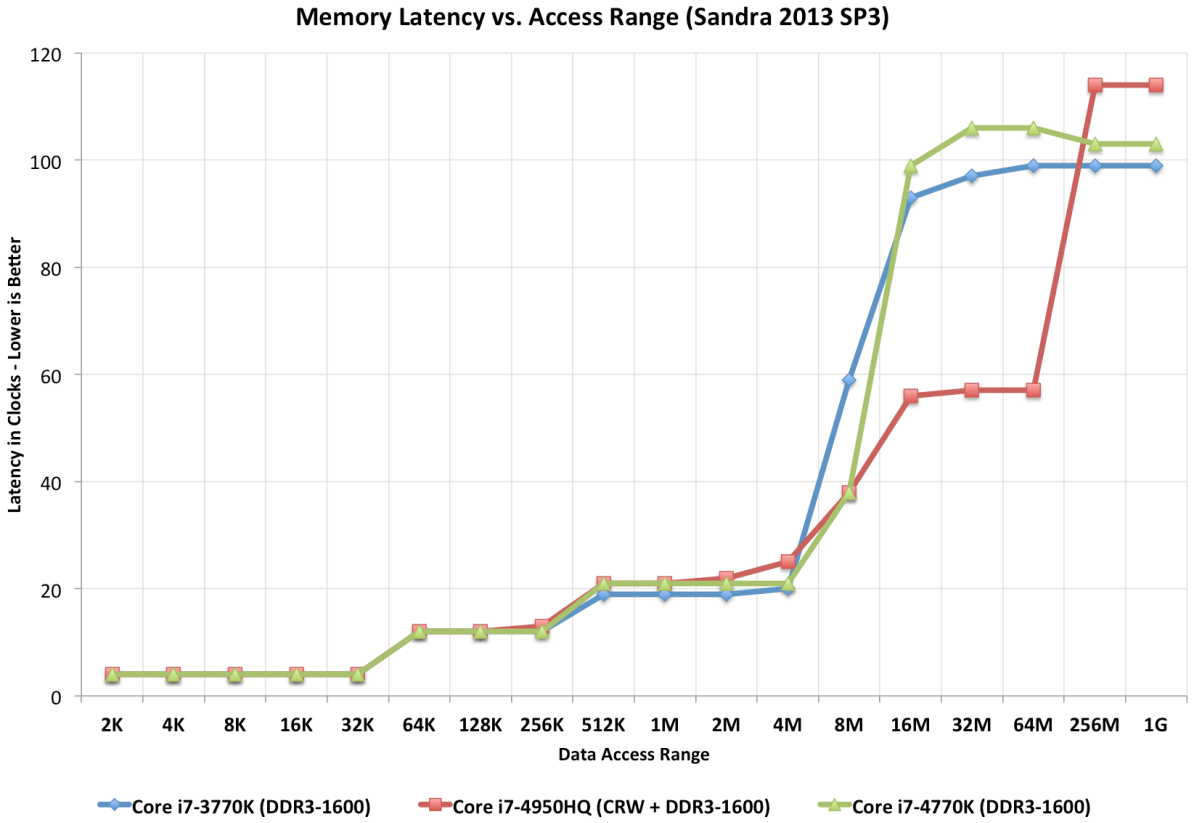
- out of order execution (super-scalar)
- speculative execution
- vector (SSE) instructions

Thus, modern CPUs are optimised for bulk transfers and bulk operations. At every level, the setup cost of an operation encourages you to work in bulk.

e.g. memory is not loaded per byte, but per multiple of cache lines, this is why alignment is becoming less of an issue.

