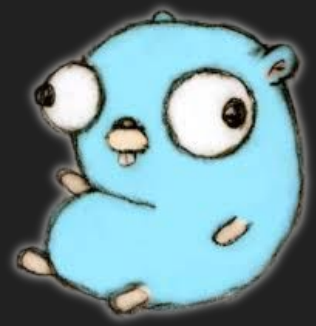
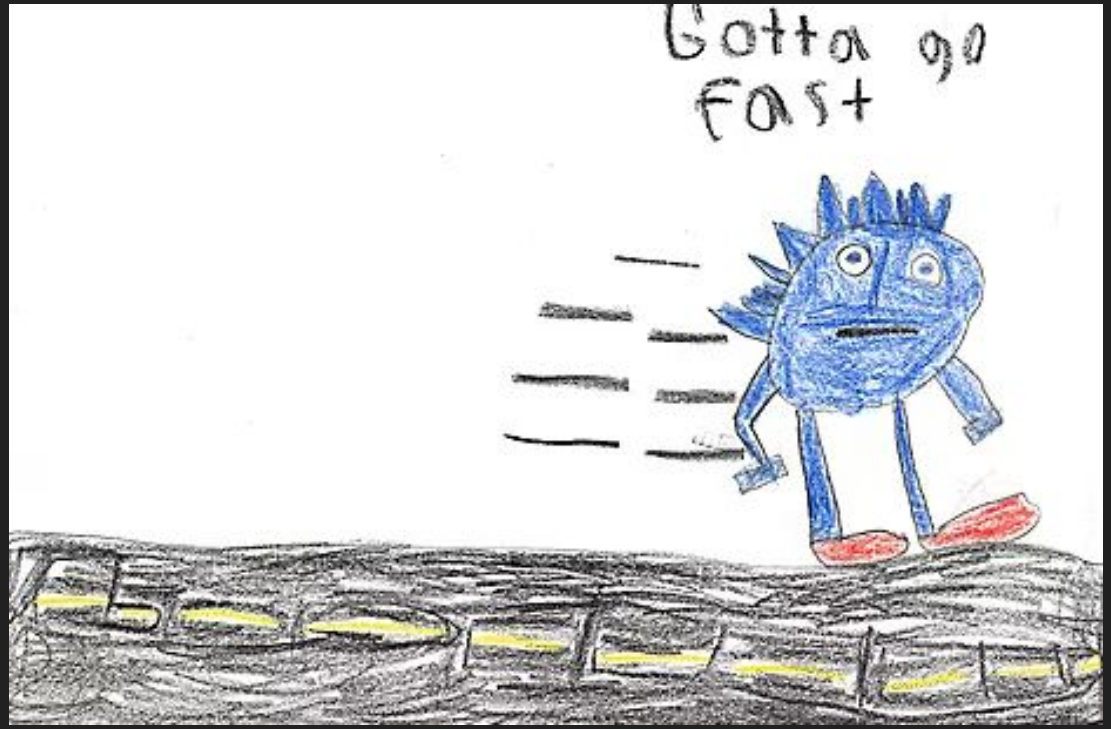
Optimizing Go code without a blindfold

GopherCon San Diego, 2019 - Daniel Martí



I'm here because optimizing is fun.



But wait.

Is your program *slow*?

Do you think it could *go fast*?

Is it *worth* optimizing?

Enter a silly example.

```
func copyList(in []string) []string {
        var out []string
        for _, s := range in {
                out = append(out, s)
        }
        return out
}
```

```
var input = []string{/* … */}

func BenchmarkCopyList(b *testing.B) {
        b.ReportAllocs()
        for i := 0; i < b.N; i++ {
                copyList(input)
        }
}
```

```
$ go test -bench=.
BenchmarkCopyList-8        300000
         5093 ns/op
         16368 B/op
         10 allocs/op
```

```
$ go tool pprof cpu.out
(pprof) list copyList
     :   var out []string
160ms:   for _, s := range in {
5.15s:           out = append(out, s)
     :   }
     :   return out
     :}
```

