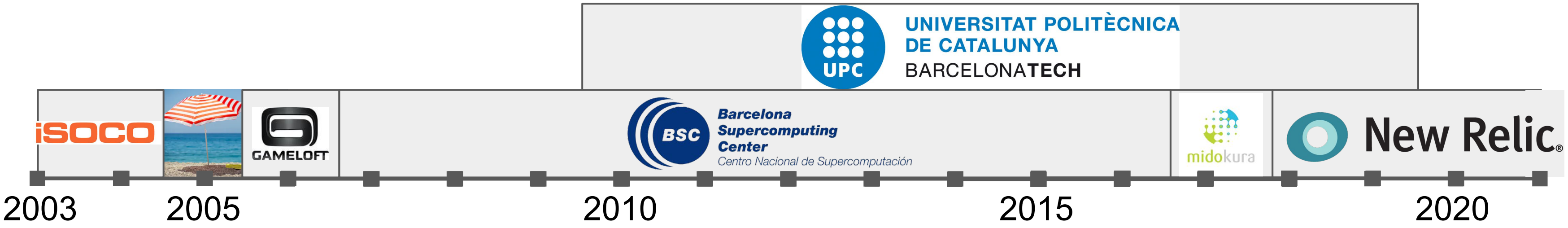
Pointers vs Values: digging into the performance war

Mario Macías Lloret
Senior software engineer at New Relic

Barcelona Golang Meetup
January 2020

twitter.com/MaciasUPC github.com/mariomac

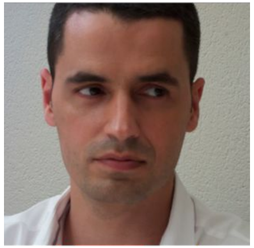
Who am I?


UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH
UPC
iSOCO
GAMELOFT
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
midokura
New Relic
2003
2005
2010
2015
2020

The Core Agents and Open Standards Team

JF

Mario

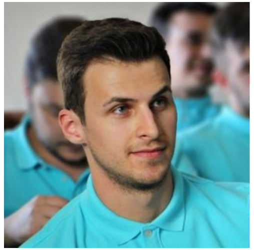


Antonio

Juan

Carlos

David



Cristian

Frank

Table of contents

Using values vs using pointers

- Values
  - Safe against nil
  - Cleaner
    - no need to check for nil
    - no pointer operators *, &
- Pointers
  - Allow passing arguments by reference
  - Allow sharing a common state between different instances

The super-optimizer opinion

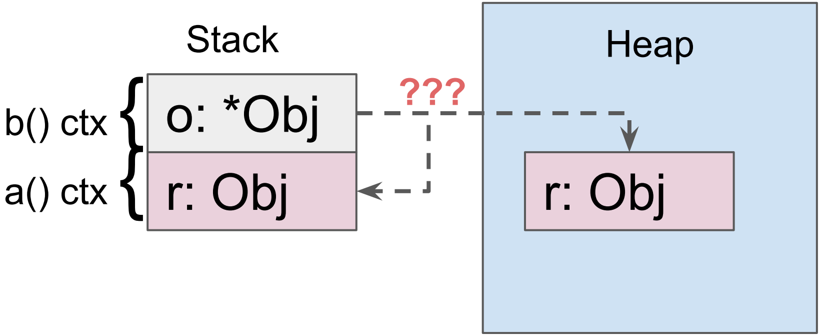
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"

-- Donald Knuth

January 2019 BCN Golang Meetup

- T(Heap allocation) >> T(Stack Allocation)
- Golang applies “Escape Analysis” techniques to infer where an object is allocated
- Abuse of pointers escape values to the heap

```
func a() *Obj {
   r := Obj{}
   // ... do something
   return &r;
}
func b() {
   o := a()
   // ... do something
}
```



January 2019 BCN Golang Meetup

- T(Heap allocation) >> T(Stack Allocation)
-
-

```
f
}
func b() {
  o := a()
  // ... do something
}
```

Benchmark Results

```
go test ./donut/. -bench=Benchmark -benchmem
```

```
BenchmarkValue-4      5000000     262 ns/op    15 B/op    0 allocs/op
BenchmarkPointers-4   5000000     332 ns/op    79 B/op    1 allocs/op
```

Our code (including random number generation and scoring operations) using values is ~23% faster than using pointers!

Table of contents

Digging more: µBenchmarks

- Small, localized benchmarks to test a single system functionality.
- Not really meaningful from a wider application point of view

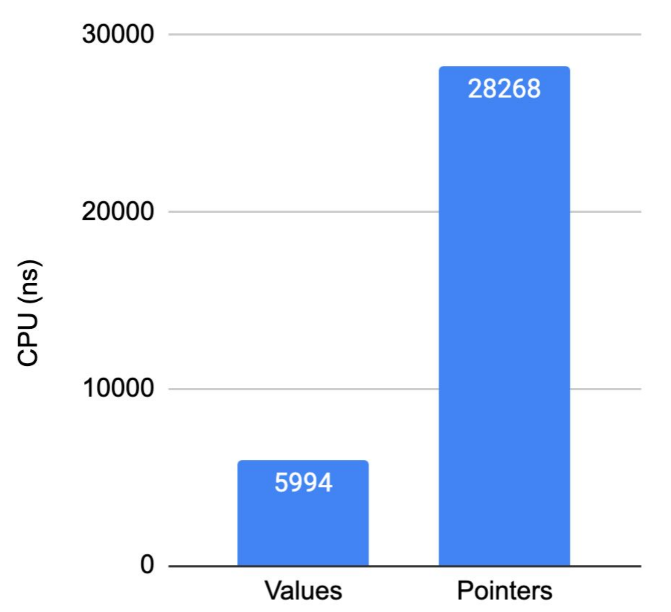
```go
type Foo struct {
    A int
    B int
    C int
}
```

