profile-guided code analysis

2022 / @quasilyte

Agenda

- Some facts about Go CPU profiles and profiling
- Go profiles parsing
- Custom profiles data aggregation
- Heatmaps intro and why we need them
- Structural code search with heatmap filters
- Profile-guided performance static analysis (PGO)
- Some pprof insights

Why does this talk exists?

A complex, big system

We want to make it faster.

But sometimes there are no obvious “bottlenecks”.

A system as a whole is slow.

There are hundreds of small performance issues.

Let’s state our goal clearly

We’re interested in making the entire system faster

We **don’t** want to change the code in “cold” paths

I **don’t** care about FooBar benchmark running 100 times faster

Our motto is: **less code changes => more performance impact**

We should not optimize blindly

You need to know which parts of your program are executed.

You also need some extra info, like timings.

The CPU profiles provide us this and more.

CPU profiling in Go

CPU profiling facts

- Interruption-based (SIGPROF on unix)
- Sample-based (runtime/pprof records 100 samples/sec)
- Writes the output in pprof format (profile.proto)

What makes a good CPU profile

- Collected for a long time (longer than a few seconds!)
- Collected under an interesting and realistic load
- Aligned with your task*

It's also nice to have several CPU profiles, collected in different configurations.

(*) If you're optimizing a single function, CPU profiles from benchmarks are OK. Otherwise they're not a good fit.

Why CPU profiles from benchmarks are bad?

They can make irrelevant code look “hot”.

They do not show the entire system execution patterns.

Merging CPU profiles from all benchmarks doesn’t help.

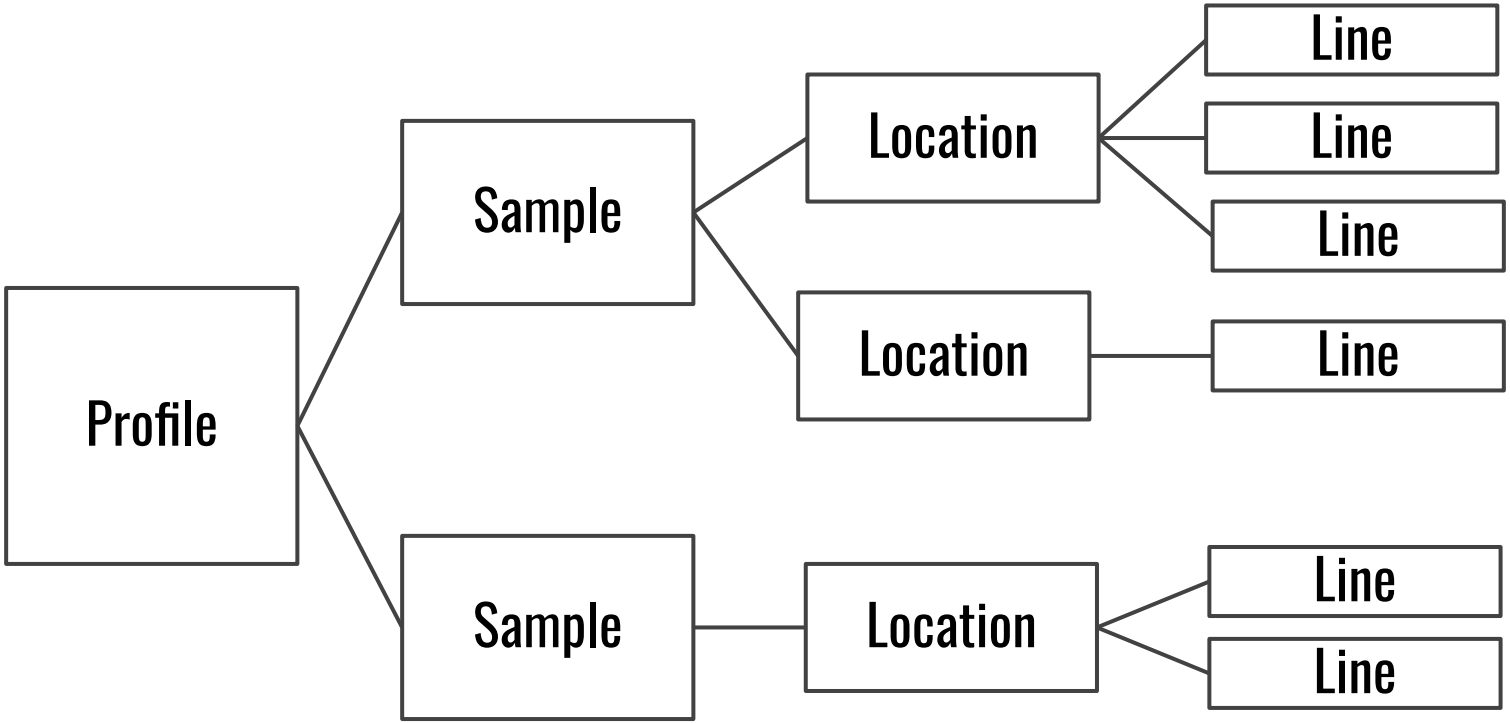
They’re not aligned with our goals.