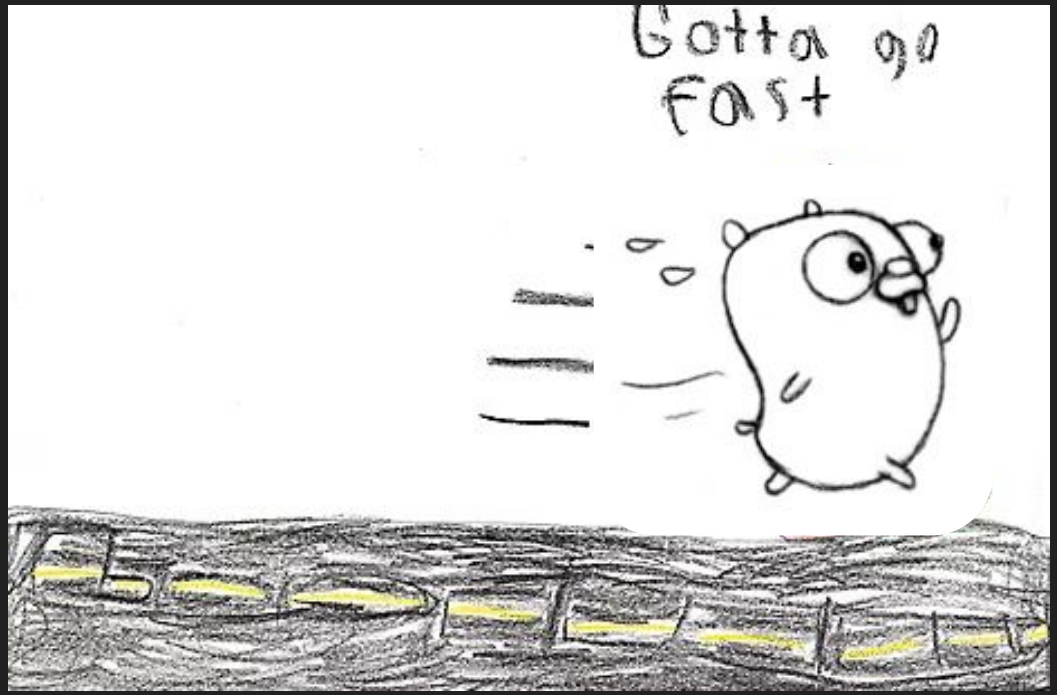
```
func copyList(in []string) []string {
        out := make([]string, len(in))
        for i, s := range in {
                out[i] = s
        }
        return out
}
```

```
$ benchcmp old.txt new.txt
benchmark  old ns/op  new ns/op  delta
CopyList   5093       1490       -70.74%

benchmark  old alloc  new alloc  delta
CopyList   10         1          -90.00%

benchmark  old bytes  new bytes  delta
CopyList   16368      4864       -70.28%
```

And we're done! Thanks for listening.



Hang on.

This **is** a silly example.

Enter a **JSON** benchmark.

```
$ go test -bench=CodeDecoder
BenchmarkCodeDecoder-8          100
10064606 ns/op
         192.80 MB/s
         2696748 B/op
         77484 allocs/op
```

This benchmark is **slow**.

And won't stay **still**.

```
CodeDecoder-8    10052610 ns/op
CodeDecoder-8     9875560 ns/op  -1.76%
CodeDecoder-8    10065054 ns/op  +1.92%
CodeDecoder-8    10468943 ns/op  +4.01%
CodeDecoder-8    10062064 ns/op  -3.89%
```