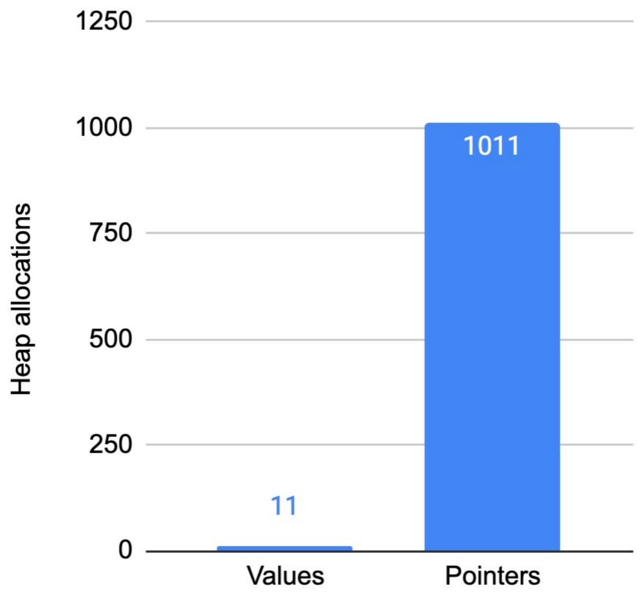
```go
const FoosLength = 1000

func addFoos(foos []Foo) []Foo {

}
```

```go
func addFoosP(foos []*Foo) []*Foo {
    for i := 0; i < FoosLength; i++ {
        foos = append(foos, &Foo{
            A: i,
        })
    }
    return foos
}
```

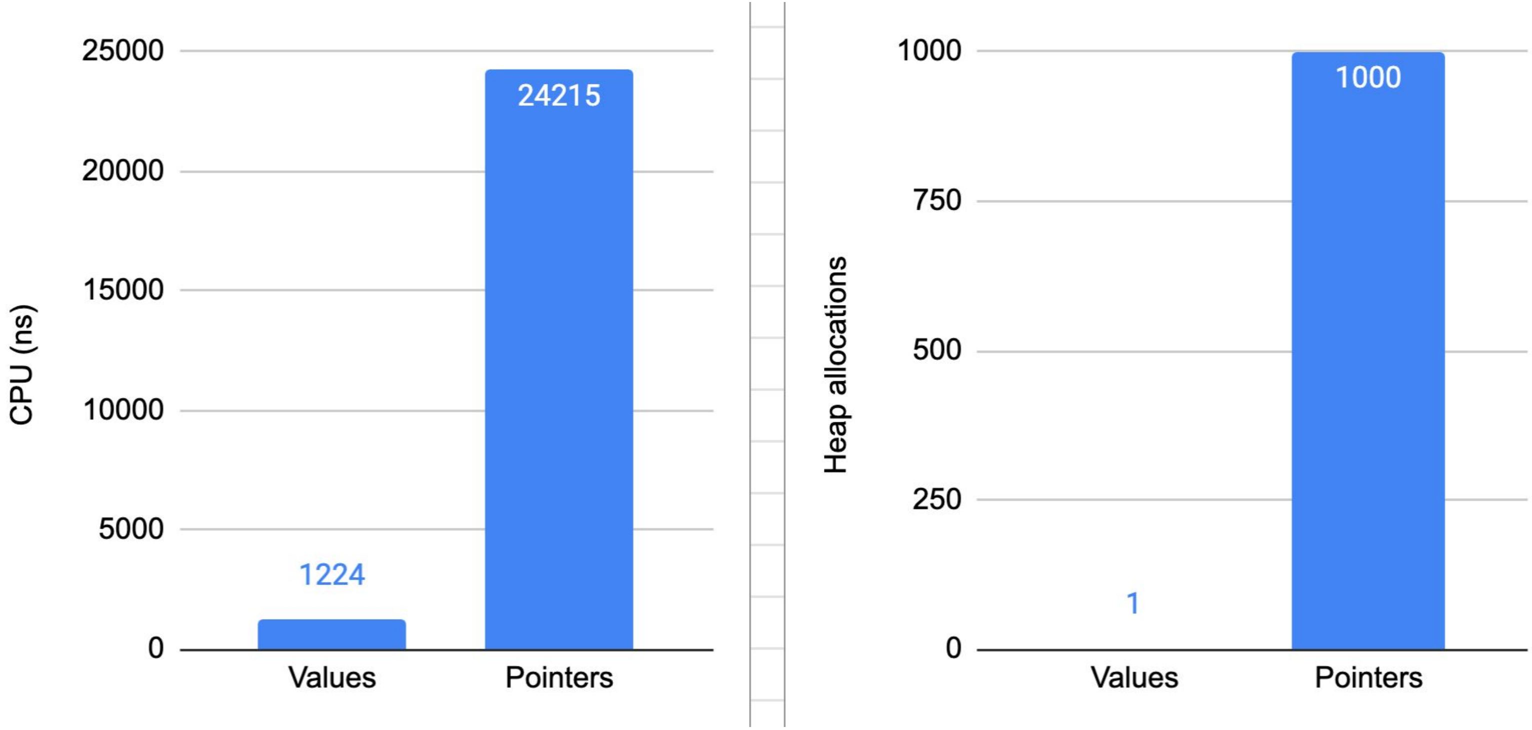
µBenchmarks: initial results

```
func BenchmarkSliceCreation_Values(b *testing.B) {
    for i
        -
    }
}

func Benc                                        st
    for i
        -
    }
}
```





