Gopher
in performance tales

Mateusz Szczyrzyca
mateusz@szczyrzyca.pl
https://devopsiarz.pl

picture from: https://golangtraining-in-web.appspot.com/

It’s about…

- **Performance in general**
- **Some important basics**
- **Interesting performance case studies**
- **From the Gopher world’s perspective:**
  - general basics and tips
  - pprof & tracer
  - recommendations

Preface

Things are not that simple as they look like.
Especially numbers.

Trade offs are always a part of software performance engineering

Performance (Software)

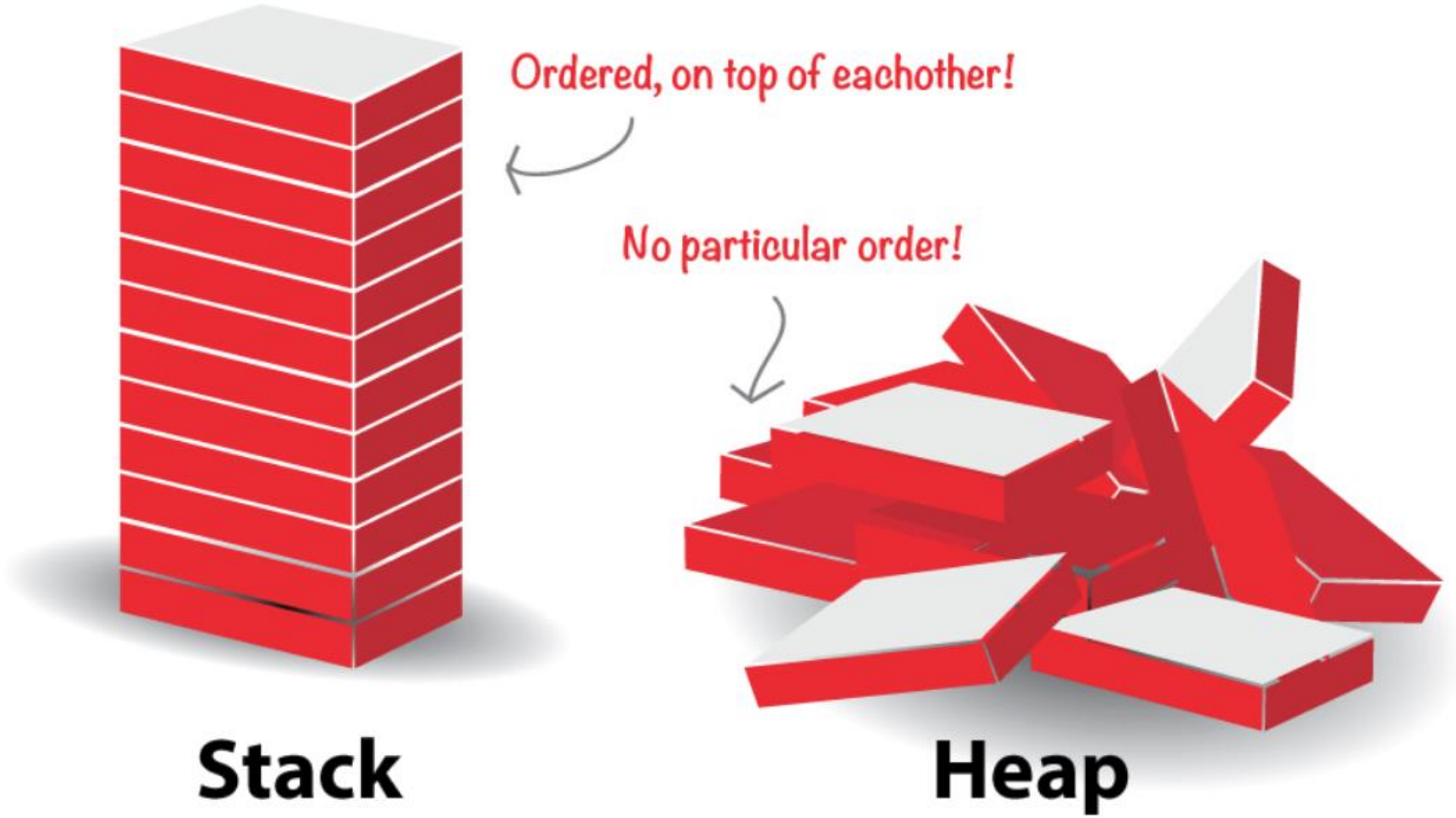
How available hardware resources are utilized by applications

app ability to operate under certain conditions (low hardware resources, big amount of traffic, etc)

Basic terms

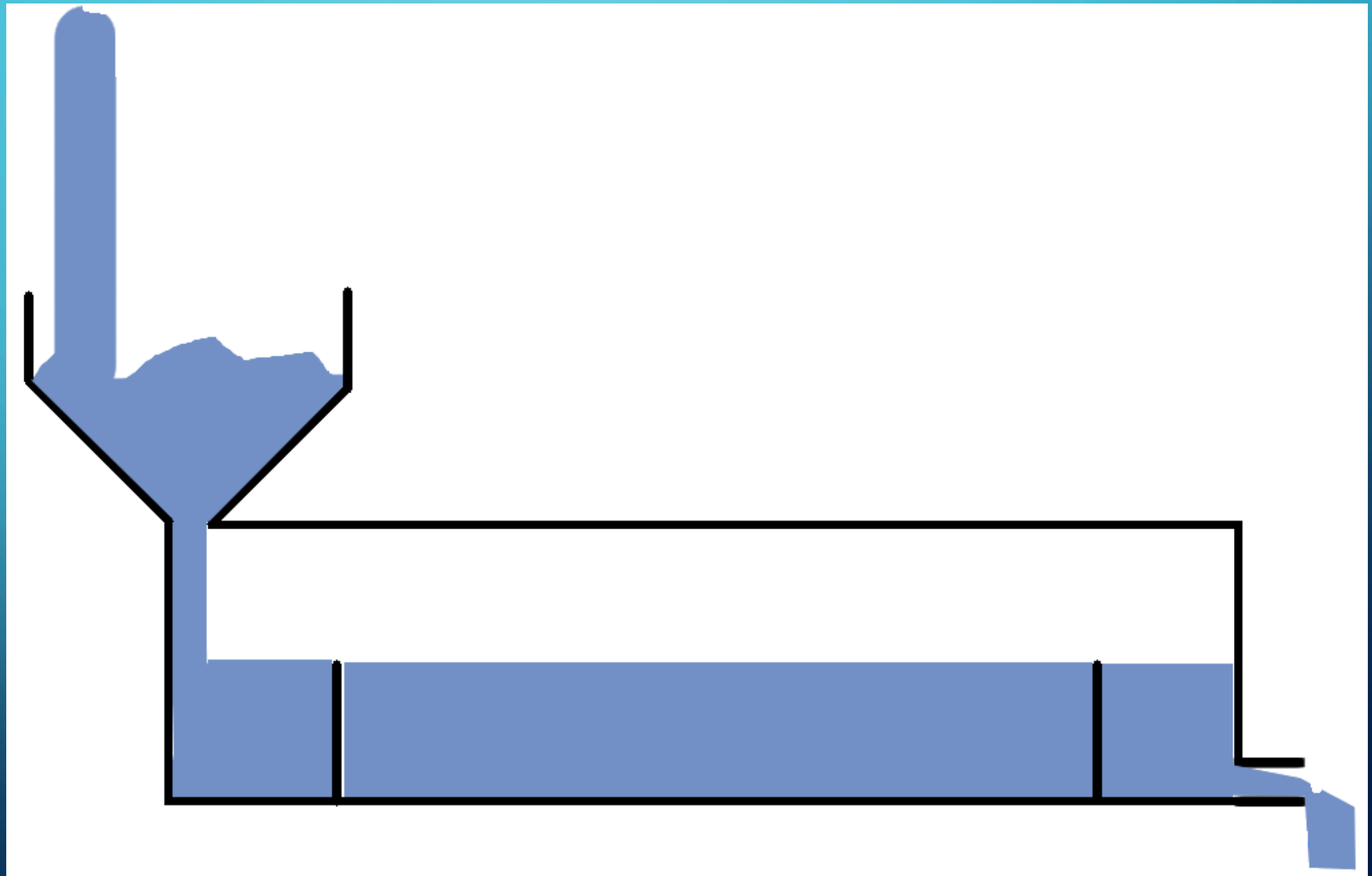
- algorithm
- runtime / compile time
- stack & heap
- GC (garbage collector)
- real/user/sys time
- Big-O notation
- concurrency (multithreading)
- parallelism
- IOPs
- Throughput, Latency, Response time
- utilization
- saturation
- bottleneck
- Workload