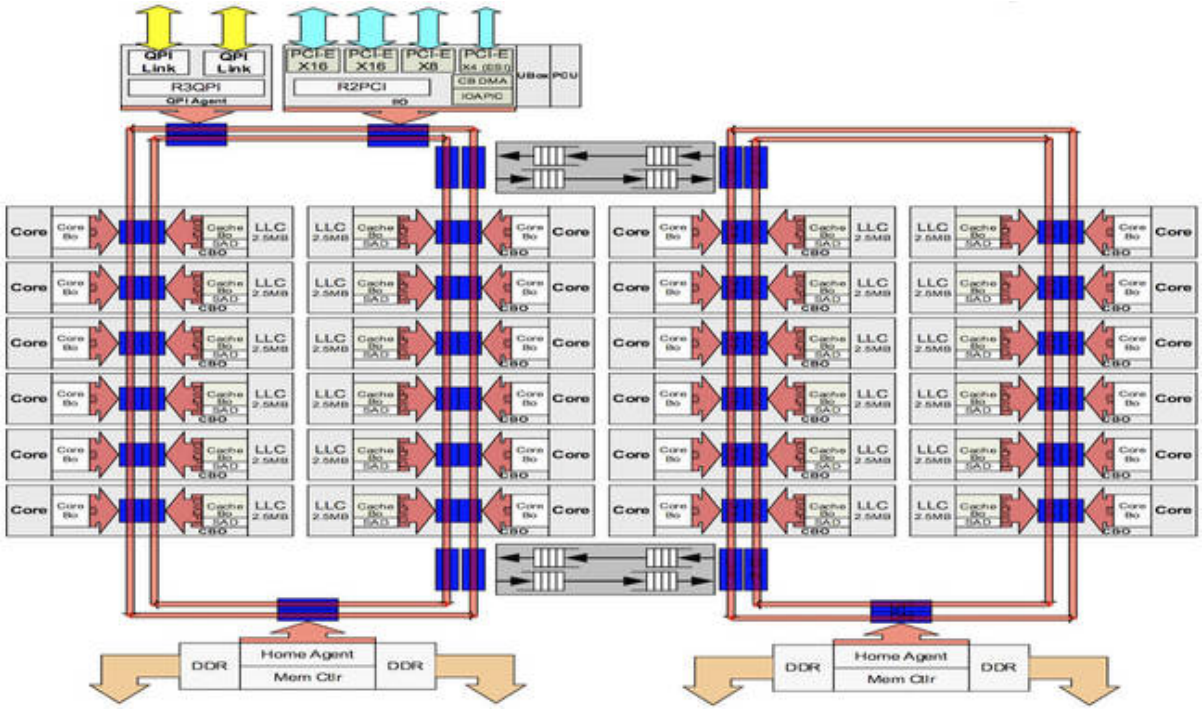
Memory

Physical memory attached to a server has increased geometrically.



But, in terms of processor cycles lost, physical memory is still as far away as ever.

Cache rules everything around it



Cache rules everything around it, but it is small, and will remain small because the speed of light determines how large a cache can be for a certain latency.

You can have a larger cache, but it will be slower because, in a universe where electricity travels a foot every nanosecond, distance equals latency.

Network and disk I/O are still expensive

Network and disk I/O are still expensive, so expensive that the Go runtime will schedule something else while those operations are in progress.

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

The free lunch is over

In 2005 Herb Sutter, the C++ committee leader, wrote an article entitled *The free lunch is over* (http://www.gotw.ca/publications/concurrency-ddj.htm) .

In his article Sutter discussed all the points I covered and asserted that programmers could no longer rely on faster hardware to fix slow programs—or slow programming languages.

Now, a decade later, there is no doubt that Herb Sutter was right. Memory is slow, caches are too small, CPU clock speeds are going backwards, and the simple world of a single threaded CPU is long gone.

A fast programming language

It's time for the software to come to the party.

As Rick Hudson noted at GopherCon (https://www.youtube.com/watch?v=aiv1JOfMjm0) in 2015, it's time for a programming language that works *with* the limitations of today's hardware, rather than continue to ignore the reality that CPU designers find themselves.

So, for best performance on today's hardware in today's world, you need a programming language which:

- Is compiled, not interpreted.
- Permits efficient code to be written.
- Lets programmers talk about memory effectively, think structs vs java objects
- Has a compiler that produces efficient code, it has to be small code as well, because cache.

Obviously the language I'm talking about is the one we're here to discuss: Go.