µBenchmarks: pre-allocated arrays



Adding a value to an array

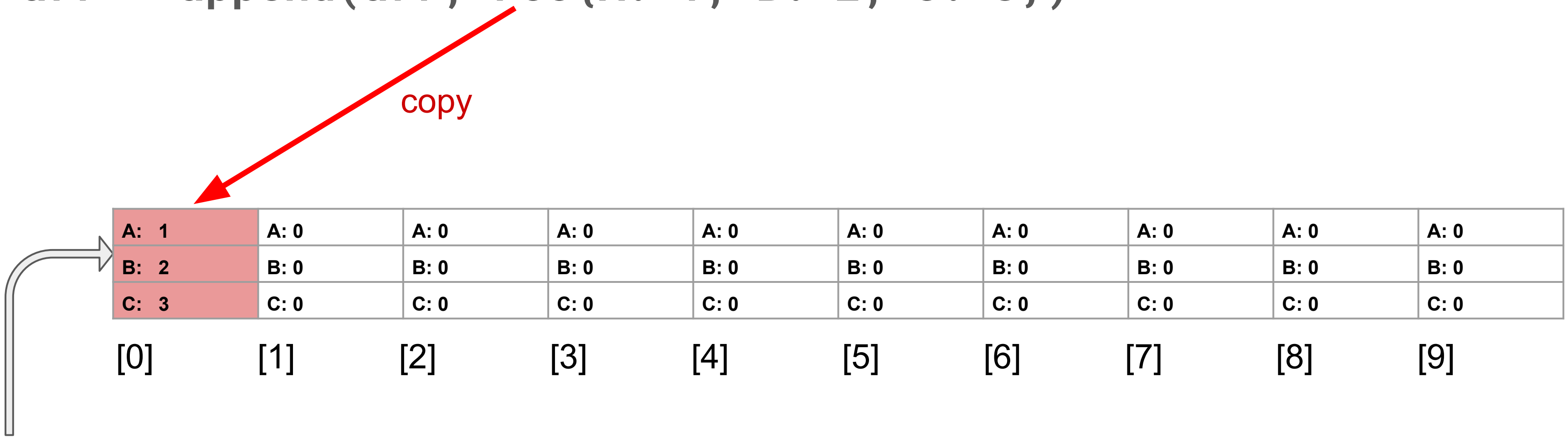
```
arr := make([]Foo, 0, 10)
```

A: 0	A: 0	A: 0	A: 0	A: 0	A: 0	A: 0	A: 0	A: 0	A: 0
B: 0	B: 0	B: 0	B: 0	B: 0	B: 0	B: 0	B: 0	B: 0	B: 0
C: 0	C: 0	C: 0	C: 0	C: 0	C: 0	C: 0	C: 0	C: 0	C: 0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

```
arr
  - length: 0
  - capacity: 10
```

Adding a value to an array

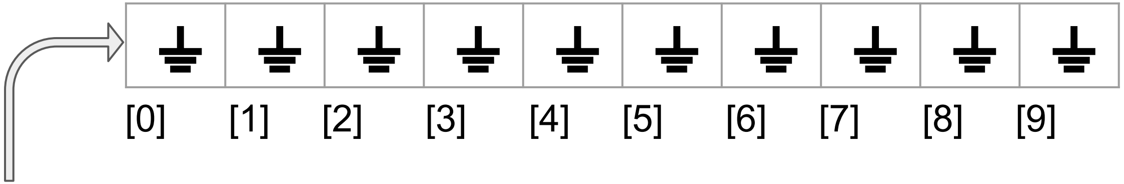
```
arr := make([]Foo, 0, 10)
arr = append(arr, Foo{A: 1, B: 2, C: 3})
```




```
arr
  - length: 1
  - capacity: 10
```

Adding a reference to an array

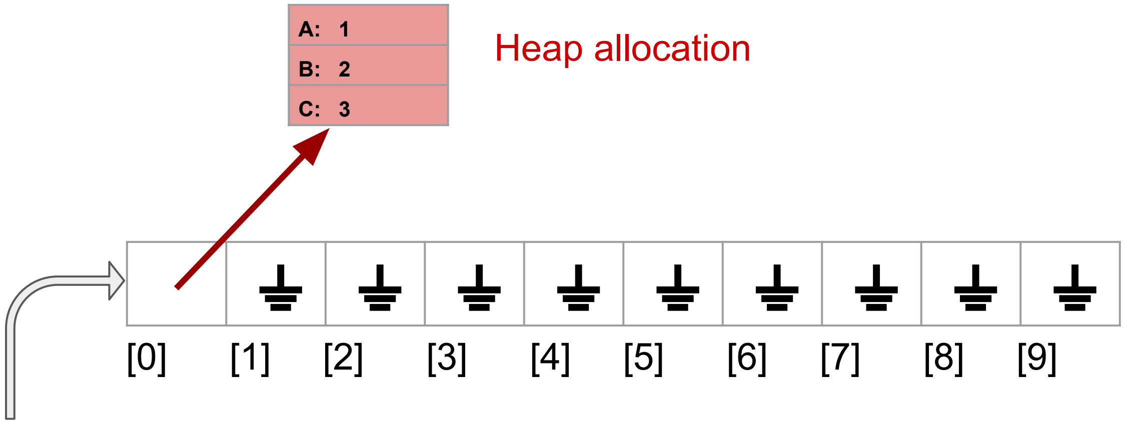
```
arr := make([]*Foo, 0, 10)
```




```
arr
  - length: 0
  - capacity: 10
```