profile.proto structure

Parsing profile.proto files

The pprof/profile Go library allows you to parse CPU profiles produced by Go.

github.com/google/pprof/profile



Samples layout

```
for _, sample := range p.Samples {
  for _, loc := range sample.Location {
    for _, l := range loc.Line {
      sampleValue := sample.Value[1] // time/ns
      funcName := l.Function.Name
      filename := l.Function.Filename
      line := l.Line
      println(filename, line, funcName, sampleValue)
    }
  }
}
```

Visiting profile samples: a straightforward approach

```
var stack []profile.Line
for _, sample := range p.Samples {
  stack = stack[:0] // reuse memory for every stack
  for _, loc := range sample.Location {
    stack = append(stack, loc.Line...)
  }
  for i, l := range stack {
    // handle l
  }
}
```

Visiting profile samples: a smarter approach

```
var stack []profile.Line
for _, sample := range p.Samples {
  stack = stack[:0]
  for _, loc := range sample.Location {
    stack = append(stack, loc.Line...)
  }
}
```

stack[0]

“self” sample, current function

Visiting profile samples: a smarter approach

```
var stack []profile.Line
for _, sample := range p.Samples {
  stack = stack[:0]
  for _, loc := range sample.Location {
    stack = append(stack, loc.Line...)
  }
}
```

stack[1:]

Callers stack

Visiting profile samples: a smarter approach

```
* somefunc
* runtime.mallocgc
* github.com/foo/pkg.Bar
* github.com/foo/pkg.Bar.func1
* github.com/foo/pkg.(*Bar).Method
```

Function.Name parsing

* somefunc
* runtime.mallocgc
* github.com/foo/pkg.Bar
* **github.com/foo/pkg.Bar.func1**
* github.com/foo/pkg.(*Bar).Method

Some symbols are ambiguous!
* Could be a Bar method named “func1”
* Could be a lambda “func1” inside “Bar” function

Use https://github.com/quasilyte/pprofutil

Function.Name parsing

Aggregation: calculating “flat”

Map **`file:line:func`** keys to sampleValue,

but use only **`stack[0]`** records.

Aggregation: calculating “cumulative” (e.g. “cum”)

Map **file:line:func** keys to sampleValue,

using all records from **stack** (all lines).

Runtime-cumulative aggregation

Does this look familiar to you?

```
(pprof) top 5
9.80% runtime.findObject
8.40% runtime.scanobject
3.47% runtime.mallocgc
3.42% runtime.heapBitsSetType
2.87% runtime.markBits.isMarked
```

So, the Go runtime is slow?

All top nodes are showing that most of the time is spent in the runtime.

Does this mean that our app itself is fast, but Go runtime is a bottleneck?

```
func copyBytes(b []byte) []byte {
  dst := make([]byte, len(b))
  copy(dst, b)
  return dst
}
```

copyBytes definition

```
func BenchmarkCopyBytes(b *testing.B) {
  dst := make([]byte, 2022)
  for i := 0; i < b.N; i++ {
    copyBytes(dst)
  }
}
```

```
go test -bench=. -cpuprofile=cpu.out
```

Benchmarking copyBytes

Where is copyBytes?

```
(pprof) top 5
37.56% runtime.mallocgc
12.16% runtime.memclrNoHeapPointers
 9.35% runtime.memmove
 8.47% runtime.scanobject
 6.42% runtime.scanblock
```

```
func copyBytes(b []byte) []byte {
  dst := make([]byte, len(b))
  copy(dst, b)
  return dst
}
```

- **mallocgc** (allocating a slice)
- **memclrNoHeapPointers** (memory zeroing)
- **memmove** (copying memory)

User-code example

Aggregation: runtime-cumulative aggregation

Map **file:line:func** keys to sampleValue,

like in a normal flat or cumulative schemes,

but for every runtime **stack[0]** add this value to

a first non-runtime caller.