Stack vs Heap

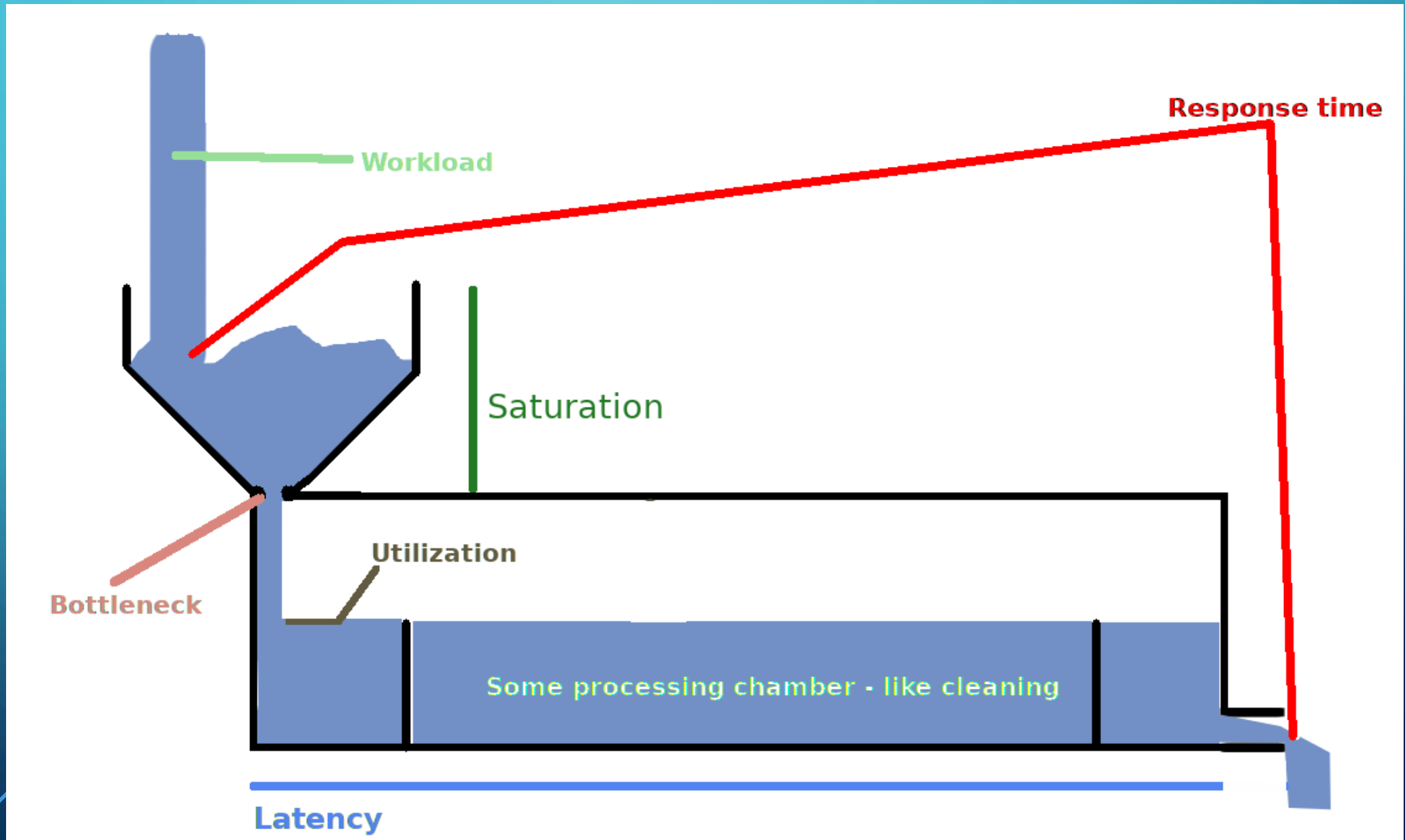


source: https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

Throughput, Latency, Response Time, Saturation



Throughput, Latency, Response Time, Saturation



Root of all evil

Premature optimizations

When you lose your time and efforts to make uneccessary optimalizations or choices (ex. changing tech stack) because you imagine it will be needed in the future.