Discussion

Discussion:

- Do you agree that computers are no longer getting faster?
- Do you think we need to change languages to take advantage of modern hardware?

Further reading

The Go Programming language (Nov 10, 2009) (https://www.youtube.com/watch?v=rKnDgT73v8s)

OSCON 2010: Rob Pike, "Public Static Void" (https://www.youtube.com/watch?v=5kj5ApnhPAE)

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 it in 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.
- There is a kernel bug on OS X versions before El Capitan; upgrade or avoid profiling on OS X. This also affects other *BSDs.

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 {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}
```

fib.go

```
import "testing"

func BenchmarkFib(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
BenchmarkFib-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`
- Run benchmarks multiple times; `go test -count=10`

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.

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

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

```
% go test -c
% mv fib.test fib.golden
```

DEMO: Improve `Fib`

```
% go test -c
% ./fib.golden -test.bench=. -test.count=5 > old.txt
% ./fib.test -test.bench=. -test.count=5 > new.txt
% benchstat old.txt new.txt
```

DEMO: `benchstat {old,new}.txt`

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
    }
}
```

DEMO: `go test -run=^$ -bench=. bufio`

Note: you can also use the `go test -benchmem` flag to do the same for *all* benchmarks.

DEMO: `go test -run=^$ -bench=. -benchmem bufio`

Watch out for compiler optimisations

This example comes from issue 14813 (https://github.com/golang/go/issues/14813#issue-140603392). How fast will this function benchmark?

```
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 i := 0; i < b.N; i++ {
        popcnt(uint64(i))
    }
}
```

What happened?

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

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 i := 0; i < b.N; i++ {
        // 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

Benchmark mistakes

The `for` loop is crucial to the operation of the benchmark.

Here are two incorrect benchmarks, can you explain what is wrong with them?

```
func Fib(n int) int {
    a, b := 0, 1
    for i := 0; i < n; i++ {
        a, b = b, a+b
    }
    return a
}

func BenchmarkFibWrong(b *testing.B) {
    Fib(b.N)
}

func BenchmarkFibWrong2(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(n)
    }
}
```

Discussion

Are there any questions?

Perhaps it is time for a break.

Performance measurement and profiling

Performance measurement and profiling

In the previous section we studied how to measure the performance of programs from the outside.

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

pprof

The primary 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 of tools.

`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.

pprof supports several types of profiling, we'll discuss three of these today:

- CPU profiling.
- Memory profiling.
- Block (or blocking) profiling.

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.

Personal Opinion: I do not find memory profiling useful for finding memory leaks. There are better ways to determine how much memory your application is using. We will discuss these later in the presentation.

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 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.

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 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.

Using pprof

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

pprof should always be invoked with *two* arguments.

```
go tool pprof /path/to/your/binary /path/to/your/profile
```

The `binary` argument **must** be the binary that produced this profile.

The `profile` argument **must** be the profile generated by this binary.

Warning: Because pprof also supports an online mode where it can fetch profiles from a running application over http, the pprof tool can be invoked without the name of your binary (issue 10863 (https://github.com/golang/go/issues/10863)):

```
go tool pprof /tmp/c.pprof
```

Do not do this or pprof will report your profile is empty.

Using pprof (cont.)

This is a sample CPU profile:

```
% go tool pprof $BINARY /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 application /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)

Using pprof (cont.)

The output of a memory profile can be similarly visualised.

```
% go build -gcflags='-memprofile=/tmp/m.p'
% go tool pprof --alloc_objects -svg $(go tool -n compile) /tmp/m.p > alloc_objects.svg
% go tool pprof --inuse_objects -svg $(go tool -n compile) /tmp/m.p > inuse_objects.svg
```

Memory profiles come in two varieties

- Alloc objects reports the call site where each allocation was made

  images/alloc_objects.svg (images/alloc_objects.svg)

- Inuse objects reports the call site where an allocation was made *iff* it was reachable at the end of the profile

  images/inuse_objects.svg (images/inuse_objects.svg)

DEMO: `examples/inuseallocs`

Using pprof (cont.)