Runtime-cumulative aggregation results

```
./runtime-cumulative cpu.out

6.25s example.copyBytes
 40ms example.BenchmarkCopyBytes
 20ms testing.(*B).runN
```

So, the Go runtime is slow?

~~All top nodes are showing that most of the time is spent in the runtime.~~

~~Does this mean that our app itself is fast, but Go runtime is a bottleneck?~~

No, we just need to aggregate the CPU data correctly.

Heatmaps (and text editors)

Idea: display how “hot” the source code line is

It would be great to see the performance-sensitive parts of our code right in our text editor.

We’ll use **heat levels** to categories the hotness.

Heatmaps: building a line-oriented index from a CPU profile

We can build a simple index that aggregates all samples from the CPU profile and splits them into categories (heat levels).

Then we can tell what is the heat level of the given source code line.

```
file.go:100 0.1s
file.go:100 0.3s
file.go:120 0.1s
file.go:120 0.1s
file.go:100 0.4s
file.go:130 0.2s
file.go:140 0.1s
file.go:145 0.1s
file.go:150 0.3s
file.go:165 0.2s
file.go:170 0.2s
```

Take all samples for a file

Building a heatmap

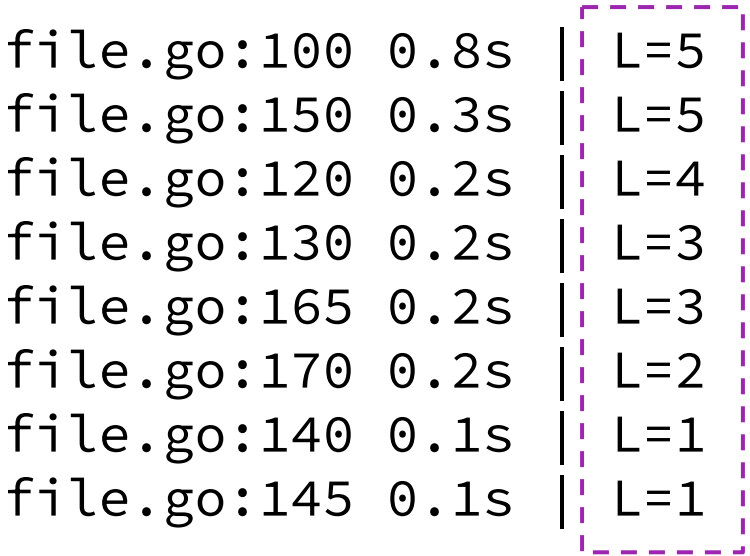
```
file.go:100 0.8s
file.go:120 0.2s
file.go:130 0.2s
file.go:140 0.1s
file.go:145 0.1s
file.go:150 0.3s
file.go:165 0.2s
file.go:170 0.2s
```

Combine sample values for the same lines

```
file.go:100 0.8s
file.go:150 0.3s
file.go:120 0.2s
file.go:130 0.2s
file.go:165 0.2s
file.go:170 0.2s
file.go:140 0.1s
file.go:145 0.1s
```

Sort samples by their value

Building a heatmap



Divide them into categories (heat levels)