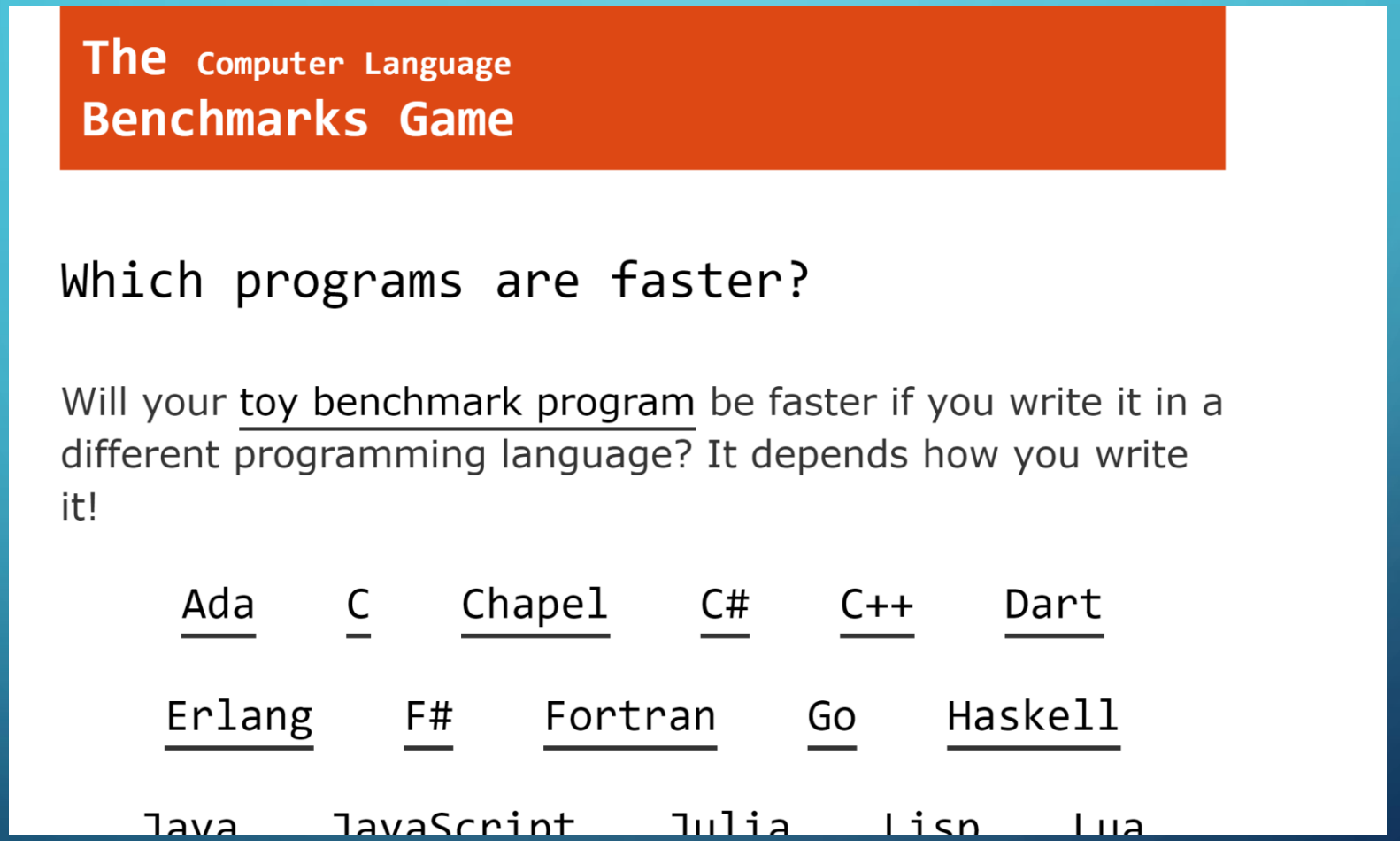
Premature optimization

casual car for everyday city driving

The most difficult part

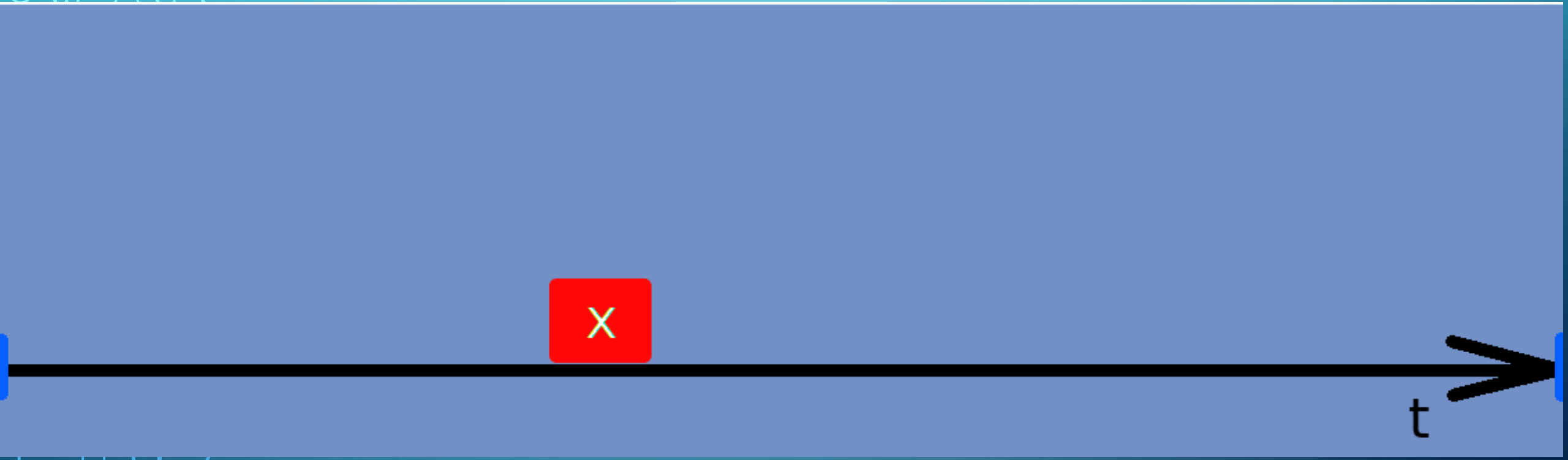
Benchmark

Wrong benchmarks

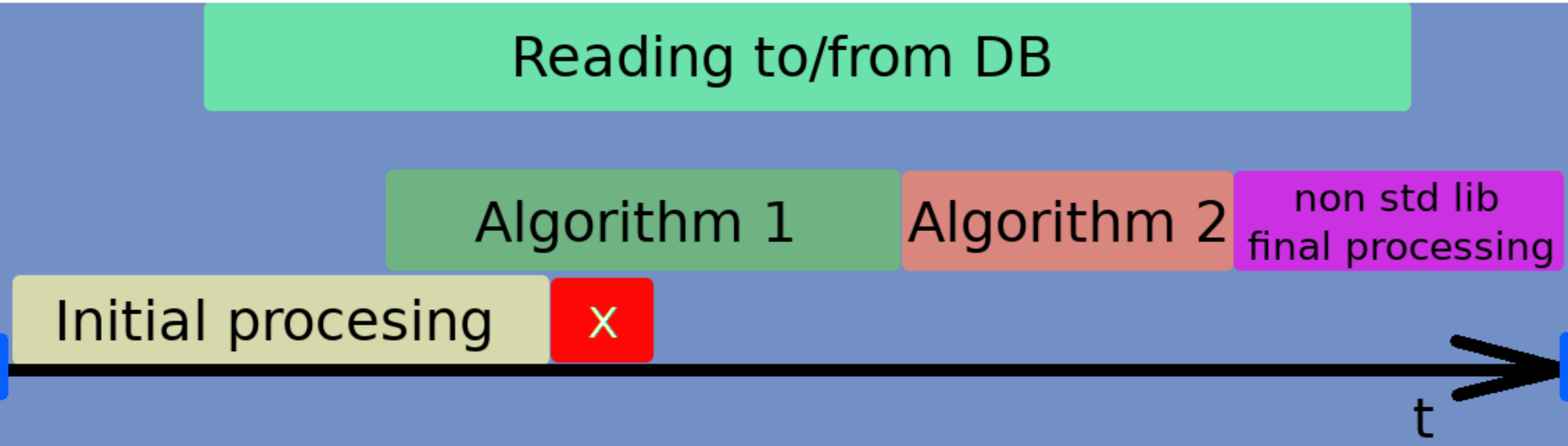


Source: https://benchmarksgame-team.pages.debian.net/benchmarksgame/

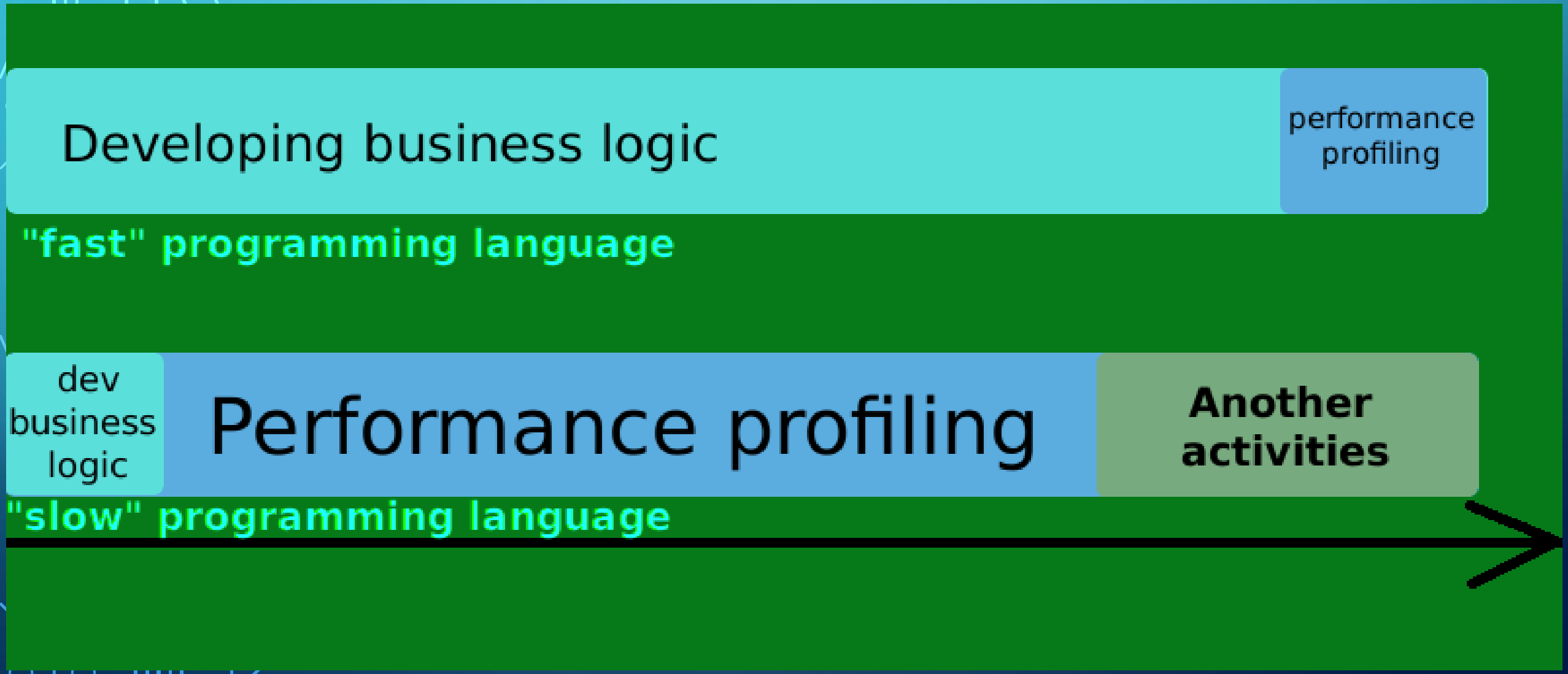
Typical benchmark



Real world app



Real world project timeline



Better benchmarks

TechEmpower

Web Framework Benchmarks

Introduction | Previous Rounds | Round 15 2018-02-14 | Round 16 2018-06-06 | **Round 17 2018-10-30** | Motivation & Questions

JSON serialization | Single query | Multiple queries | Fortunes | Data updates | Plaintext | Physical | Cloud

Fortunes

Best (bar chart) | Data table | Latency | Framework overhead

Best fortunes responses per second, Dell R440 Xeon Gold + 10 GbE (330 tests)