Adding a reference to an array

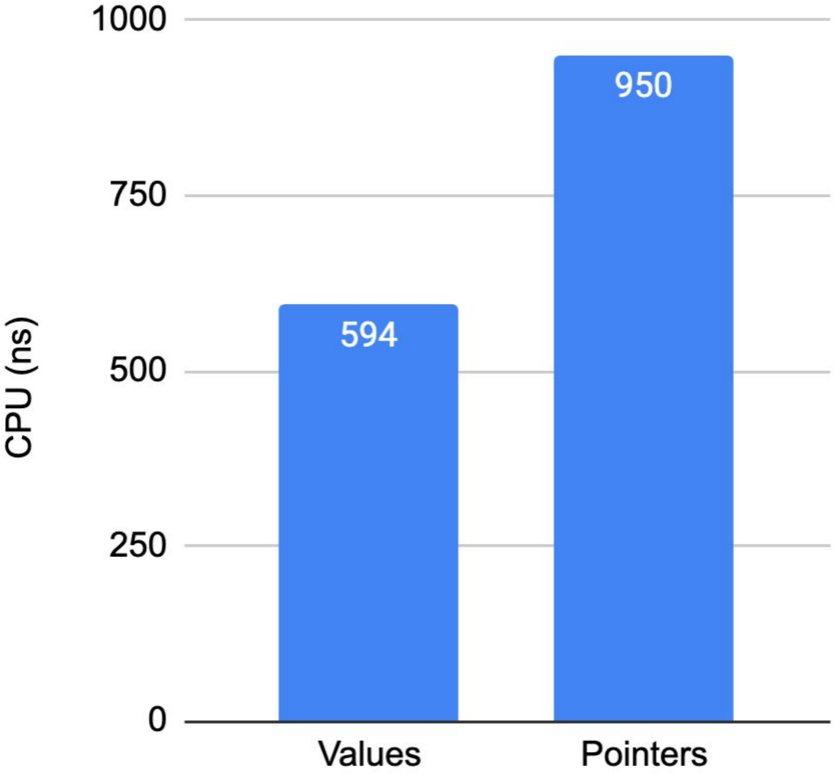
```
arr := make([]*Foo, 0, 10)
arr = append(arr, &Foo{A: 1, B: 2, C: 3})
```




```
arr
  - length: 1
  - capacity: 10
```

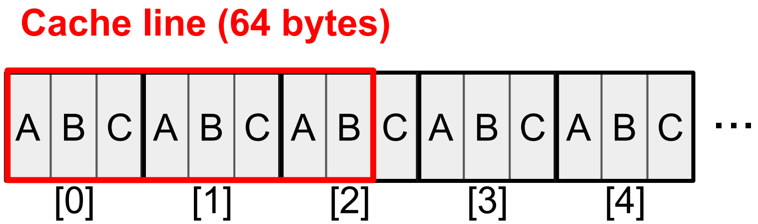
µBenchmark: array iteration (no allocations)

```
func sumAll(foo
    sum := 0
    for _, f :=
        sum +=
    }
    return sum
}
```

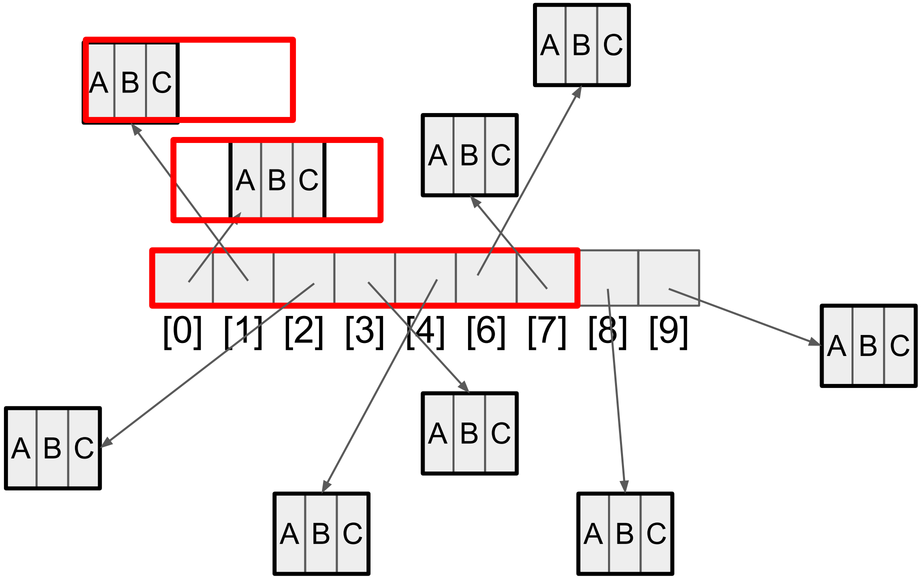



```
oos []*Foo) int {

= range foos {
f.A + f.B + f.C
```

Enforcing cache memory contiguity



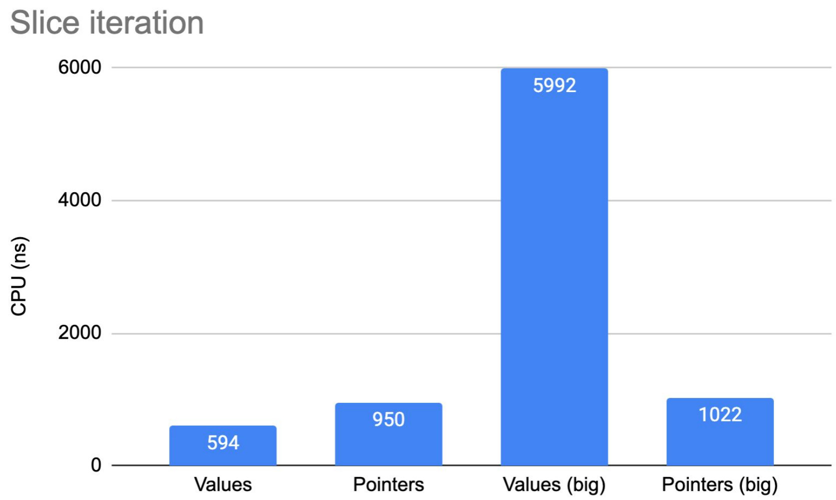
[ ]Foo{ … }



Increased cache misses

µBenchmark: structs >64 bytes

```go
type Foo struct {
    A int
    B int
    C int
    D int
    E int
    F int
    G int
    H int
    I int
    J int
    K int
}
```