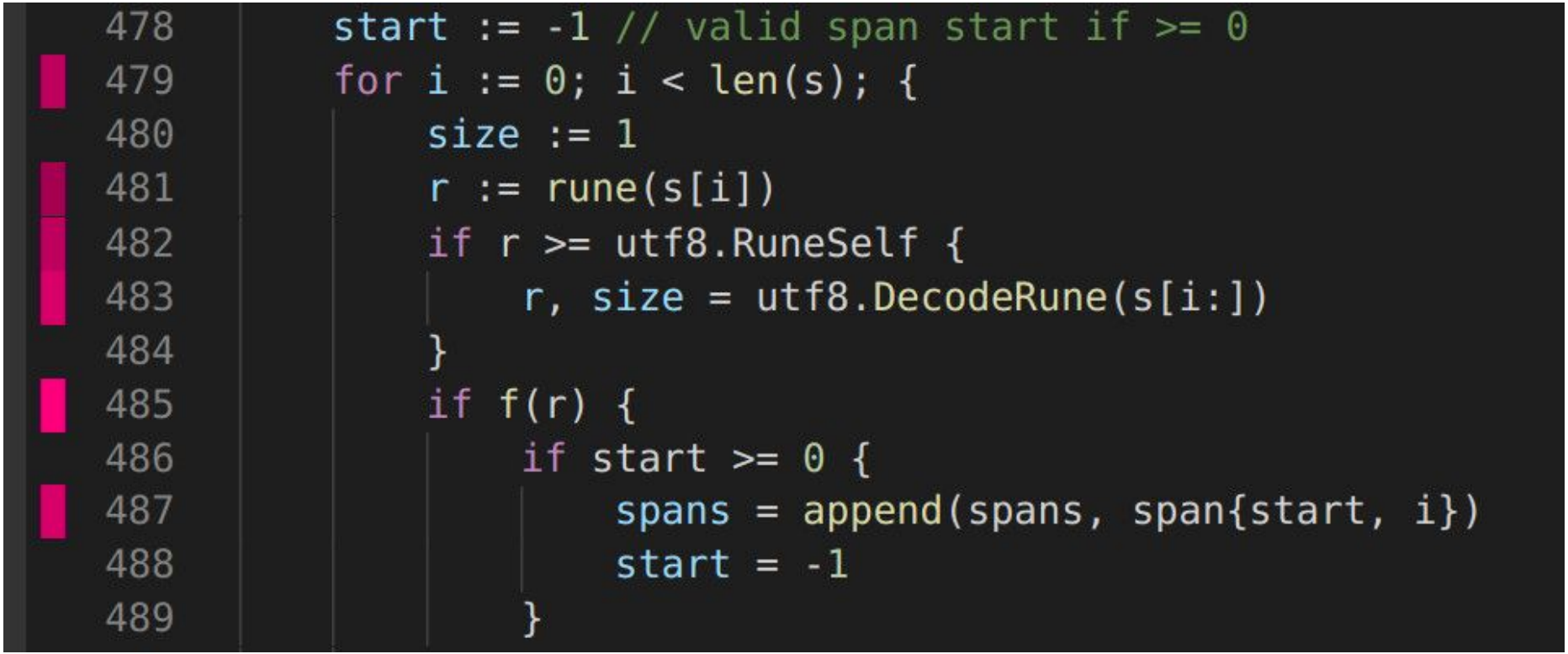
Building a heatmap

```
file.go:100 0.8s | L=5
file.go:150 0.3s | L=4
file.go:120 0.2s | L=3
file.go:130 0.2s | L=2
file.go:165 0.2s | L=0
file.go:170 0.2s | L=0
file.go:140 0.1s | L=0
file.go:145 0.1s | L=0
```

A threshold can control the % of samples we're using (top%)

Let's use **threshold=0.5**

Building a heatmap



https://github.com/quasilyte/vscode-perf-heatmap

perf-heatmap library

I created a library that can be used to build a heatmap index from profile.proto CPU profiles.

It’s used in all profile-guided tools presented today.

https://github.com/quasilyte/perf-heatmap

perf-heatmap index properties

- Fast line (or line range) querying
- Relatively compact, low memory usage
- Has both flat and cumulative values
- Only one simple config option: threshold value
- Reliable symbol mapping*

(*) It can match the location even if its absolute path differs in profile and local machine.

gogrep + heatmap

Structural code search

- Intellij IDE - structural code search and replace (SSR)
- go-ruleguard static analyzer (**gogrep lib**)
- go-critic static analyzer (**gogrep lib**)
- gocorpus queries (**gogrep lib**)
- gogrep code search tool (**gogrep lib**)

Try gogrep - it's simple and very useful.

`reflect.TypeOf($x).Size()`

This pattern finds all reflect.TypeOf() calls that are followed by a Size() call on its result.

`$x` is a wildcard, it'll match any expression.

Let's try looking for some patterns!

```
$ gogrep . 'reflect.TypeOf($x).Size()'

src/foo.go:20: strSize := int(reflect.TypeOf("").Size()
src/lib/bar.go:43: return reflect.TypeOf(pair).Size(), nil
…
+ 15 matches
```

Should we rewrite all these 17 cases?

Let’s try looking for some patterns!

```
$ gogrep . --heatmap cpu.out 'reflect.TypeOf($x).Size()' '$$.IsHot()'
```

--heatmap cpu.out

A CPU-profile that will be used to build a heatmap

gogrep + heatmap filter

```
$ gogrep . --heatmap cpu.out 'reflect.TypeOf($x).Size()' '$$.IsHot()'
```

$$.IsHot()

A filter expression.

$$ references the entire match, like $0.