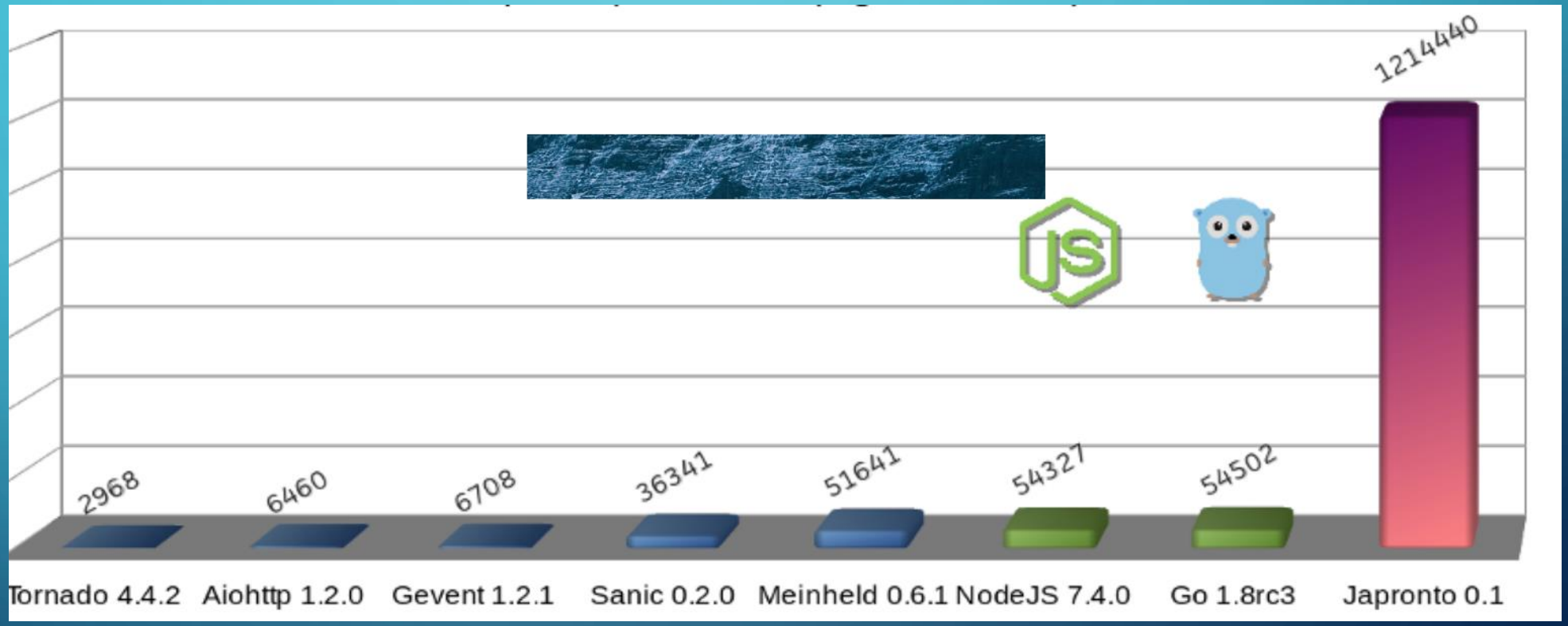
Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1	h2o	423,298	100.0%	0	Plt	C	Non	Non	Lin	Pg	Lin	Raw	Rea
2	vertx-postgres	336,138	79.4%	0	Plt	Jav	ver	Non	Lin	Pg	Lin	Raw	Rea
3	fasthttp-postgresql	329,433	77.8%	0	Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
4	swoole	303,514	71.7%	0	Plt	PHP	Non	swo	Lin	My	Lin	Raw	Rea
5	revenj-jvm	298,539	70.5%	0	Ful	Jav	Svt	Res	Lin	Pg	Lin	Ful	Rea
6	aspcore-ado-pg	294,285	69.5%	1	Plt	C#	.NE	kes	Lin	Pg	Lin	Raw	Rea
7	proteus	293,550	69.3%	0	Mcr	Jav	Utw	Non	Lin	Pg	Lin	Raw	Rea
8	cpoll_cppsp-raw	270,826	64.0%	0	Plt	C++	Non	Non	Lin	My	Lin	Raw	Rea
9	jlhttp-postgres	252,441	59.6%	0	Plt	Jav	JLH	Non	Lin	Pg	Lin	Raw	Rea
10	actix-raw	251,204	59.3%	0	Plt	Rus	Non	act	Lin	Pg	Lin	Raw	Rea

source: https://www.techempower.com/benchmarks/

Fast (cpu-bound) languages

- Assembler?
- C?
- C++?
- Rust?
- Java?
- Go?
- Python?

Case study: Python (Japronto)



source: https://github.com/squeaky-pl/japronto

Case study: Python (Japronto)

As user @heppu points out Go's stdlib HTTP server can be 12% faster than the graph shows when written more carefully. Also there is the awesome fasthttp server for Go that apparently is only 18% slower than Japronto in this particular benchmark. Awesome! For details see https://github.com/squeaky-pl/japronto/pull/12 and https://github.com/squeaky-pl/japronto/pull/14.

These results of a simple "Hello world" application were obtained on AWS c4.2xlarge instance. To be fair all the contestants (including Go) were running single worker process. Servers were load tested using wrk with 1 thread, 100 connections and 24 simultaneous (pipelined) requests per connection (cumulative parallelism of 2400 requests).

The source code for the benchmark can be found in benchmarks directory.

The server is written in hand tweaked C trying to take advantage of modern CPUs. It relies on picohttpparser for header & chunked-encoding parsing while uvloop provides asynchronous I/O. It also tries to save up on system calls by combining writes together when possible.

source: https://github.com/squeaky-pl/japronto

Case study: Go (fasthttp)

Fast HTTP package for Go. Tuned for high performance. Zero memory allocations in hot paths. Up to 10x faster than net/http

- *Are there known net/http advantages comparing to fasthttp?*

  Yes:

  - net/http supports HTTP/2.0 starting from go1.6.
  - net/http API is stable, while fasthttp API constantly evolves.
  - net/http handles more HTTP corner cases.
  - net/http should contain less bugs, since it is used and tested by much wider audience.
  - net/http works on Go older than 1.5.

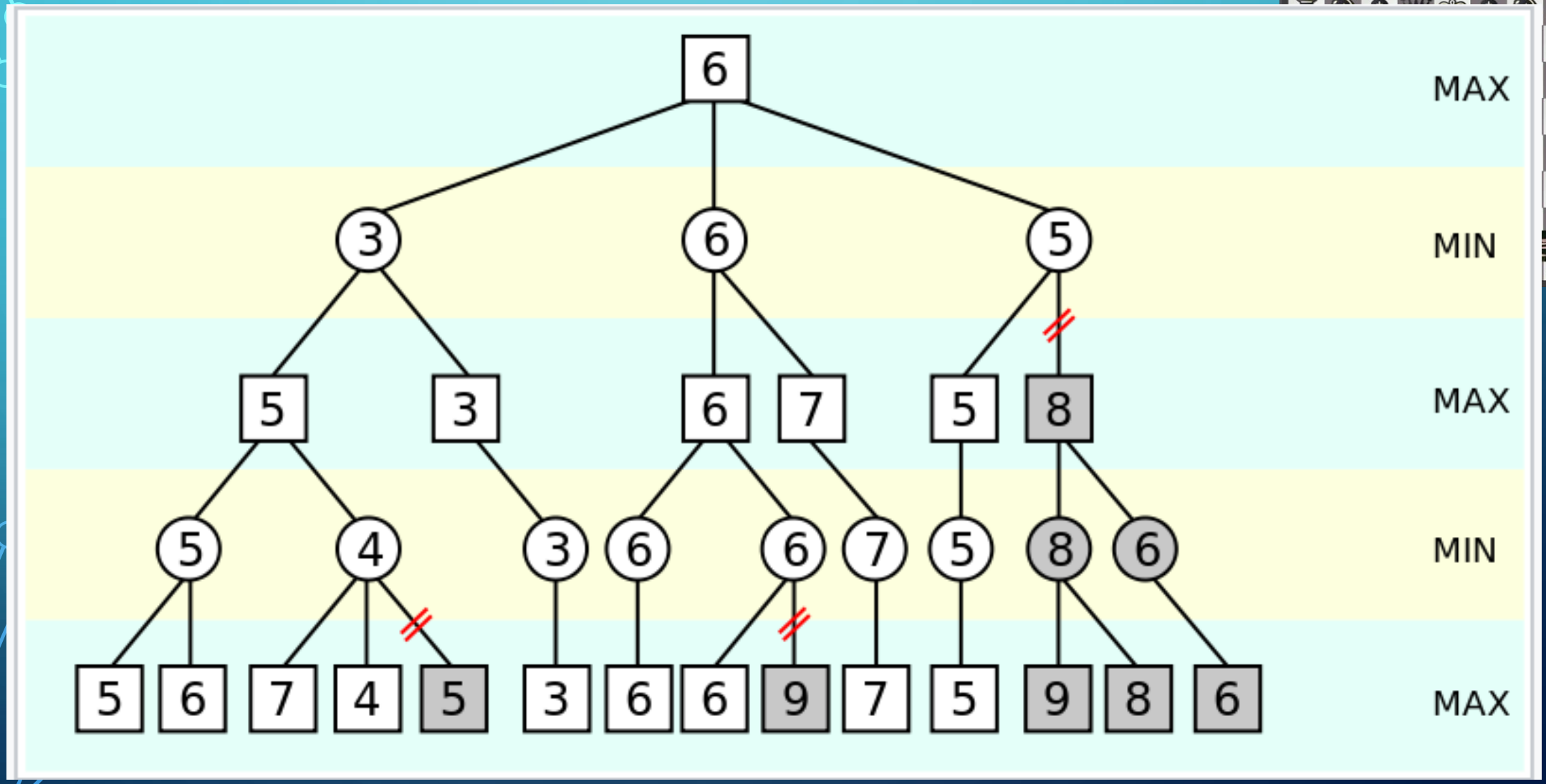
source: https://github.com/valyala/fasthttp

Case study: chess engines

The most important factors:

1) playing strength (ELO)
2) analysis speed (nodes per sec), especially in alpha-beta prunning engines