Local µoptimization: minimize local var copy

```
func sumAllLR(foos []Foo) int {
    sum := 0
    for i := range foos {
        f := &foos[i]
        sum += f.A + f.K
    }
    return sum
}
```

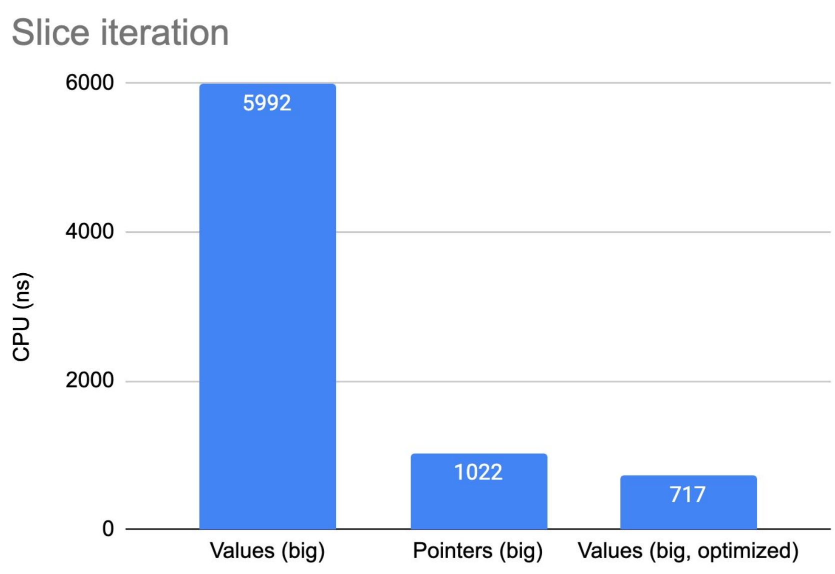


Table of contents

Real world bench: dimensional metrics translator

New relic Flat sample

- event_type: SystemSample
- operatingSystem: Linux
- agentVersion: 1.8.82
- cpuPercent: 30
- diskFreePercent: 85
- hostname: ip-AC1F0D60
- instanceType: t2.small
- memoryUsedBytes: 1109519701
- etc….

Dimensional metrics sample

Common:

- event_type: SystemSample
- operatingSystem: Linux
- agentVersion: 1.8.82
- hostname: ip-AC1F0D60
- instanceType: t2.small

Metrics:

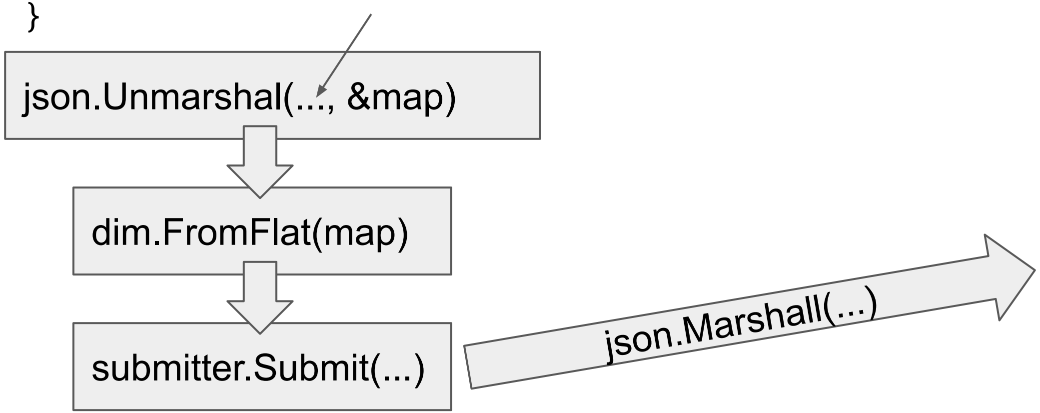
name: cpuPercent
value: 30

name: diskFreePercent
value: 85

name: memoryUsedBytes
value: 1109519701

Dimensional metrics translator

```json
{
  "event_type": "SystemSample",
  "operatingSystem": "Linux",
  "agentVersion": "1.8.82",
  "cpuPercent": 30,
  "diskFreePercent": 85,
  "hostname": "ip-AC1F0D60",
  "instanceType": "t2.small",
  "memoryUsedBytes": 1109519701
}
```