IsHot() applies a heatmap filter.

Only one match now!

```
reflect.TypeOf(value.Elem().Interface()).Size()
```

Only one match now!

```
reflect.TypeOf(value.Elem().Interface()).Size()
```

=>

```
value.Elem().Type().Size()
```

5 times faster (-80%), 0 allocations

Hard to find using CPU profiles!

```
reflect.TypeOf(value.Elem().Interface()).Size()
```

Calls involved:

reflect.Value.Elem()
reflect.Value.Interface()
reflect.TypeOf()
reflect.Type.Size()

+ all functions called from them

Useful resources: structural code search

- gogrep intro: RU, EN (old CLI interface)
- Profile-guided code search articles: RU, EN
- quasilyte.dev/gocorpus Go corpus with gogrep queries

https://github.com/quasilyte/gogrep

ruleguard + heatmap = ?

What is go-ruleguard?

What is go-perfguard?

perfguard: profile-guided Go optimizer

- Works on the source code level
- Has two main modes: lint and optimize
- Finds performance issues in Go code
- Most issues reported have autofixes

perfguard: lint mode

- Doesn’t require a CPU profile
- Can be used on CI to avoid slow code being merged
- Less precise and powerful than optimize mode

perfguard: optimize mode

- Requires a CPU profile
- Finds only real issues
- Contains checks that are impossible in lint mode

It’s better than an ordinary static analyzer because it uses a CPU profile to get the information about the actual program execution patterns.

```
$ perfguard optimize --heatmap cpu.out ./...
```

`--heatmap cpu.out`

A CPU profile we collected for this application

Running perfguard on our code

```
$ perfguard optimize --heatmap cpu.out ./...
```

./...