Here is a visualisation of a block profile:

```
% go test -run=XXX -bench=ClientServer -blockprofile=/tmp/b.p net/http
% go tool pprof -svg http.test /tmp/b.p > block.svg
```

images/block.svg (images/block.svg)

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`.

Using any of these flags also preserves the binary.

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

Profiling applications

Profiling `testing` benchmarks is useful for *microbenchmarks*.

We use microbenchmarks inside the standard library to make sure individual packages do not regress, but what if you want to profile a complete application?

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()
      ...
}
```

Exercise

- Generate a profile from a piece of code you know well. If you don't have a code sample, try profiling godoc.
- If you were to generate a profile on one machine and inspect it on another, how would you do it?

Framepointers

Go 1.7 has been released and along with a new compiler for amd64, the compiler now enables frame pointers by default.

The frame pointer is a register that always points to the top of the current stack frame.

Framepointers enable tools like `gdb(1)`, and `perf(1)` to understand the Go call stack.

We won't cover these tools in this workshop, but you can read and watch a presentation I gave on seven different ways to profile Go programs.

Further reading:

- Seven ways to profile a Go program (slides) (https://talks.godoc.org/github.com/davecheney/presentations/seven.slide)
- Video (30 mins) (https://www.youtube.com/watch?v=2h_NFBFrciI)
- Recording (60 mins) (https://www.bigmarker.com/remote-meetup-go/Seven-ways-to-profile-a-Go-program)

go tool trace

In Go 1.5, Dmitry Vyukov added a new kind of profiling to the runtime; execution trace profiling (https://golang.org/doc/go1.5#trace_command).

Gives insight into dynamic execution of a program.

Captures with nanosecond precision:

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

Execution traces are essentially undocumented , see github/go#16526

(https://github.com/golang/go/issues/16526)

go tool trace (cont.)

Generating an execution trace

```
% cd $(go env GOROOT)/test/bench/go1
% go test -bench=HTTPClientServer -trace=/tmp/t.p
```

Viewing the trace:

```
% go tool trace /tmp/t.p
2016/08/13 17:01:04 Parsing trace...
2016/08/13 17:01:04 Serializing trace...
2016/08/13 17:01:05 Splitting trace...
2016/08/13 17:01:06 Opening browser
```

Bonus: `github.com/pkg/profile` (https://github.com/pkg/profile/releases/tag/v1.2.0) supports generating trace profiles.

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

Exercises

- Create a trace profile of your application using `go tool trace` and `github.com/pkg/profile` (https://github.com/pkg/profile/releases/tag/v1.2.0).

Compiler optimisations

Compiler optimisations

This section gives a brief background on three important optimisations that the Go compiler performs.

- Escape analysis
- Inlining
- Dead code elimination

These are all handled in the front end of the compiler, while the code is still in its AST form; then the code is passed to the SSA compiler for further optimisation.

Escape analysis

A compliant Go implementation *could* store every allocation on the heap, but that would put a lot of pressure on the gc.

However, the stack exists as a cheap place to store local variables; there is no need to garbage collect things on the stack.

In some languages, like C and C++, stack/heap allocation is manual, and a common cause of memory corruption bugs.

In Go, the compiler automatically moves local values to the heap if they live beyond the lifetime of the function call. It is said that the value *escapes* to the heap.

But the compiler can also do the opposite, it can find things which would be assumed to be allocated on the heap, `new`, `make`, etc, and move them to stack.

Escape analysis (example)

Sum adds the ints between 1 and 100 and returns the result.

```
// Sum returns the sum of the numbers 1 to 100
func Sum() int {
    const count = 100
    numbers := make([]int, 100)
    for i := range numbers {
        numbers[i] = i + 1
    }

    var sum int
    for _, i := range numbers {
        sum += i
    }
    return sum
}
```

examples/esc/sum.go

Because the numbers slice is only referenced inside Sum, the compiler will arrange to store the 100 integers for that slice on the stack, rather than the heap. There is no need to garbage collect `numbers`, it is automatically free'd when `Sum` returns.

Investigating escape analysis

Prove it!

To print the compilers escape analysis decisions, use the `-m` flag.