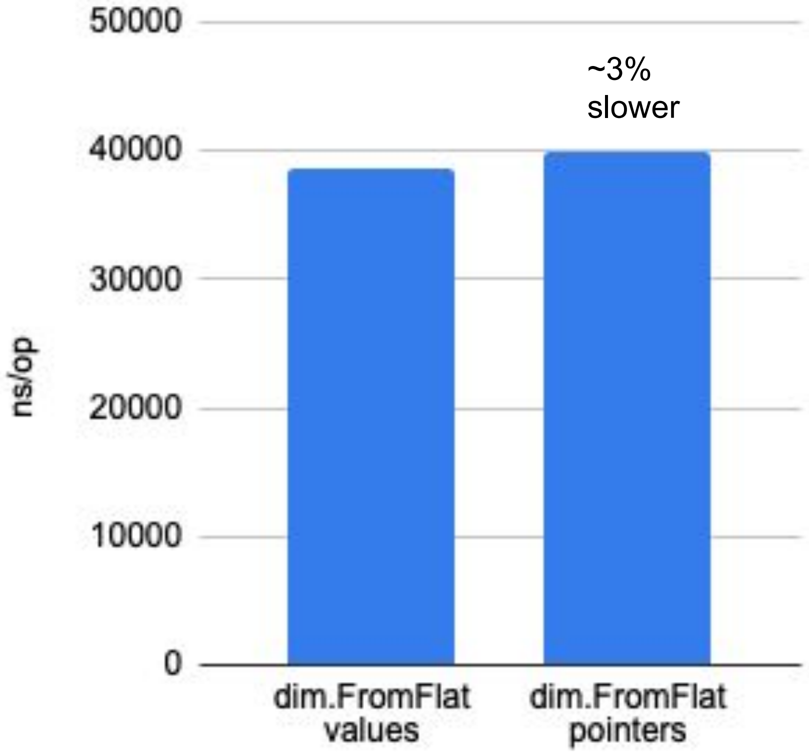

```json
{
  "Common": {
    "event_type": "SystemSample",
    "operatingSystem": "Linux",
    "agentVersion": "1.8.82",
    "hostname": "ip-AC1F0D60",
    "instanceType": "t2.small"
  },
  Metrics: [{
    "name": "cpuPercent",
    "type": "Gauge", "value": 30
  }, {
    "name": "diskFreePercent",
    "type": "Gauge", "value": 85
  }, {
    "name": "memoryUsedBytes",
    "type": "Gauge",
    "value": 1109519701
  }]
}
```

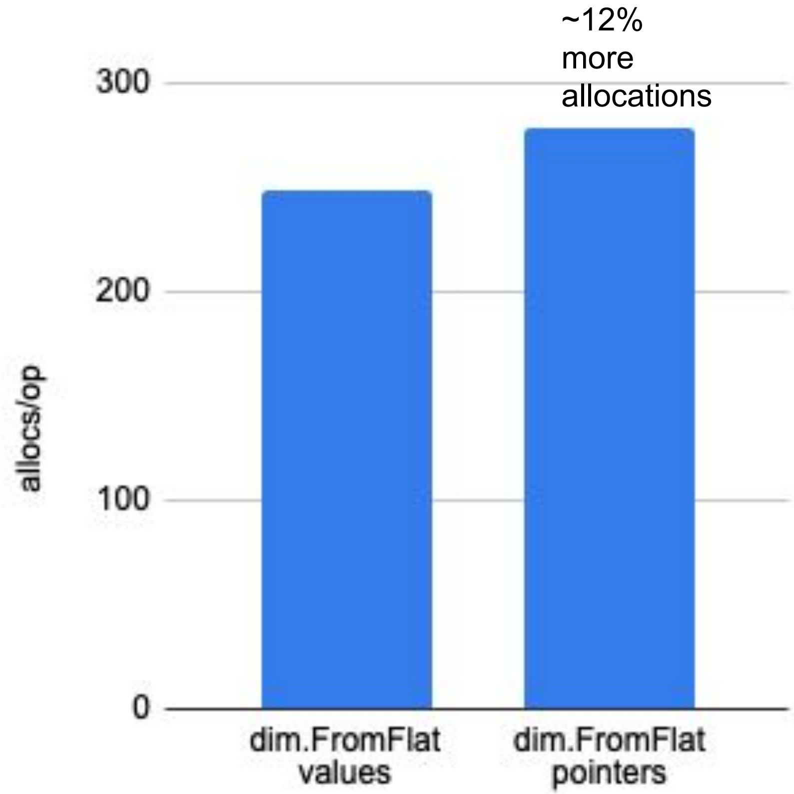
Benchmark: same code, 2 versions

```go
type Type string
type Payload struct {
    Common  *Common
    Metrics []*Metric
}
type Common struct {
    Attributes map[string]string
    Timestamp  int64
}
type Metric struct {
    Name  string
    Type  Type
    Value float64
}
type Submitter interface {
    Submit(p *Payload) error
}
func FromFlat(values map[string]interface{}) *Payload
```

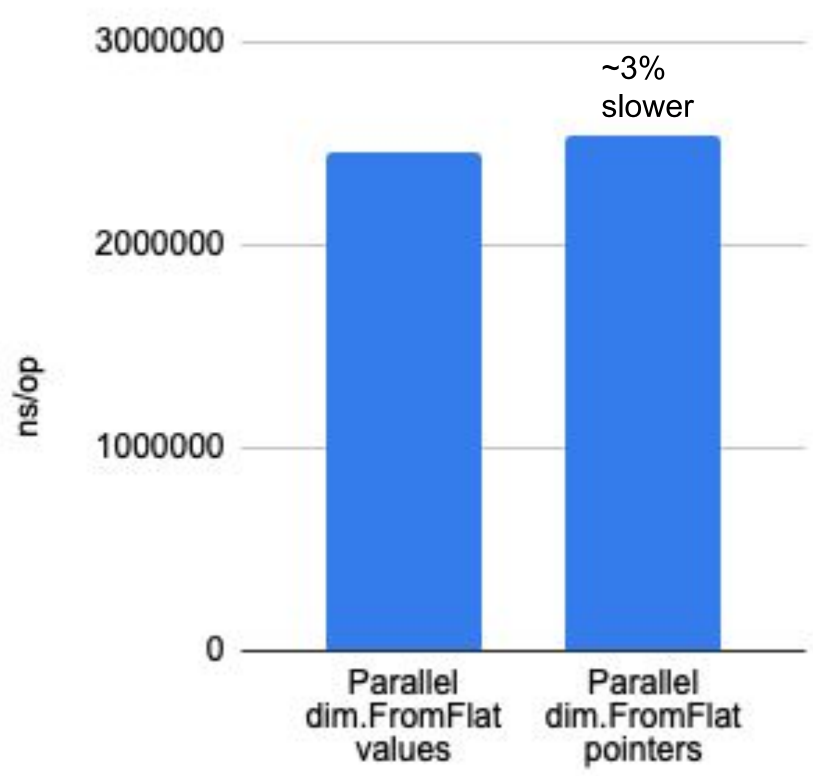
```go
type Type string
type Payload struct {
    Common  Common
    Metrics []Metric
}
type Common struct {
    Attributes map[string]string
    Timestamp  int64
}
type Metric struct {
    Name  string
    Type  Type
    Value float64
}
type Submitter interface {
    Submit(p Payload) error
}
func FromFlat(values map[string]interface{}) Payload
```