Analyzing all packages, recursively

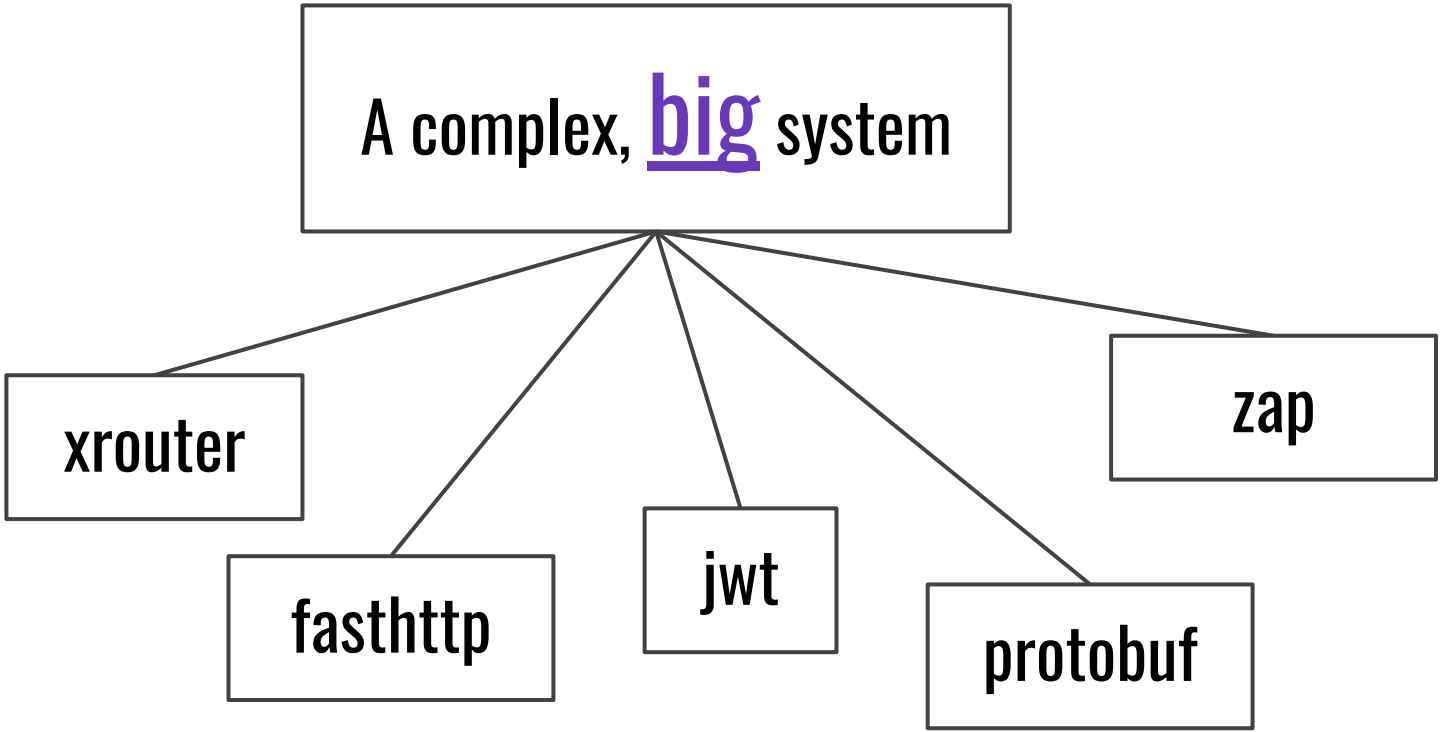
Running perfguard on our code

Analyzing the results

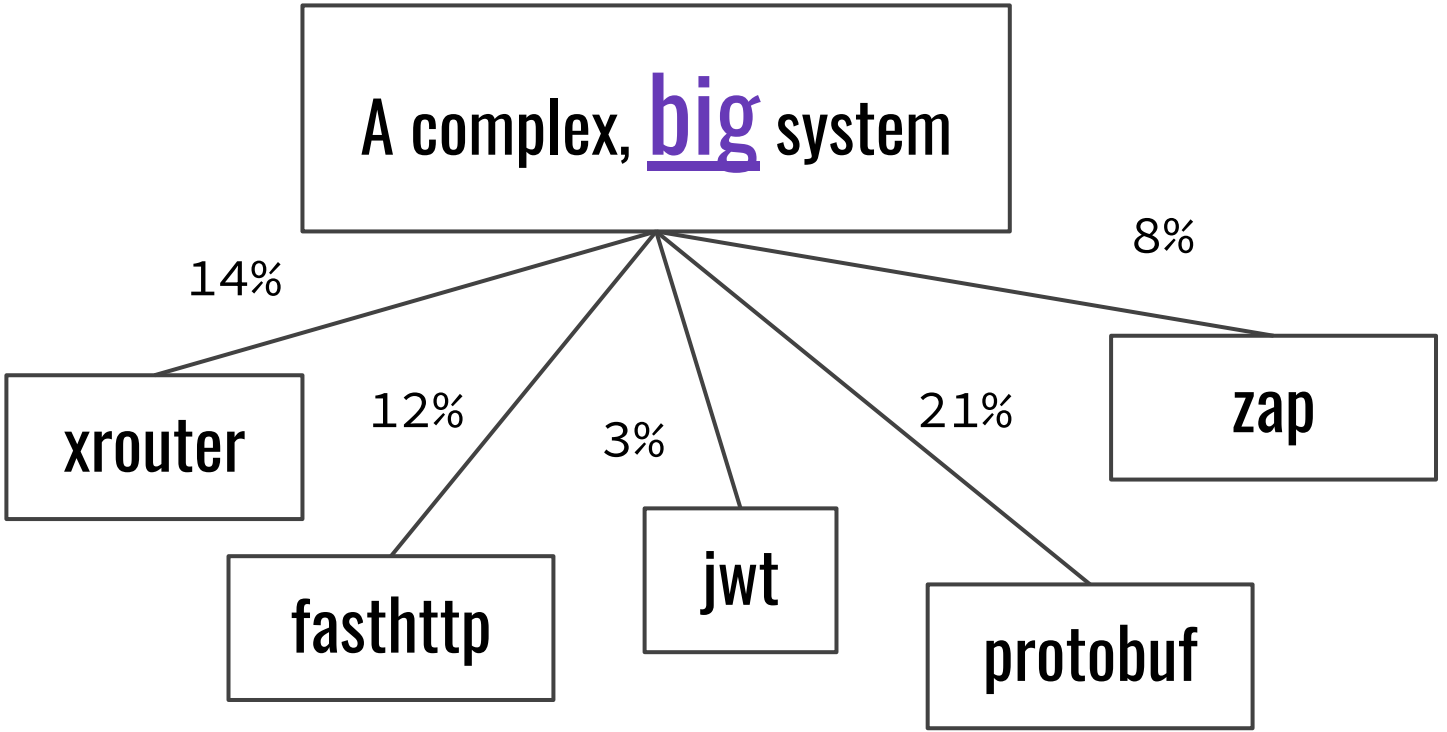
If you get some output, be sure to try out the --fix option to autofix the issues found.