Case study: chess engines



Case study: stockfish

- it's a chess engine (strongest alpha-beta prunning)
- it's written in C++
- it uses multithreading efficiently
- It has many derivates, asmFish is one of them (written in x86 asm)

Case study: stockfish

Rank	Name	Rating			Score	Average Opponent	Draws	Games	LOS
		Elo	+	−					
1	Stockfish 270918 64-bit 4CPU	3457	+28	−27	77.9%	−188.7	43.3%	457	
	SugaR XPrO 1.5.3 64-bit 4CPU	3435	+25	−25	72.4%	−147.8	50.6%	512	87.2%
	Stockfish 9 64-bit 4CPU	3434	+15	−15	72.3%	−152.7	50.6%	1416	54.7%
	SugaR XPrO 1.4 64-bit 4CPU	3412	+20	−20	72.3%	−149.8	51.8%	840	95.9%
	asmFish 051117 64-bit 4CPU	3410	+20	−19	69.8%	−129.6	55.7%	819	55.3%
	SugaR XPrO 1.2 64-bit 4CPU	3401	+21	−20	70.4%	−136.4	52.5%	754	75.5%
	ShashChess Pro 1.0 64-bit 4CPU	3399	+36	−35	67.9%	−107.3	60.7%	224	53.4%
	Stockfish 8 64-bit 4CPU	3380	+14	−13	65.1%	−96.7	62.1%	1647	82.8%
	SugaR XPrO 1.3 64-bit 4CPU	3375	+23	−22	66.0%	−103.1	57.4%	598	64.2%
	Stockfish 9 64-bit	3369	+13	−13	73.4%	−165.1	46.4%	2277	67.2%
									48.2%

Suprisingly asmFish is neither the strongest or „fastest” type of stockfish version.

Stockfish: the strongest because of The evaluation speed?

- No: Houdini 6.03 (alpha-beta prunning) chess engine is faster (nodes per sec on same machine). But Houdini 6.03 is a slighty weaker engine.
- Yes: Better chess algorithms (but the main algorithm is the same)
- Leela Chess Zero: NN-based chess engine, 1000x slower in terms of nodes per sec. Currently stronger than stockfish.

Stockfish vs Leela Chess Zero

- **Leela Chess Zero:** NN-based (MCTS) chess engine, build to reflect AlphaZero DeepMind ideas (using NN and MCTS in chess)
- Self learning algorithm (games between LC0 vs LC0 and LC0 vs rest of the world)
- Slower more than **1000x** in terms of nodes per second than stockfish due to the different algorithm
- It's playing strength it's very close to stockfish, especially at very fast games (bullet and blitz chess)

Case study: fibonacci numbers

Task: get 50th numer from the fibonacci sequence

C vs Python

```
mateusz@matpc ~ gcc -O3 fibonacci.c -o fibonacci_in_c
mateusz@matpc ~ time ./fibonacci_in_c
Fibonacci of 12586269025: ./fibonacci_in_c  28,18s user 0,01s system 99% cpu 28,276 total
```

```
mateusz@matpc ~ time ./fibonacci.py
12586269025
./fibonacci.py  0,01s user 0,02s system 97% cpu 0,032 total
```

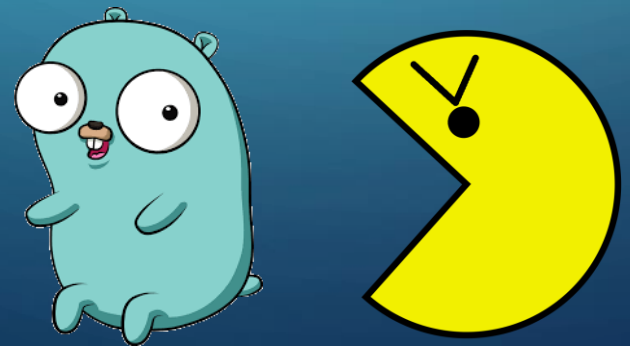
Case study: fibonacci numbers

```c
#include <stdio.h>

unsigned long int fibbonacci(unsigned long int n) {
   if(n == 0){
      return 0;
   }

   if(n == 1) {
      return 1;
   }

   return (fibbonacci(n-1) + fibbonacci(n-2));
}

int main() {
   printf("Fibonacci of %li: " , fibbonacci(50));
}
```

```python
#!/usr/bin/env python

def fib(num):
  a,b = 0, 1
  for i in range(0, num):
    a, b = b, a + b

  return a

print(fib(50))
```

Performance eaters

- algorithms
- doing unnecessary work (GC, logging)
- non cpu-bound waiting
- not using multithreading
- using too many threads
- slow (cpu-bound) programming language?



Gopher Performance World

Go: benchmarking

```
timeStart := time.Now()

// a lot of code for measure

timeStop := time.Since(timeStart)
```

Go: benchmarking

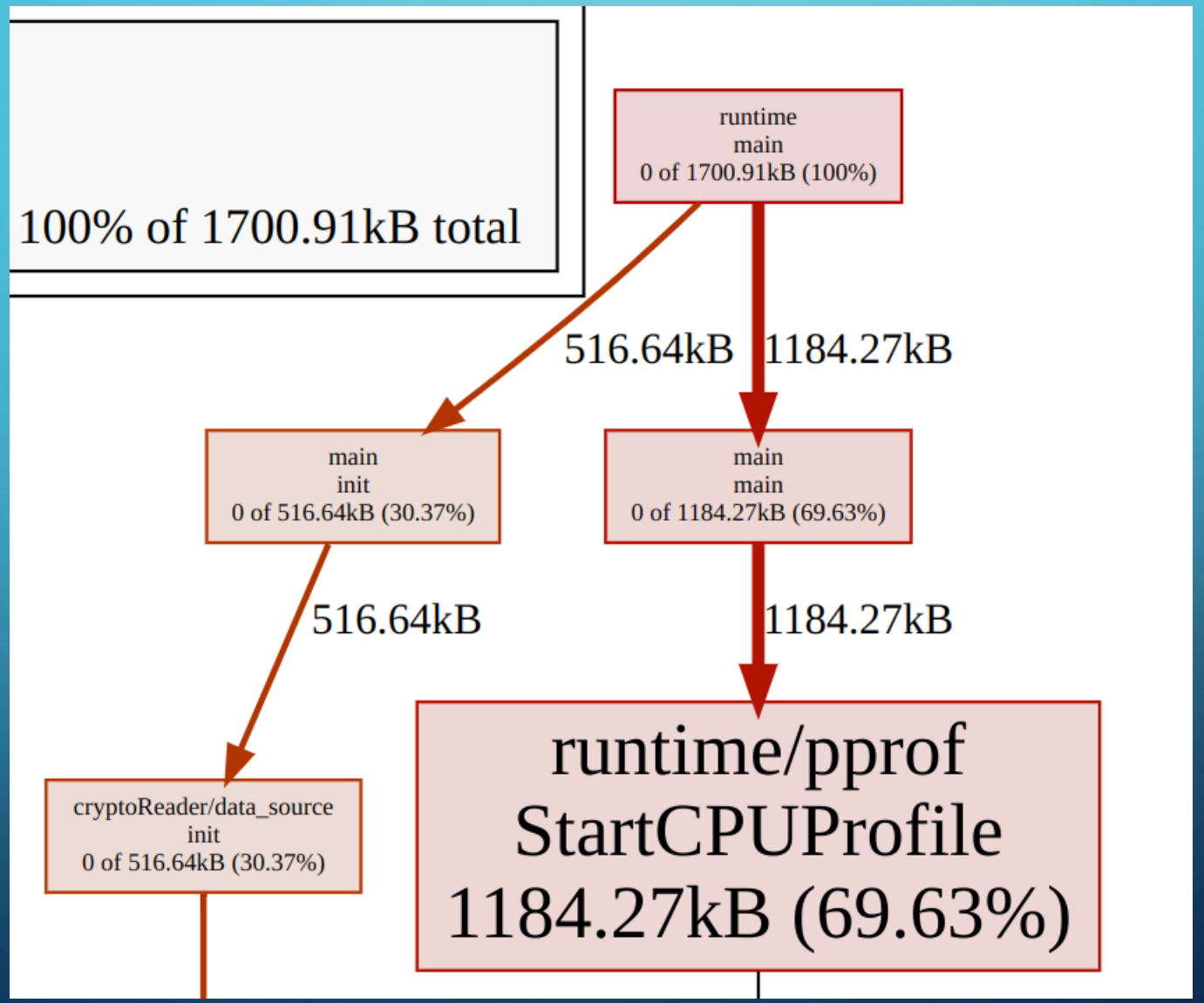
```go
import (
    "bytes"
    "strings"
    "testing"
)

var strLen int = 1000

func BenchmarkConcatString(b *testing.B) {
    var str string

    i := 0

    b.ResetTimer()
    for n := 0; n < b.N; n++ {
        str += "x"

        i++
        if i >= strLen {
            i = 0
            str = ""
        }
    }
}
```

Go: pprof

Package pprof writes runtime profiling data in the format expected by the pprof visualization tool. Useful for profiling CPU & Memory.