Benchmark Results





Benchmark Results (100 parallel goroutines)



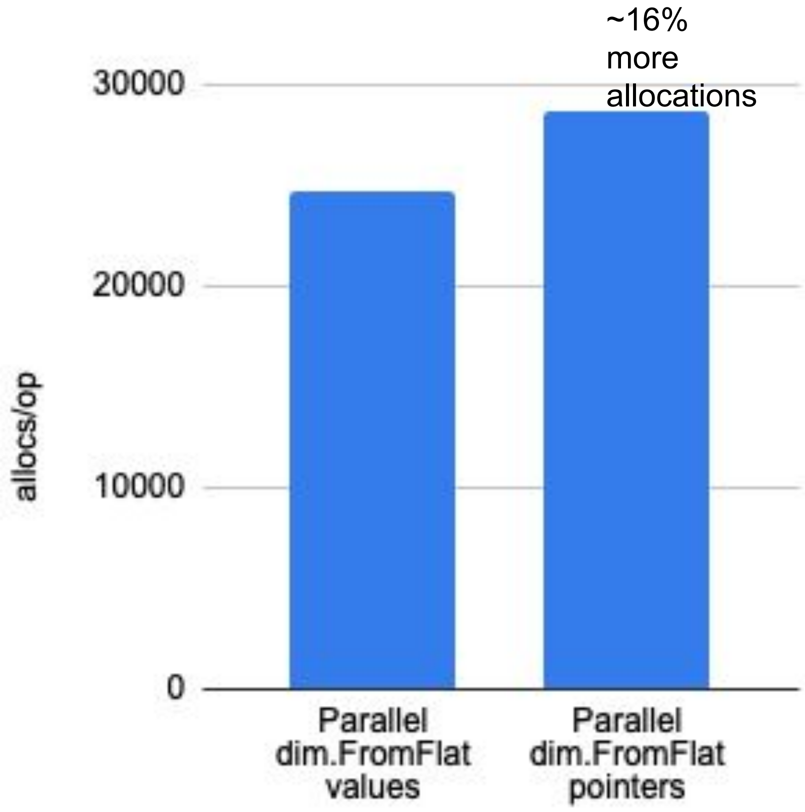
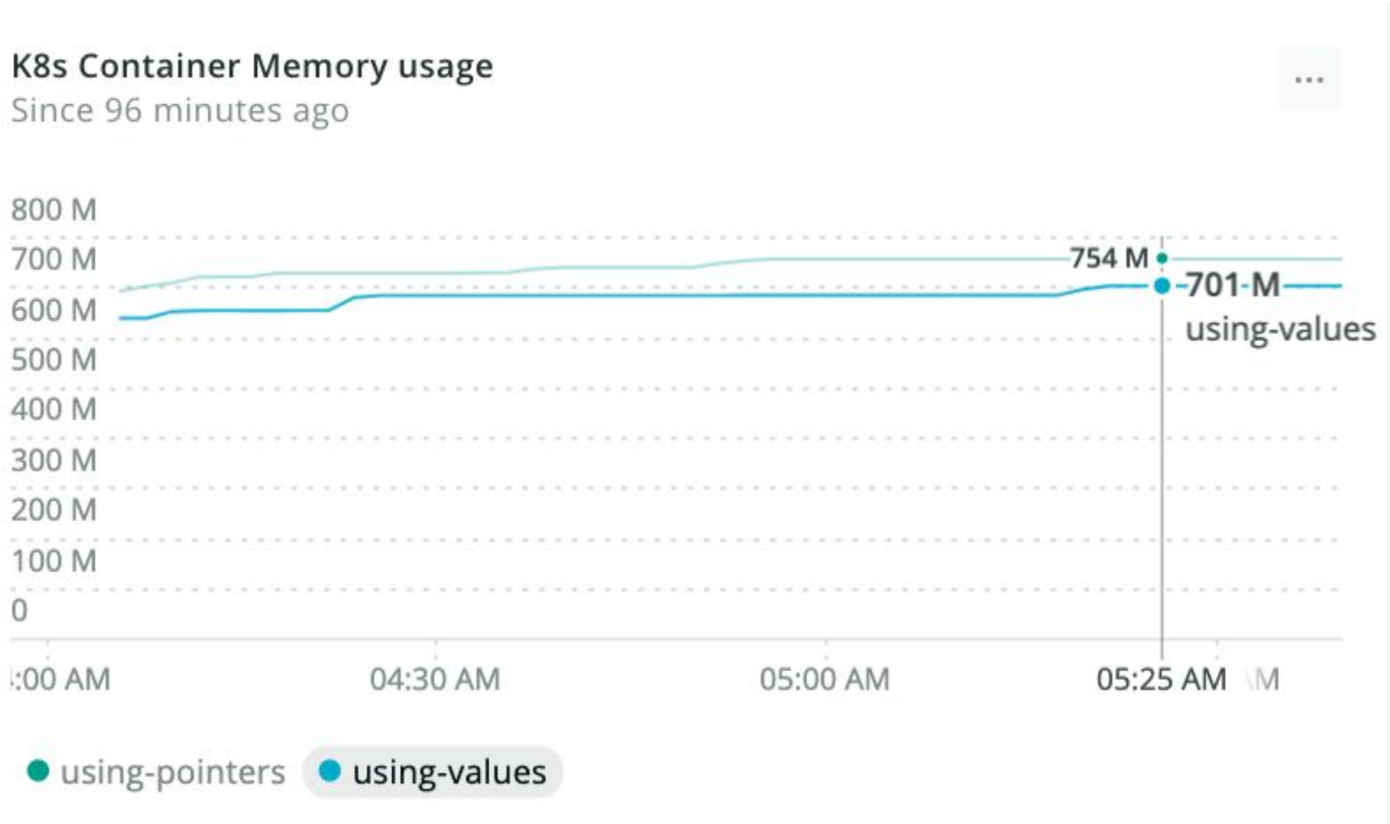