```
% go build -gcflags=-m examples/esc/sum.go
# command-line-arguments
examples/esc/sum.go:10: Sum make([]int, 100) does not escape
examples/esc/sum.go:25: Sum() escapes to heap
examples/esc/sum.go:25: main ... argument does not escape
```

Line 10 shows the compiler has correctly deduced that the result of `make([]int, 100)` does not escape to the heap.

We'll come back to line 25 soon.

Escape analysis (example)

This example is a little contrived.

```
type Point struct{ X, Y int }

const Width = 640
const Height = 480

func Center(p *Point) {
    p.X = Width / 2
    p.Y = Height / 2
}

func NewPoint() {
    p := new(Point)
    Center(p)
    fmt.Println(p.X, p.Y)
}
```

`NewPoint` creates a new `*Point` value, `p`. We pass `p` to the `Center` function which moves the point to a position in the center of the screen. Finally we print the values of `p.X` and `p.Y`.

Escape analysis (example)

```
% go build -gcflags=-m examples/esc/center.go
# command-line-arguments
examples/esc/center.go:12: can inline Center
examples/esc/center.go:19: inlining call to Center
examples/esc/center.go:12: Center p does not escape
examples/esc/center.go:20: p.X escapes to heap
examples/esc/center.go:20: p.Y escapes to heap
examples/esc/center.go:18: NewPoint new(Point) does not escape
examples/esc/center.go:20: NewPoint ... argument does not escape
```

Even though p was allocated with the new function, it will not be stored on the heap, because no reference p escapes the `Center` function.

Question: What about line 20, if p doesn't escape, what is escaping to the heap?

```
examples/esc/center.go:20: p.X escapes to heap
examples/esc/center.go:20: p.Y escapes to heap
```

Escape analysis is not perfect (https://github.com/golang/go/issues/7714)

Exercise

- What is happening on line 25? Open up `examples/esc/sum.go` and see.
- Write a benchmark to provide that `Sum` does not allocate

Inlining

In Go function calls in have a fixed overhead; stack and preemption check.

Some of this is ameliorated by hardware branch predictors, but it's still a cost in terms of function size and clock cycles.

Inlining is the classical optimisation to avoid these costs.

Inlining only works on *leaf functions*, a function that does not call another. The justification for this is:

- If your function does a lot of work, then the preamble overhead will be negligible. That's why functions over a certain size (currently some count of instructions, plus a few operations which prevent inlining all together (eg. switch before Go 1.7)
- Small functions on the other hand pay a fixed overhead for a relatively small amount of useful work performed. These are the functions that inlining targets as they benefit the most.

The other reason is it makes stack traces harder to follow.

Inlining (example)

```
func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

func F() {
    const a, b = 100, 20
    if Max(a, b) == b {
        panic(b)
    }
}
```

Run

Inlining (cont.)

Again we use the `-m` flag to view the compilers optimisation decision.

```
% go build -gcflags=-m examples/max/max.go
# command-line-arguments
examples/max/max.go:4: can inline Max
examples/max/max.go:13: inlining call to Max
```

Compile `max.go` and see what the optimised version of `F()` became.

DEMO: `go build -gcflags="-m -S" examples/max/max.go 2>&1 | less`

Discussion

- Why did I declare `a` and `b` in `F()` to be constants?
- What happens if they are variables?
- What happens if they are passing into `F()` as parameters?

Dead code elimination

Why is it important that a and b are constants?

After inlining, this is what the compiler saw

```
func F() {
    const a, b = 100, 20
    if a > b {
        return
    }
    panic(b)
}
```

Run

- The call to `Max` has been inlined.
- If `a > b` then there is nothing to do, so the function returns.
- If `a < b` then the branch is false and we fall through to `panic`
- But, because a and b are constants, we know that the branch will never be false, so the compiler can optimise `F()` to a return.

Dead code elimination (cont.)

Dead code elimination work together with inlining to reduce the amount of code generated by removing loops and branches that are proven unreachable.

You can take advantage of this to implement expensive debugging, and hide it behind