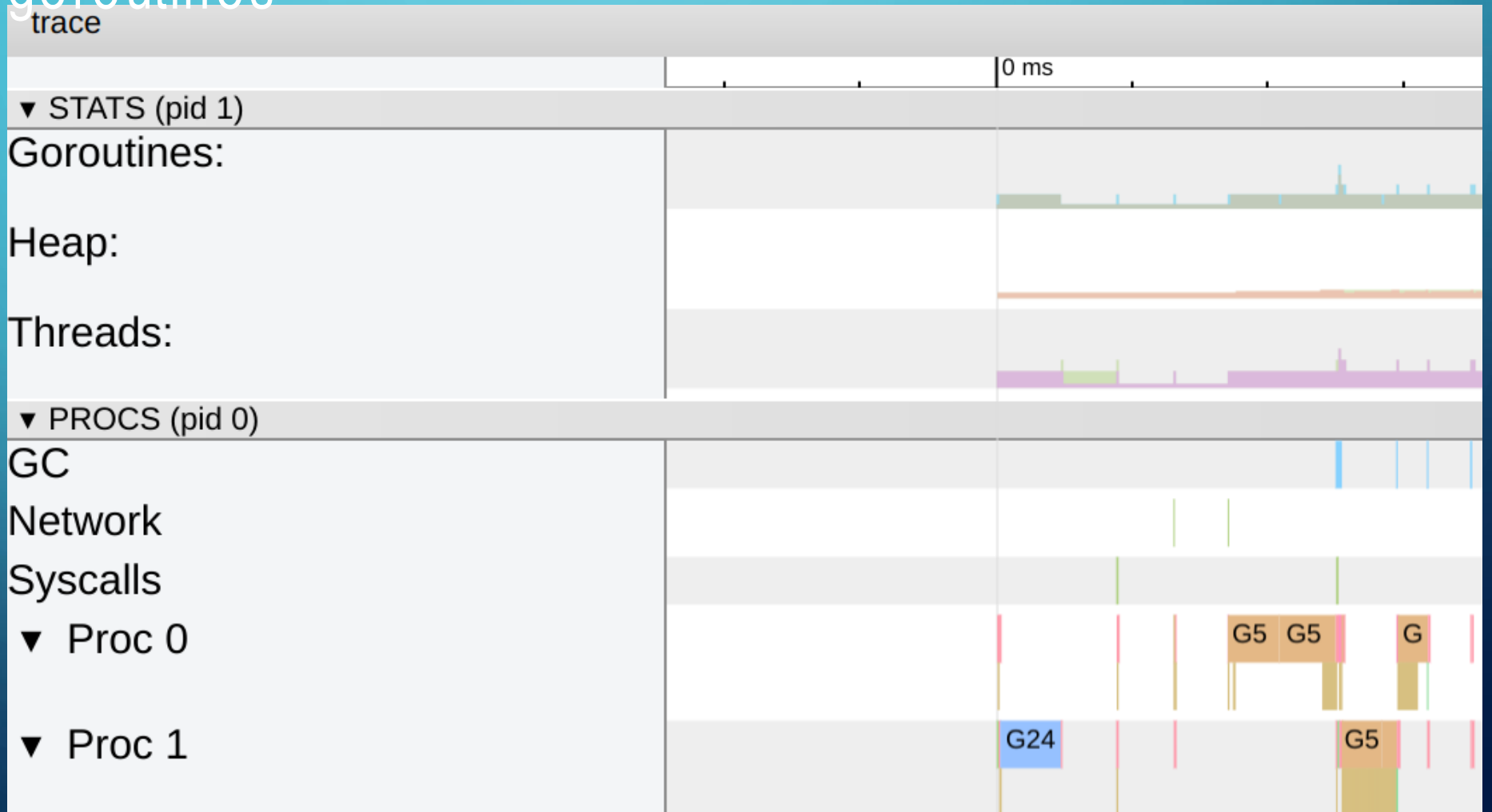
```
Type: cpu
Time: Jan 4, 2019 at 6:47pm (CET)
Duration: 802.68ms, Total samples = 450ms (56.06%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 250ms, 55.56% of 450ms total
Showing top 10 nodes out of 164
      flat  flat%   sum%        cum   cum%
      50ms 11.11% 11.11%       50ms 11.11%  golang.org/x/net/html.(*Tokenizer).re
      30ms  6.67% 17.78%       30ms  6.67%  runtime.heapBitsSetType
      30ms  6.67% 24.44%       30ms  6.67%  runtime.memmove
      20ms  4.44% 28.89%       20ms  4.44%  bufio.(*Reader).ReadByte
      20ms  4.44% 33.33%       30ms  6.67%  compress/flate.(*decompressor).huffSy
      20ms  4.44% 37.78%       20ms  4.44%  crypto/elliptic.p256Sqr
      20ms  4.44% 42.22%       70ms 15.56%  encoding/asn1.parseField
      20ms  4.44% 46.67%       30ms  6.67%  github.com/andybalholm/cascadia.attri
      20ms  4.44% 51.11%       20ms  4.44%  runtime.futex
      20ms  4.44% 55.56%       30ms  6.67%  syscall.Syscall
```

Go: pprof


100% of 1700.91kB total
runtime
main
0 of 1700.91kB (100%)
516.64kB
1184.27kB
main
init
0 of 516.64kB (30.37%)
main
main
0 of 1184.27kB (69.63%)
516.64kB
1184.27kB
cryptoReader/data_source
init
0 of 516.64kB (30.37%)
runtime/pprof
StartCPUProfile
1184.27kB (69.63%)

Go: trace

Useful for trace execution of the program over time and goroutines



Performance profiling steps

1. Measurement:
   - make benchmark and get results,
   - do profiling to determine a bottleneck,

2. Make appropiate changes in your code

3. Repeat 1) and 2) if results are still no acceptable

Go: string concatenation

```
var str string

i := 0

b.ResetTimer()
for n := 0; n < b.N; n++ {
    str += "x"

    i++
    if i >= strLen {
        i = 0
        str = ""
    }
}
```

Go: string concatenation

```
var buffer bytes.Buffer

i := 0

b.ResetTimer()
for n := 0; n < b.N; n++ {
    buffer.WriteString( s: "x")

    i++
    if i >= strLen {
        i = 0
        buffer = bytes.Buffer{}
    }
}
```

Go: string concatenation

```
var builder strings.Builder

i := 0

b.ResetTimer()
for n := 0; n < b.N; n++ {
    builder.WriteString( s: "x")

    i++
    if i >= strLen {
        i = 0
        builder = strings.Builder{}
    }
}
```

```
BenchmarkConcatString-6      10000000          172 ns/op
BenchmarkConcatBuffer-6     200000000            7.37 ns/op
BenchmarkConcatBuilder-6   1000000000            2.35 ns/op
```

Go: GC

Garbage Collector (GC) allows you to focus on business logic instead of memory management. However, this can lead to some performance tradeoffs in some cases.

GC usually does what you would have done in your code without such mechanism.

GC improves slightly (usually) in every new Go version, but don’t treat this statement as a general rule

Turning off GC completely is highly not recommended unless you really know what are you doing (you risk crash of your app)

Go: GC

runtime/debug – some tunning/stats options

```
const toAllocate = 65536

func main() {
    debug.SetGCPercent( percent: -1)
    now := time.Now()
    loop := 1000000
    for i := 0; i < loop; i++ {
        slice := make([]byte, toAllocate)
        i += len(slice) * 0
    }
    howMuch := time.Since(now)
    fmt.Printf( format: "took %s to allocate %d bytes %d times", howMuch, toAllocate, loop)
}
```

Disabling GC may improve performance if there are many short lived memory allocations but it's not recommended overall due to it's side effects

Go: pointer vs value

Performance dilemma

pointer	value
stack or heap (mostly) – GC traces it, exception: unsafe.Pointer (as uintptrs)	stack, no GC pressure
passing bigger data structures	passing small values
underlying value can be modified	value is for read-only
no thread safe (synch needed)	thread safe

Go: array vs slice

[...]array
Size is known during compile time thus not flexibe. Compiler checks validity of indexes. Good for performance (keep on stack) if you know exact size of array.

[]slice
the rest cases that does not apply to arrays. Accessing elements out of scope results
in runtime panic. Preallocated slices are better for performance.