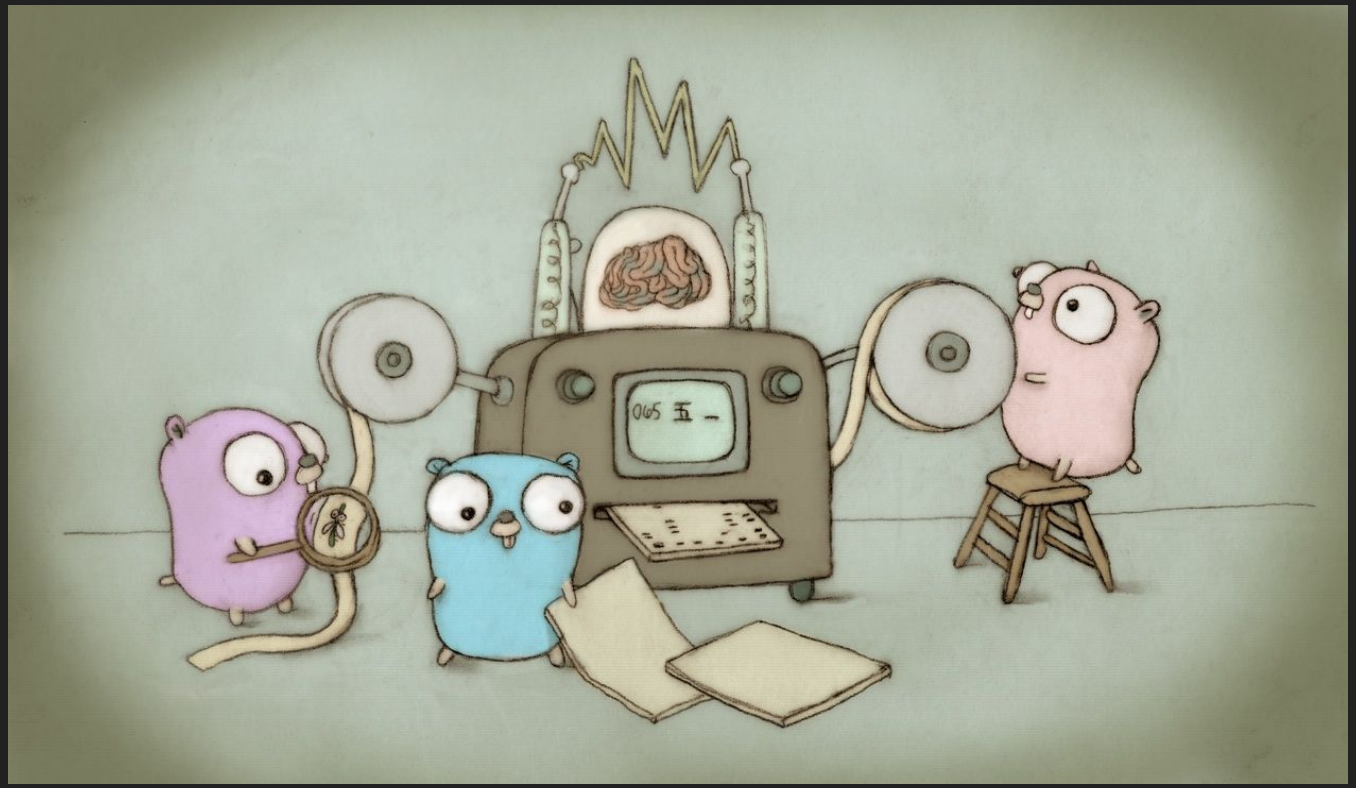
Recent JSON speed-ups:

1.3%, 1.6%, 7.8%, 3.7%

Too much **noise**.

How can we measure **progress**?

Math! Err, statistics.



Get **multiple** samples.

Now we can measure **variance**.

Better benchcmp:

golang.org/x/perf/cmd/**benchstat**

```
$ go test -bench=CodeDecoder -count=8
$ benchstat old.txt
name            time/op
CodeDecoder-8    10.1ms ± 3%

name            speed
CodeDecoder-8   192MB/s ± 3%
```

We need **less** noise.

First: is our machine **idle**?

Introducing my work friends!

CPU usage sits at 0-15%.

Benchmark **demands** 100%.

CPU spike fun: dancing badger

2% CPU use per :badger:

~50 badgers to get the fans spinning

Close resource hungry apps.

CPU usage now at 0-4%.

```
$ go test -bench=CodeDecoder -count=8
$ benchstat old.txt
name            time/op
CodeDecoder-8    10.0ms ± 1%

name            speed
CodeDecoder-8   193MB/s ± 1%
```