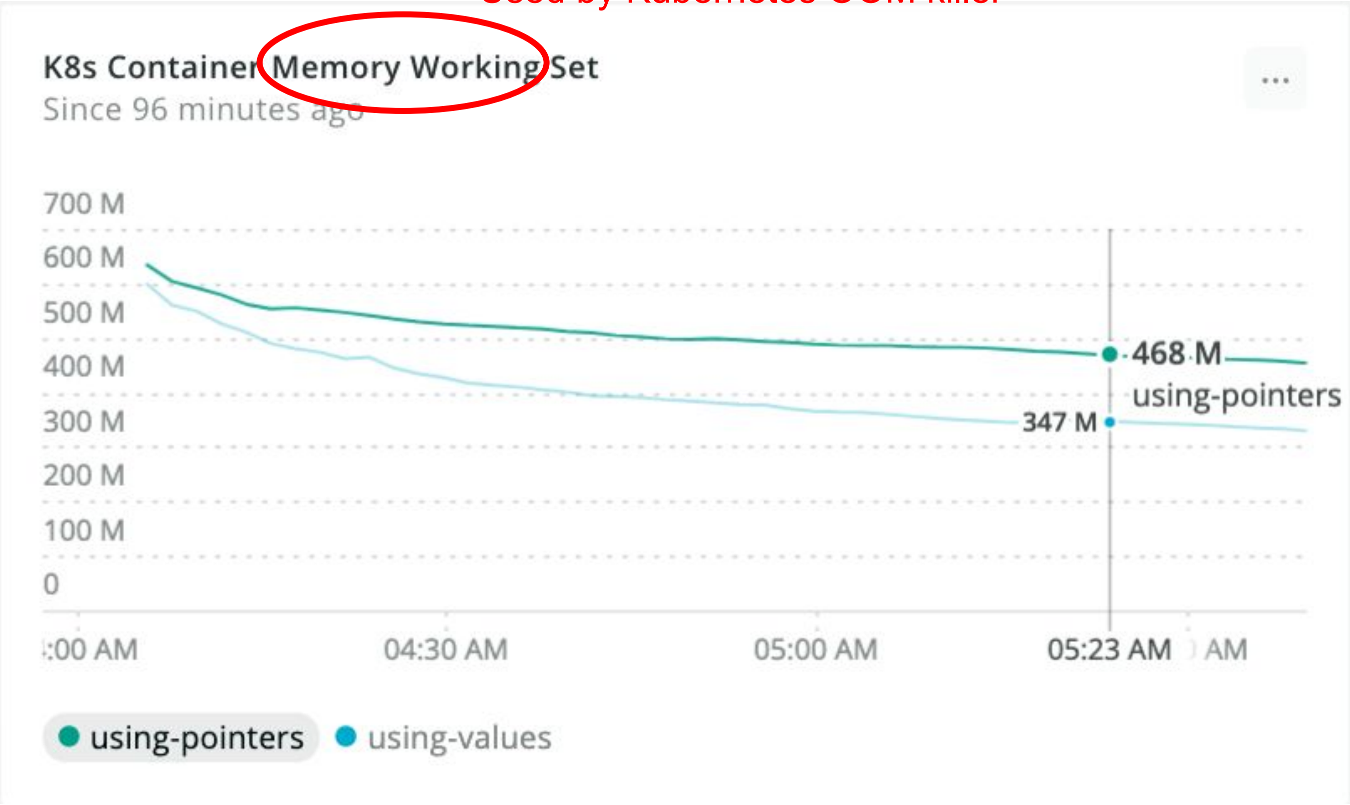
Table of contents

Running the example in Kubernetes. New Relic Metrics



Running the example in Kubernetes. New Relic Metrics

Used by Kubernetes OOM killer



With thousands of containers, this +35% might do the difference

Table of contents

Conclusions

- First, aim for clean and robust code
- If, in a hot spot, performance is so critical that you must start micro-optimizing:
  - Consider reducing your memory generation (Heap allocations) rather than the memory copy
  - Consider the memory contiguity
  - Consider adding a comment:

```
for i := range foos {
    f := &foos[i]  // DON'T CHANGE THIS!!
    sum += f.A + f.K
}
```

😅

Thank you for your attention!

Mario Macías Lloret
Senior software engineer at New Relic

Barcelona Golang Meetup
January 2020

twitter.com/MaciasUPC github.com/mariomac