Go: Escape analysis

The compiler warns if variables will be stored on heap.
It applies for dynamic data structures which size cannot be determined during compile time

```
go build -gcflags '-m'
```

Go: Escape analysis

```go
type MyStruct struct {
    name    string
    list    [3]string
}

func main() {
    a := MyStruct{
        name: "name",
        list: [3]string{"a","b","c"},
    }

    somefunc1(a)
    if a.name == "name" {
        fmt.Printf( format: "somefunc1 NOT MODIFIED struct\n")
    }

    somefunc2(&a)
    if a.name != "name" {
        fmt.Printf( format: "somefunc2 MODIFIED struct\n")
    }
}

func somefunc1(a MyStruct) {
    a.name = "xyz"
}

func somefunc2(s *MyStruct){
    s.name = "x"
}
```

Go: Escape analysis

```
 mateusz@matpc  ~/gocracow/escape_analysis  go build -gcflags '-m' esc
# command-line-arguments
./escape_analysis_struct.go:29:6: can inline somefunc1
./escape_analysis_struct.go:33:6: can inline somefunc2
./escape_analysis_struct.go:18:11: inlining call to somefunc1
./escape_analysis_struct.go:23:11: inlining call to somefunc2
./escape_analysis_struct.go:23:12: main &a does not escape
./escape_analysis_struct.go:29:16: somefunc1 a does not escape
./escape_analysis_struct.go:33:16: somefunc2 s does not escape
```

Go: Escape analysis

```go
type MyStruct struct {
    name    string
    list    []string
}

func main() {
    a := MyStruct{
        name: "name",
        list: []string{"a","b","c"},
    }

    somefunc1(a)
    if a.name == "name" {
        fmt.Printf( format: "somefunc1 NOT MODIFIED struct\n")
    }

    somefunc2(&a)
    if a.name != "name" {
        fmt.Printf( format: "somefunc2 MODIFIED struct\n")
    }
}

func somefunc1(a MyStruct) {
    a.name = "xyz"
}

func somefunc2(s *MyStruct){
    s.name = "x"
}
```

Go: Escape analysis

```
mateusz@matpc ~/gocracow/escape_analysis go build -gcflags '-m' escap
# command-line-arguments
./escape_analysis_struct.go:29:6: can inline somefunc1
./escape_analysis_struct.go:33:6: can inline somefunc2
./escape_analysis_struct.go:18:11: inlining call to somefunc1
./escape_analysis_struct.go:23:11: inlining call to somefunc2
./escape_analysis_struct.go:15:11: main []string literal does not escape
./escape_analysis_struct.go:23:12: main &a does not escape
./escape_analysis_struct.go:29:16: somefunc1 a does not escape
./escape_analysis_struct.go:33:16: somefunc2 s does not escape
```

Go: Escape analysis

```go
type MyStruct struct {
    name    string
    list    []string
}

func main() {
    a := MyStruct{
        name: "name",
        list: []string{"a","b","c"},
    }

    somefunc1(a)
    if a.name == "name" {
        fmt.Printf( format: "somefunc1 NOT MODIFIED struct\n")
    }

    somefunc2(&a)
    if a.name != "name" {
        fmt.Printf( format: "somefunc2 MODIFIED struct\n")
    }
}

func somefunc1(a MyStruct) {
    a.name = "xyz"
}

func somefunc2(s *MyStruct){
    s.name = "x"
    s.list = append(s.list, elems...: "x")
}
```

Go: Escape analysis

```
mateusz@matpc ~/gocracow/escape_analysis go build -gcflags '-m'
# command-line-arguments
./escape_analysis_struct.go:29:6: can inline somefunc1
./escape_analysis_struct.go:33:6: can inline somefunc2
./escape_analysis_struct.go:18:11: inlining call to somefunc1
./escape_analysis_struct.go:23:11: inlining call to somefunc2
./escape_analysis_struct.go:15:11: []string literal escapes to heap
./escape_analysis_struct.go:23:12: main &a does not escape
./escape_analysis_struct.go:29:16: somefunc1 a does not escape
./escape_analysis_struct.go:33:16: leaking param content: s
```

Go: too few/many goroutines

Sometimes not using many threads (goroutines in Go) may affect performance.

The opposite scenario is also possible – using too many goroutines can impact performance negatively

func GOMAXPROCS

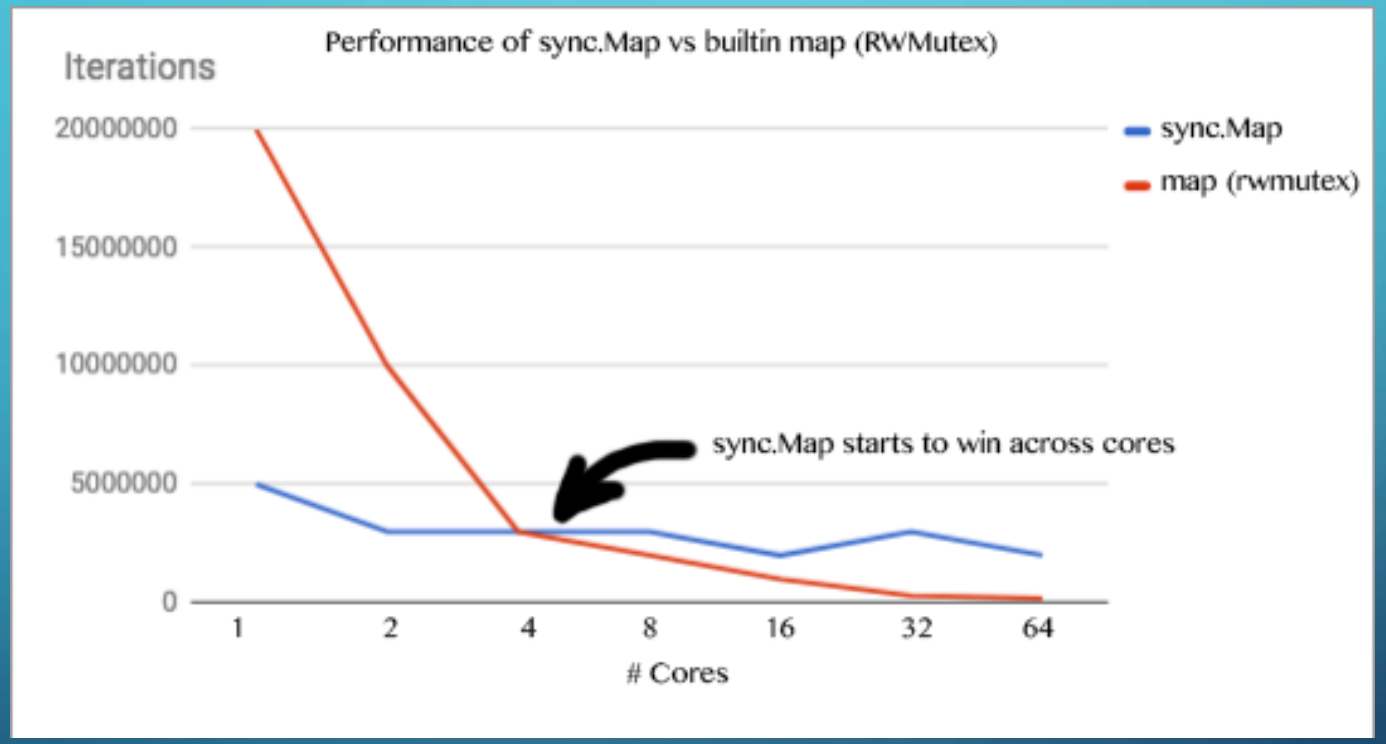
```
func GOMAXPROCS(n int) int
```

GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. If n < 1, it does not change the current setting. The number of logical CPUs on the local machine can be queried with NumCPU. This call will go away when the scheduler improves.

https://golang.org/pkg/runtime/#GOMAXPROCS

Go: sync.Map

Use sync.Map instead of "normal" map with mutexes



https://medium.com/@deckarep/the-new-kid-in-town-gos-sync-map-de24a6bf7c2c

Go: sync.Pool

Use it to reduce memory allocations temporary objects than can be stored/retrievied later

By allocation reduction you can reduce GC activities

Go: sync.Pool

```go
type Database struct {
    FieldA    string
    FieldB    string
    FieldC    string
    Names
    Lists
}

type Names struct {
    first string
    last string
}

type Lists struct {
    ListInts []int
    ListStrings []string
}

var pool = sync.Pool{
    New: func() interface{} {
        return &Database{}
    },
}
```

Go: sync.Pool

```go
func BenchmarkNoPool(b *testing.B) {
    var db *Database

    for n := 0; n < b.N; n++ {
        db = &Database{
            FieldA: "Some text in FieldA",
            FieldB: "Some text in FieldB",
            FieldC: "Some text in FieldC",
            Names: Names{
                first: "FirstName",
                last: "LastName",
            },
            Lists: Lists{
                ListInts: []int{1,2,3,4,5,6,7,8,9,10},
                ListStrings: []string{"a","b","c"},
            },
        }
    }

    _ = db
}
```

```go
func BenchmarkWithPool(b *testing.B) {
    for n := 0; n < b.N; n++ {
        db := pool.Get().(*Database)

        db.FieldA = "Changed text in FieldA"
        db.FieldB = "Changed text in FieldB"
        db.FieldC = "Changed text in FieldC"
        db.Names.first = "Name"
        db.Names.last = "Last"
        db.ListInts = []int{10,20,30,40,50}
        db.ListStrings = []string{"x", "y", "z"}

        pool.Put(db)
    }
}
```