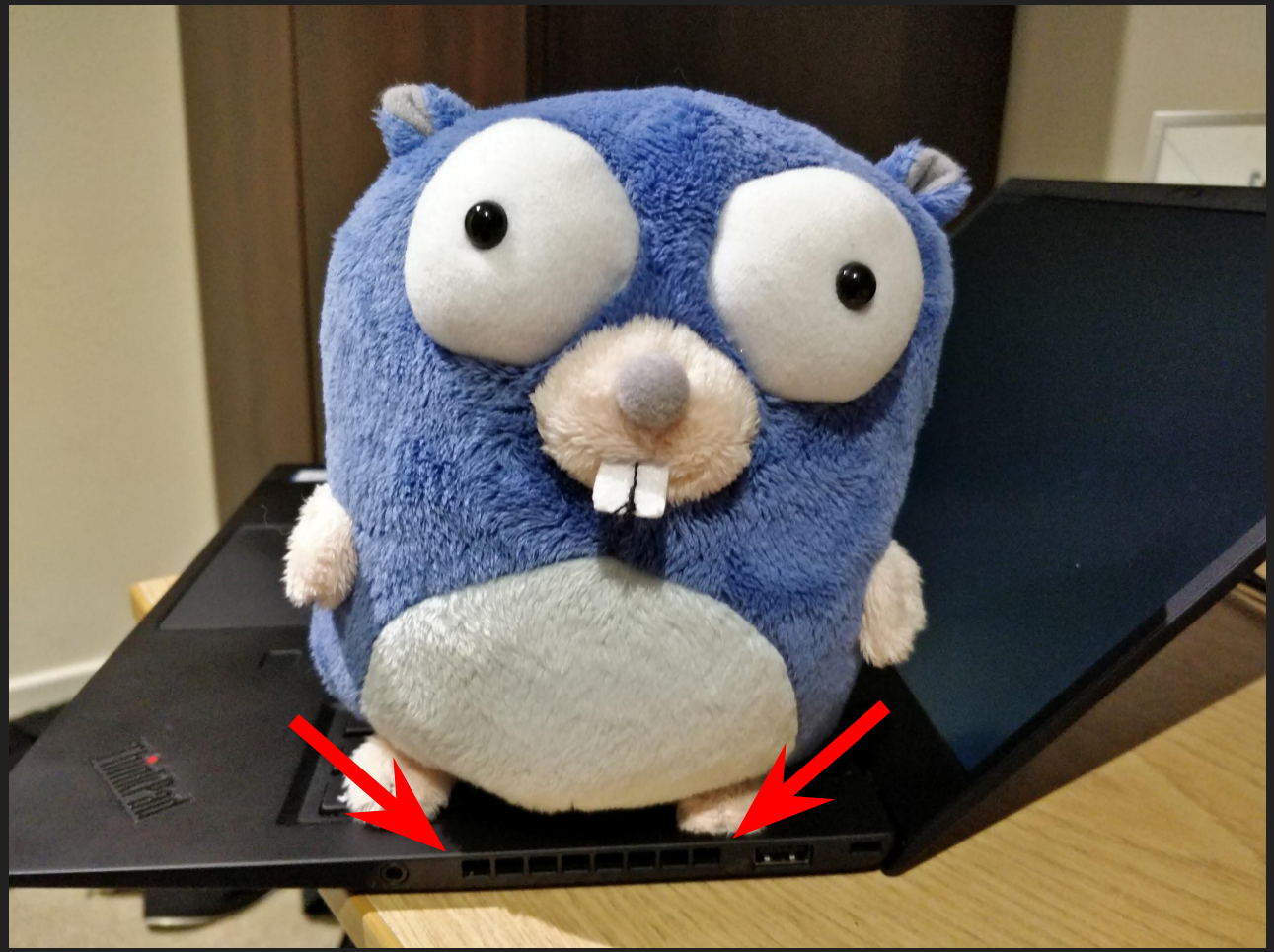
We can work with 1% variance.

However, the CPU **burns**.

```
CodeDecoder-8                10143103 ns/op
CodeDecoder-8                10312887 ns/op
CodeDecoder-8                10233764 ns/op
CodeDecoder-8                10232297 ns/op
[after 9 runs...]
CodeDecoder-8                13774231 ns/op
CodeDecoder-8                14155513 ns/op
CodeDecoder-8                13526806 ns/op
CodeDecoder-8                13755990 ns/op
```

Laptops **throttle**.

And I have some **evidence**.



4 cores with turbo @ 3.4GHz.

Air vents measuring 2 gopher feet.


THIS IS FINE

We **cannot** use turbo speeds.

More tooling!

github.com/aclements/**perflock**

```
$ perflock -daemon &
$ perlock -governor=70% go test -...
CodeDecoder-8                13433640 ns/op
CodeDecoder-8                13508700 ns/op
[after 20 runs...]
CodeDecoder-8                13443626 ns/op
CodeDecoder-8                13263873 ns/op
```

Caveat: only for Linux.

Should be portable to Mac/Win.

```
$ go test -count=8 … > old.txt
$ go test -count=8 … > new.txt
$ benchstat old.txt new.txt
old time/op    new time/op
13.5ms ± 1%    13.4ms ± 1%

       delta
       ~     (p=0.247 n=10+10)
```

What -count to use?