It’s possible to go deeper and analyze the dependencies.

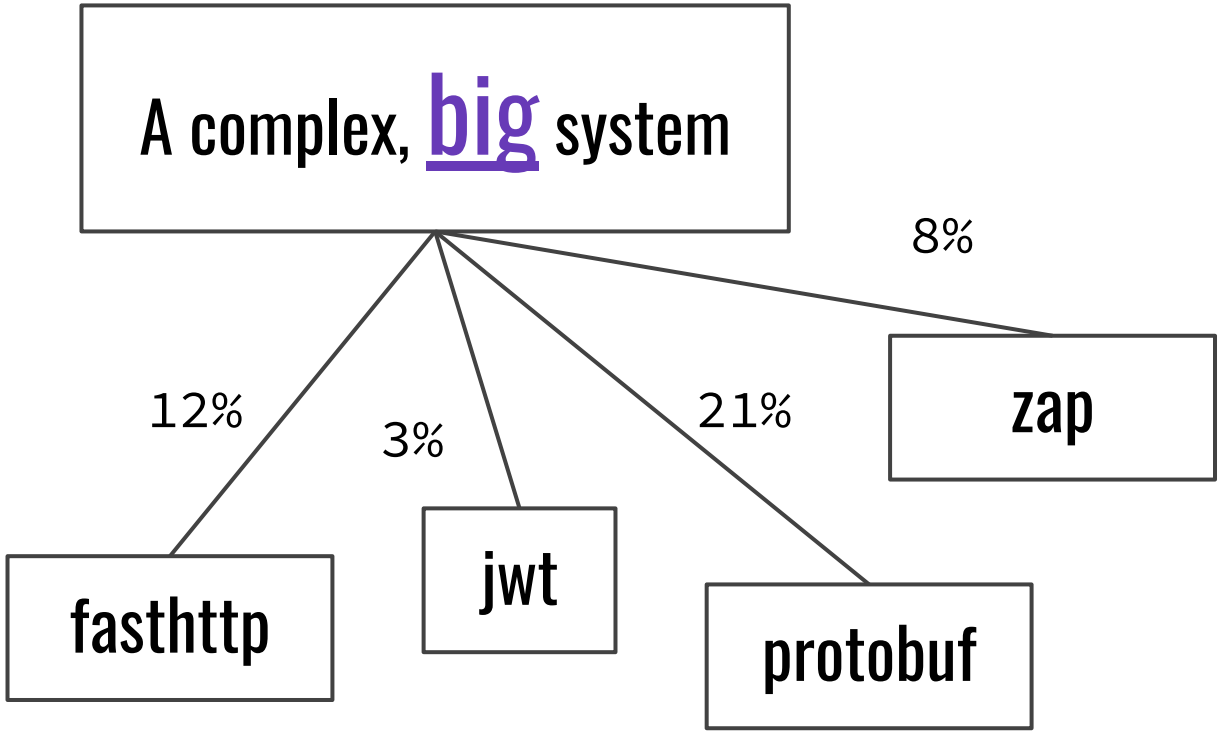
Usually, your code has some dependencies...



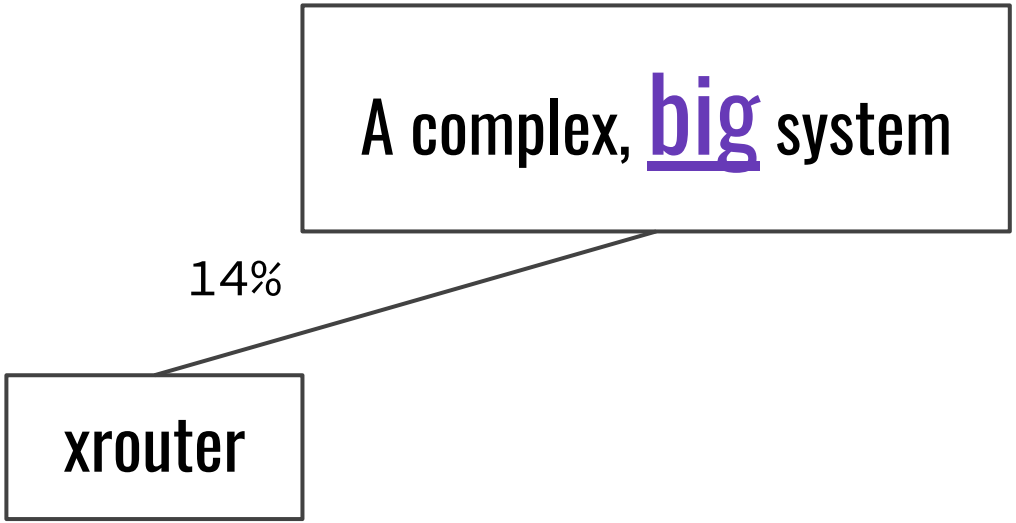
And some of your bottlenecks may live inside them...