Go: sync.Pool

```
mateusz@matpc ~/gocracow/pool go test -bench=. -benchmem pools_test.go
goos: linux
goarch: amd64
BenchmarkNoPool-6       10000000               253 ns/op             256 B/op          3 allocs/op
BenchmarkWithPool-6     10000000               157 ns/op              96 B/op          2 allocs/op
PASS
ok      command-line-arguments  4.453s
```

Go: interface{} problem

```go
type NumbersInt64 struct {
    numbers []int64
}

func (n *NumbersInt64) initialize() {
    for i:=0 ; i<=1000; i++ {
        n.numbers = append(n.numbers, int64(i*2))
    }
}

func (n *NumbersInt64) addNums() {
    slice := n.numbers
    for idx, n := range slice {
        slice[idx] = n + 10
    }
}
```

```go
type NumbersEmptyInterface struct {
    numbers []interface{}
}

func (n *NumbersEmptyInterface) initialize() {
    for i:=0 ; i<=1000; i++ {
        n.numbers = append(n.numbers, i*2)
    }
}

func (n *NumbersEmptyInterface) addNums() {
    slice := n.numbers
    for idx, n := range slice {
        slice[idx] = n.(int) + 10
    }
}
```

Go: interface{ } problem

```go
func BenchmarkAddIntegerNumbers(b *testing.B) {
    b.ResetTimer()
    for n := 0; n < b.N; n++ {
        numbers := &NumbersInt64{}
        numbers.initialize()
        numbers.addNums()
    }
}

func BenchmarkAddEmptyInterfaceNumbers(b *testing.B) {
    b.ResetTimer()
    for n := 0; n < b.N; n++ {
        numbers := &NumbersEmptyInterface{}
        numbers.initialize()
        numbers.addNums()
    }
}
```

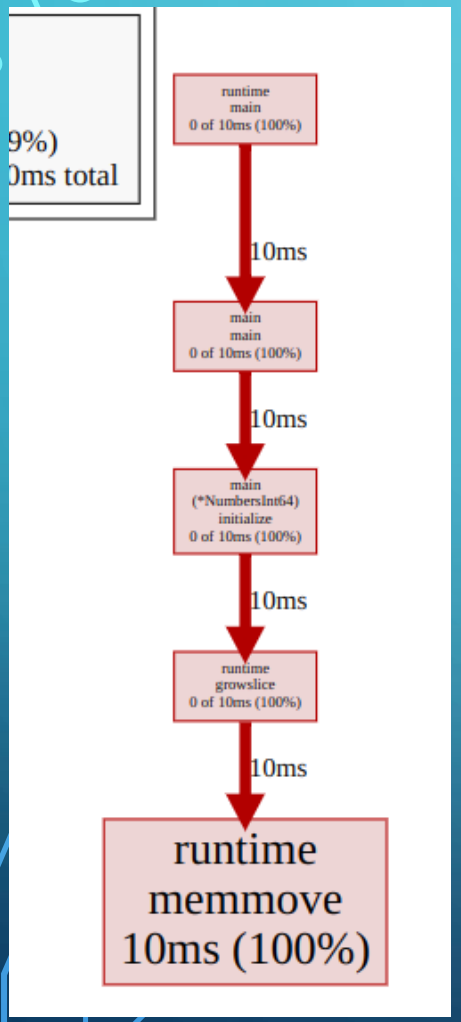
```
BenchmarkAddIntegerNumbers-6              300000          5263 ns/op
BenchmarkAddEmptyInterfaceNumbers-6        50000         63439 ns/op
PASS
```

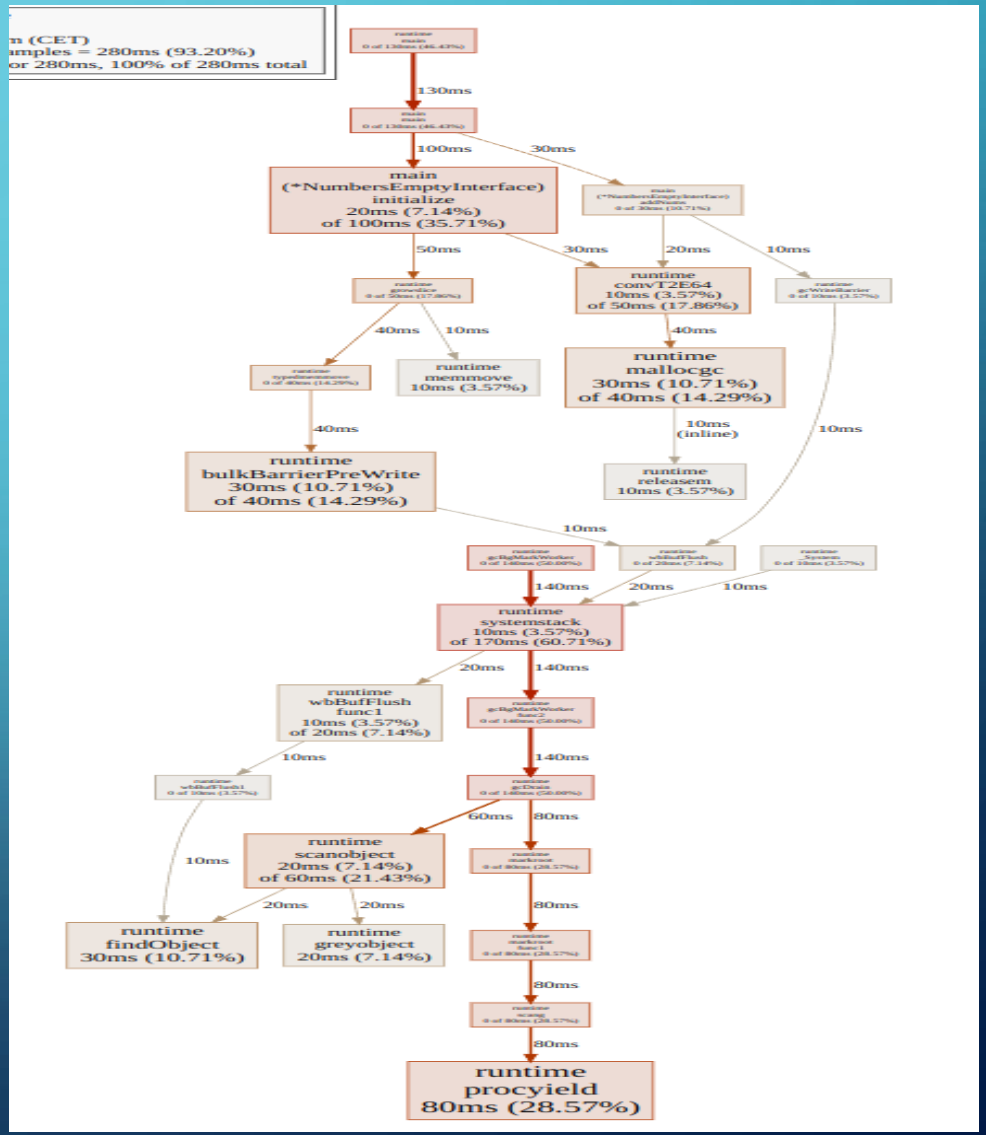
Go: interface{ } problem

why?

...lets find out using pprof

Go: interface{ } problem





Go: faster libs than std

std lib	non std (fast) lib
net/http	fasthttp
html/template	fasttemplate
encoding/json	gojay

Go: non discussed here

- channels vs mutexes
- Advanced GC tunning and GC-wise programming
- struct paddings (memory saving)
- unsafe.Pointer and unsafe package

Conclusions

Focus on your business logic and on the architecture

Conclusions

Write idiomatic and clean Go code

Conclusions

Use apropiate algorithms

Conclusions

Avoid using interface{} if it’s possible and use a specific type instead

Conclusions

Use tests, linters, code reviews, etc

Conclusions

Detect your bottlenecks and profile your code when it’s needed

Conclusions

Use microoptimalisations if they are required

Conclusions

Rewrite a performance-problematic part in another programming language if it offers functionality which you need.

Use it when rewritting does not cost too much time and/or there is the lib in another language for you purpose

Conclusions

Rewrite the entire app in a performant cpu-bound language if it won't take too much time, all required libs are available and the effort is really worth of it

Links

- https://golang.org/pkg/runtime/pprof/
- https://golang.org/pkg/runtime/trace/
- https://blog.golang.org/profiling-go-programs
- https://github.com/golang/go/wiki/Performance
- http://www.brendangregg.com/
- https://github.com/dgryski/go-perfbook
- https://dave.cheney.net/tag/performance
- http://bigocheatsheet.com/

Q&A

https://github.com/mateusz-szczyrzyca/gocracow3