What's a "p-value"?

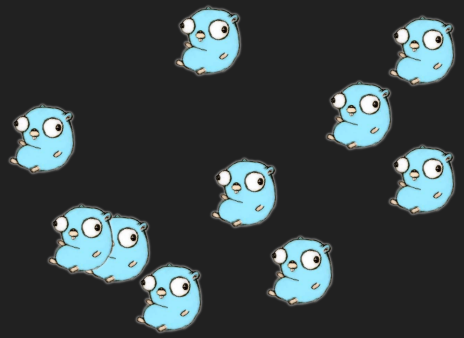
Higher variance -> Higher -count

Now with visual aids!

N=3 gopher data points

N=10 gopher data points; lower p-value

>



Gotcha: Don't **search** for p-values.

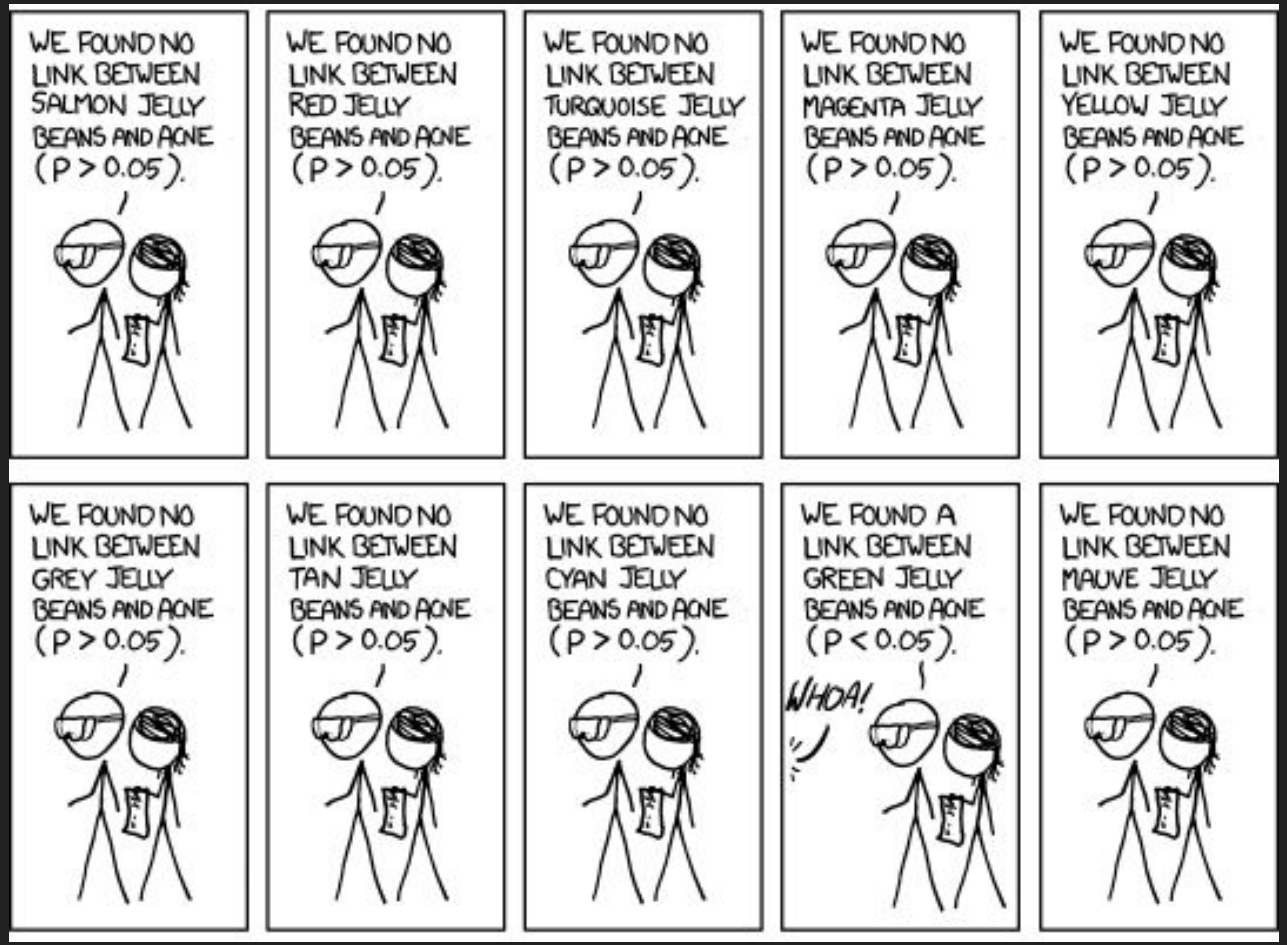
Multiple testing problem!