Some of them are 3rd party -> hard to fix



But some of them can be under your control



Let’s grab the dependencies for the analysis

Execute this:

```
$ go mod vendor
```

Now we should have a “vendor” folder that contains the sources of all our dependencies.

You can undo this after we finish with analysis.

```
$ perfguard optimize --heatmap cpu.out ./vendor/...
```

--heatmap cpu.out

The same CPU profile we collected and used on our own source code

```
$ perfguard optimize --heatmap cpu.out ./vendor/...
```

./vendor/...

Running the analysis on our dependencies

Running perfguard on dependencies

Analyzing the results

```
vendor/jwt/jwt.go:20: []byte(s)... => s...

vendor/b/b.go:15: bytes.Buffer => strings.Builder

vendor/xrouter/xrouter.go: compiling regexp on hot path

vendor/c/c.go:50: allocating const error, use global var
```

Analyzing the results

```
---

vendor/jwt/jwt.go:20: []byte(s)... => s...

vendor/b/b.go:15: bytes.Buffer => strings.Builder

vendor/xrouter/xrouter.go: compiling regexp on hot path

vendor/c/c.go:50: allocating const error, use global var
```

Now let’s get the xrouter sources

```
$ git clone github.com/quasilyte/xrouter

$ cd xrouter
```

```
$ perfguard optimize --heatmap cpu.out ./...
```

`--heatmap cpu.out`

The very same CPU profile again

```
$ perfguard optimize --heatmap cpu.out ./...
```

./...

Note: running on the xrouter “own” sources

Running perfguard on xrouter

xrouter results

```
vendor/xrouter/xrouter.go: compiling regexp on hot path
```

Only relevant results.

Can also use --fix option to apply autofixes.

Why bother with “go mod vendor”?

Maybe it’s possible for perfguard to figure out where to find the relevant sources in the Go mod pkg dir, but vendoring the sources is easier for now.

This could change in the future, but for now you’ll need to make the dependencies code easily accessible.

What perfguard can fix?

Suggesting one types over the others

```
bytes.Buffer => strings.Builder
```

If local bytes.Buffer is used using the API also covered with strings.Builder and the final result is constructed with String() method, using strings.Builder will result in 1 less allocation.

Alternative API suggestions

```
io.WriteString(w, fmt.Sprintf("%s:%d", f, l))
```

=>

```
fmt.Fprintf(w, "%s:%d", f, l)
```

Removing redundant data copies

```
b = append(b, []byte(s)...) => b = append(b, s...)
copy(b, []byte(s)) => copy(b, s)
re.Match([]byte(s)) => re.MatchString(s)
w.Write([]byte(s)) => w.WriteString(s)
```

And many more transformations that remove excessive data conversions.

Condition reordering

```
f() && isOK => isOk && f()
```

Putting cheaper expressions first in the condition is almost always a win: you can avoid doing unnecessary calls due to a short-circuit nature of logical operators.

map[T]bool -> map[T]struct{}

When local **map[T]bool** is used as a set, perfguard can automatically rewrite it to **map[T]struct{}**, updating all relevant code parts that use it.

Useful resources: ruleguard & perfguard

- ruleguard intro: RU, EN
- ruleguard by example tour
- ruleguard comparison with Semgrep and CodeQL

https://github.com/quasilyte/go-perfguard
https://github.com/quasilyte/go-ruleguard

Final thoughts

- Heatmaps do not replace pprof/flamegraphs/etc
- Filtering results depend on the CPU profile quality
- perfguard can’t magically solve all of your problems

profile-guided code analysis

2022 / @quasilyte