```
const debug = false
```

Combined with build tags this can be very useful.

Further reading:

- Using // +build to switch between debug and release builds (http://dave.cheney.net/2014/09/28/using-build-to-switch-between-debug-and-release)
- How to use conditional compilation with the go build tool (http://dave.cheney.net/2013/10/12/how-to-use-conditional-compilation-with-the-go-build-tool)

Compiler flags Exercises

Compiler flags are provided with:

```
go build -gcflags=$FLAGS
```

Investigate the operation of the following compiler functions:

- `-S` prints the (Go flavoured) assembly of the *package* being compiled.
- `-l` controls the behaviour of the inliner; `-l` disables inlining, `-l -l` increases it (more `-l` 's increases the compiler's appetite for inlining code). Experiment with the difference in compile time, program size, and run time.
- `-m` controls printing of optimisation decision like inlining, escape analysis. `-m -m` prints more details about what the compiler was thinking.
- `-l -N` disables all optimisations.

> Further reading: Codegen Inspection by Jaana Burcu Dogan (http://go-talks.appspot.com/github.com/rakyll/talks/gcinspect/talk.slide#1)

Perhaps we shall take a break now.

Memory management and GC tuning

Memory management and GC tuning

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, how to measure the memory usage of your program and strategies for lowering memory usage if garbage collector performance is a bottleneck.

Garbage collector world view

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, further work to reduce STW times, now down to the 100 microsecond range.
- Go 1.9, 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=:8080
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.

Exercise: Can you name examples in the std lib which follow this pattern?

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.

Using []byte as a map key

It is very common to use a `string` as a map key, but often you have a `[]byte`.

The compiler implements a specific optimisation for this case

```
var m map[string]string
v, ok := m[string(bytes)]
```

This will avoid the conversion of the byte slice to a string for the map lookup. This is very specific, it won't work if you do something like

```
key := string(bytes)
val, ok := m[key]
```

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`

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.

Personal opinion: `sync.Pool` is hard to use safely. Don't use `sync.Pool`.

Exercises

- Using godoc (or another program) observe the results of changing `GOGC` using `GODEBUG=gctrace=1`.
- Benchmark byte's string(byte) map keys
- Benchmark allocs from different concat strategies.

Concurrency

Concurrency

Go's signature feature is its lightweight concurrency model.

While cheap, these features are not free, and their overuse often leads to unexpected performance problems.

This final section concludes with a set of do's and don't's for efficient use of Go's concurrency primitives.

Goroutines

The key feature of Go that makes it a great fit for modern hardware are goroutines.

Goroutines are so easy to use, and so cheap to create, you could think of them as *almost* free.

The Go runtime has been written for programs with tens of thousands of goroutines as the norm, hundreds of thousands are not unexpected.

However, each goroutine does consume a minimum amount of memory for the goroutine's stack which is currently at least 2k.

2048 * 1,000,000 goroutines == 2GB of memory, and they haven't done anything yet.

Know when to stop a goroutine

Goroutines are cheap to start and cheap to run, but they do have a finite cost in terms of memory footprint; you cannot create an infinite number of them.

Every time you use the go keyword in your program to launch a goroutine, you must **know** how, and when, that goroutine will exit.

If you don't know the answer, that's a potential memory leak.

In your design, some goroutines may run until the program exits. These goroutines are rare enough to not become an exception to the rule.

Never start a goroutine without knowing how it will stop.

Go uses efficient network polling for some requests

The Go runtime handles network IO using an efficient operating system polling mechanism (kqueue, epoll, windows IOCP, etc). Many waiting goroutines will be serviced by a single operating system thread.

However, for local file IO, Go does not implement any IO polling. Each operation on a `*os.File` consumes one operating system thread while in progress.

Heavy use of local file IO can cause your program to spawn hundreds or thousands of threads; possibly more than your operating system allows.

Your disk subsystem does not expect to be able to handle hundreds or thousands of concurrent IO requests.

io.Reader and io.Writer are not buffered

`io.Reader` and `io.Writer` implementations are not buffered.

This includes `net.Conn` and `os.Stdout`.

Use `bufio.NewReader(r)` and `bufio.NewWriter(w)` to get a buffered reader and writer.

Don't forget to `Flush` or `Close` your buffered writers to flush the buffer to the underlying `Writer`.

Watch out for IO multipliers in your application

If you're writing a server process, its primary job is to multiplex clients connected over the network, and data stored in your application.

Most server programs take a request, do some processing, then return a result. This sounds simple, but depending on the result it can let the client consume a large (possibly unbounded) amount of resources on your server. Here are some things to pay attention to:

- The amount of IO requests per incoming request; how many IO events does a single client request generate? It might be on average 1, or possibly less than one if many requests are served out of a cache.
- The amount of reads required to service a query; is it fixed, N+1, or linear (reading the whole table to generate the last page of results).

If memory is slow, relatively speaking, then IO is so slow that you should avoid doing it at all costs. Most importantly avoid doing IO in the context of a request—don't make the user wait for your disk subsystem to write to disk, or even read.

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.

Timeouts, timeouts, timeouts

Never start an IO operating without knowing the maximum time it will take.

You need to set a timeout on every network request you make with `SetDeadline`, `SetReadDeadline`, `SetWriteDeadline`.

You need to limit the amount of blocking IO you issue. Use a pool of worker goroutines, or a buffered channel as a semaphore.