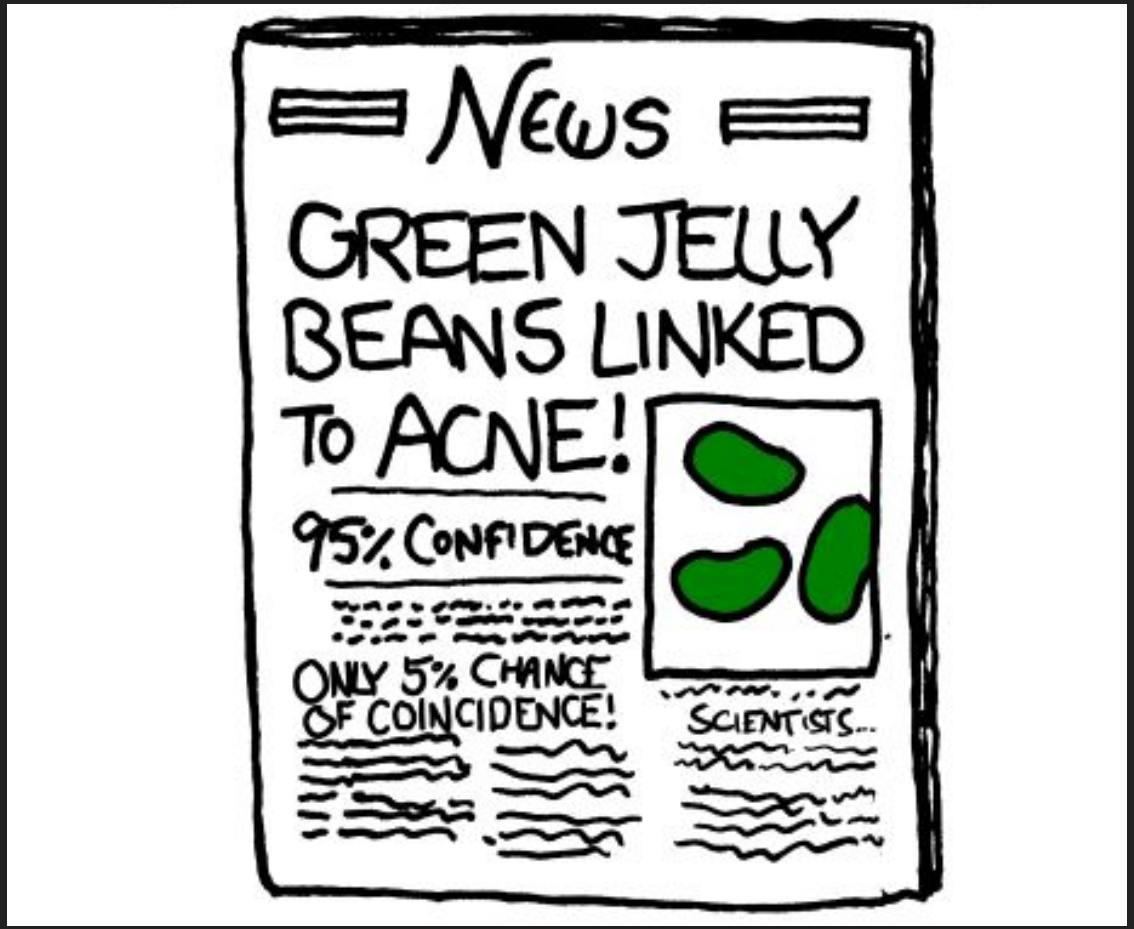
```
CodeDecoder-8    10052610 ns/op
CodeDecoder-8     9875560 ns/op  -1.76%
CodeDecoder-8    10065054 ns/op  +1.92%
CodeDecoder-8    10468943 ns/op  +4.01%
CodeDecoder-8    10062064 ns/op  -3.89%
```

```
new time/op    delta
13.4ms ± 1%      ~       (p=0.343 n=10+10)
13.3ms ± 2%      ~       (p=0.343 n=10+10)
13.3ms ± 1%      ~       (p=0.886 n=10+10)
13.5ms ± 2%      ~       (p=0.114 n=10+10)
13.3ms ± 0%    -1.39%    (p=0.041 n=10+10)
```

If the data looks bad...

don't get **new** data!


WE FOUND NO LINK BETWEEN SALMON JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN RED JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TURQUOISE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN MAGENTA JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN YELLOW JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN GREY JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TAN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN CYAN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND A LINK BETWEEN GREEN JELLY BEANS AND ACNE (P < 0.05).
WHOA!
WE FOUND NO LINK BETWEEN MAUVE JELLY BEANS AND ACNE (P > 0.05).


NEWS
GREEN JELLY BEANS LINKED TO ACNE!
95% CONFIDENCE
ONLY 5% CHANCE OF COINCIDENCE!
SCIENTISTS...

Sidenote: **bottlenecks**.

Some tools are for CPU loads.

pprof, perflock.

Statistics work for **all** benchmarks.

Recap:

benchstat to compare statistics.
perflock to avoid noise.

Compiler **tricks**!

```
$ go build -gcflags='-m -m' io
io.go:371: cannot inline CopyBuffer:
    function too complex:
    cost 84 exceeds budget 80

$ … | grep 'function too complex'
```

```
$ go build -gcflags='-m -m' io
io.go:293: ([]byte)(s) escapes to heap

$ … | grep 'escapes to heap'
```

```
$ … -gcflags=-d=ssa/check_bce/debug=1 io
io.go:310: Found IsSliceInBounds
multi.go:30: Found IsInBounds
```

```
$ … -gcflags=-d=ssa/prove/debug=1 io
io.go:446: Proved IsSliceInBounds
multi.go:21: Proved IsInBounds

$ … -gcflags=-d=ssa/prove/debug=2 io
multi.go:59: x+d >= w; x:v24 b6 delta:1 …
```

Gotcha: code suddenly gets slower.

Or faster?

```
Author: Russ Cox <rsc@golang.org>
Date:   Wed Nov 16 19:18:25 2011

json.BenchmarkSkipValue     -24.66%

I cannot explain why BenchmarkSkipValue
gets faster. Maybe it is one of those
code alignment things.
```

The compiler is getting **better**!

```
# replace a map
m = make(map[string]string)

# clear a map; faster since Go 1.11!
for k := range m {
      delete(m, k)
}
```

```
# count manually
n := 0
for range str {
      n++
}

# simple, and fast since Go 1.11!
n := len([]rune(str))
```

Give the compiler a chance.

If it could do better, **file bugs**.

We use the **Performance** label on the issue tracker.

To go deeper...

GOSSAFUNC=pattern go build

```
$ cat f.go
package p
func HelloWorld() {
        println("hello, world!")
}
$ GOSSAFUNC=HelloWorld go build
dumped SSA to ssa.html
$ chromium ssa.html
```

start

```
b1:
v1 (?) = InitMem <mem>
v2 (?) = SP <uintptr>
v3 (?) = SB <uintptr>
v4 (4) = StaticCall <mem> {runtime.printlock} v1
v5 (?) = OffPtr <*string> [0] v2
v6 (?) = ConstString <string> {"hello, world!\n"}
v7 (4) = Store <mem> {string} v5 v6 v4
v8 (4) = StaticCall <mem> {runtime.printstring} [16] v7
v9 (4) = StaticCall <mem> {runtime.printunlock} v8
Ret v9 (5)
```

genssa

```
    00000 (3) TEXT      "".HelloWorld(SB), ABIInternal
[...]
v4  00006 (4) CALL      runtime.printlock(SB)
v12 00007 (4) PCDATA    $2, $1
v12 00008 (4) LEAQ      go.string."hello, world!\n"(SB), AX
v13 00009 (4) PCDATA    $2, $0
v13 00010 (4) MOVQ      AX, (SP)
v7  00011 (4) MOVQ      $14, 8(SP)
v8  00012 (4) CALL      runtime.printstring(SB)
v9  00013 (4) CALL      runtime.printunlock(SB)
b1  00014 (+5) RET
    00015 (?) END
```

cmd/compile/README

cmd/compile/internal/ssa/README

Benchmarking demo!

What could go wrong?

Use these tools to optimize responsibly.

twitter.com/mvdan_