```
var semaphore = make(chan struct{}, 10)

func processRequest(work *Work) {
    semaphore <- struct{}{} // acquire semaphore
    // process request
    <-semaphore // release semaphore
}
```

Defer is expensive, or is it?

`defer` is expensive because it has to record a closure for defer's arguments.

```
defer mu.Unlock()
```

is equivalent to

```
defer func() {
        mu.Unlock()
}()
```

`defer` is expensive if the work being done is small, the classic example is `defer` ing a mutex unlock around a struct variable or map lookup. You may choose to avoid `defer` in those situations.

This is a case where readability and maintenance is sacrificed for a performance win.

Always revisit these decisions.

> github.com/golang/go/issues/9704#issuecomment-251003577 (https://github.com/golang/go/issues/9704#issuecomment-251003577)

Minimise cgo

cgo allows Go programs to call into C libraries.

C code and Go code live in two different universes, cgo traverses the boundary between them.

This transition is not free and depending on where it exists in your code, the cost could be substantial.

cgo calls are similar to blocking IO, they consume a thread during operation.

Do not call out to C code in the middle of a tight loop.

Actually, avoid cgo

cgo has a high overhead.

For best performance I recommend avoiding cgo in your applications.

- If the C code takes a long time, cgo overhead is not as important.
- If you're using cgo to call a very short C function, where the overhead is the most noticeable, rewrite that code in Go -- by definition it's short.
- If you're using a large piece of expensive C code is called in a tight loop, why are you using Go?

Is there anyone who's using cgo to call expensive C code frequently?

Further reading: cgo is not Go. (http://dave.cheney.net/2016/01/18/cgo-is-not-go)

Always use the latest released version of Go

Old versions of Go will never get better. They will never get bug fixes or optimisations.

- Go 1.4 should not be used.
- Go 1.5 and 1.6 had a slower compiler, but it produces faster code, and has a faster GC.
- Go 1.7 delivered roughly a 30% improvement in compilation speed over 1.6, a 2x improvement in linking speed (better than any previous version of Go).
- Go 1.8 will deliver a smaller improvement in compilation speed (at this point), but a significant improvement in code quality for non Intel architectures.

Old version of Go receive no updates. Do not use them. Use the latest and you will get the best performance.

Go 1.7 toolchain improvements (http://dave.cheney.net/2016/04/02/go-1-7-toolchain-improvements)

Go 1.8 performance improvements (http://dave.cheney.net/2016/09/18/go-1-8-performance-improvements-one-month-in)

Discussion

Any questions?

Conclusion

Conclusion

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

Start with the simplest possible code.

Measure.

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.)

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

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)