Grafana Labs

Understanding Why Your CPU is Slow

Hardware Performance Insights with PerfGo

Christian Simon
Software Engineer at Grafana Labs
2026-02-01

Agenda

1. **How far CPUs have come**
   Take a look back at the CPU evolution

2. **CPU caches**
   How do CPU get access to data/instructions

3. **Branch prediction**
   Why CPU need to predict branches and how that affects your code

4. **How do I measure this?**
   How to gain insights into the CPU microarchitecture

Take a look at my previous Laptop CPUs

How far CPUs have come

Cores	2
Threads	2
Frequency	2GHz
Boost	2GHz
L1	64KB
L2	4MB (shared)
Bandwidth	10.6GB/s

2008
Intel Core 2 Duo
T7300

Cores	4
Threads	8
Frequency	1.9GHz
Turbo	4.8GHz
L1	64KB
L2	256KB
L3	8MB (shared)
Bandwidth	38.4GB/s

2020
Intel Core i7
8665U

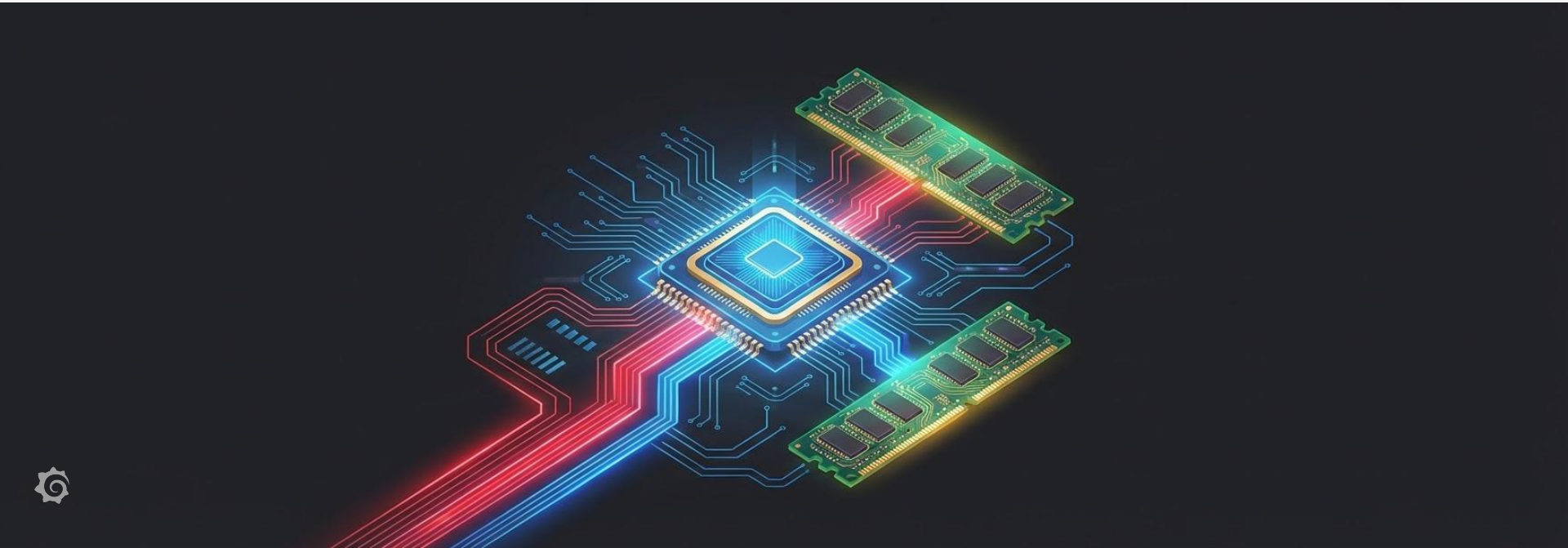
Cores	20
Threads	20
Frequency	2.4GHz
Turbo	5.2GHz
L1	192KB
L2	3MB
L3	30MB (shared)
Bandwidth	102.4GB/s

2026
Intel Core Ultra 7
255HX

What hasn't evolved as much

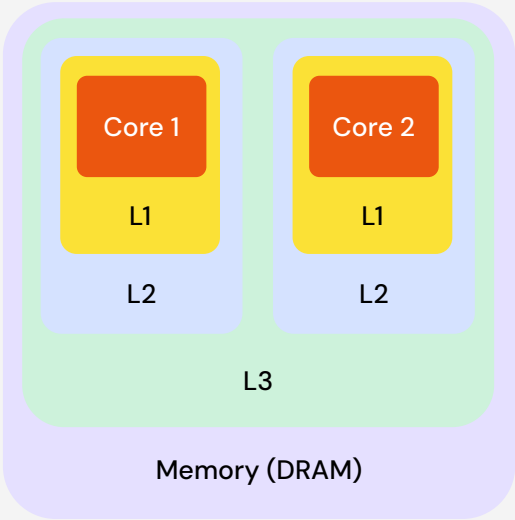
Latency to read from

CPU cache architecture



CPU cache architecture

- The further away the more cycles are needed
  - Rule of thumb: 1ns,10ns,100ns (L1, LL, Memory)
  - Last level is typically shared on multi core CPU
- Cache Lines typically 64 bytes
  - The cache line is smallest unit
- CPU tries to prefetch, data it thinks is needed soon



Cache architecture friendly code

- CPU are optimized for locality of reference, so should your code
  - Temporal locality: You accessed something once, you likely access again soon
  - Spatial locality: You accessed something at a particular address x, you are likely access another address nearby
- Move as predictable as possible through your data
- Design your data structures with that in mind

In Go: Array of Structs vs. Struct of Arrays

- Use case: Person database, where you would like to filter/aggregate information on many people.
- The AoS is already larger than a cache line
- SoA data layout will benefit SIMD: (Go 1.26 is containing a amd64 experiment)

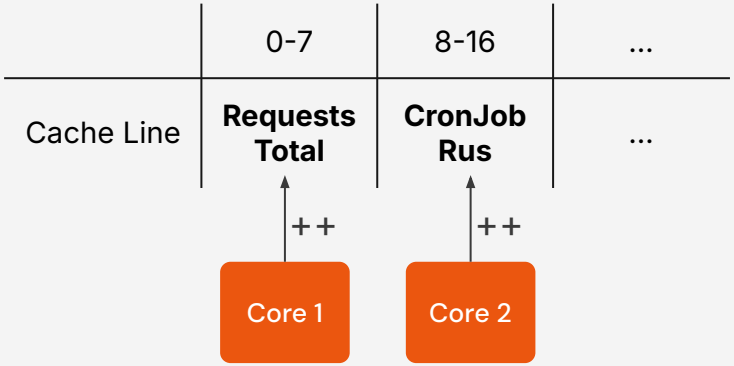
```
// Array of Structs (AoS)
type PersonDatabaseAoS []Person
type Person struct {
	Name        string
	Age         int
	Placeholder [64]byte //
padding to make struct larger
}

// Struct of Arrays (SoA)
type PersonDatabaseSoA struct {
	Names        []string
	Ages         []int
	Placeholders [][64]byte
}
```

Example data-locality

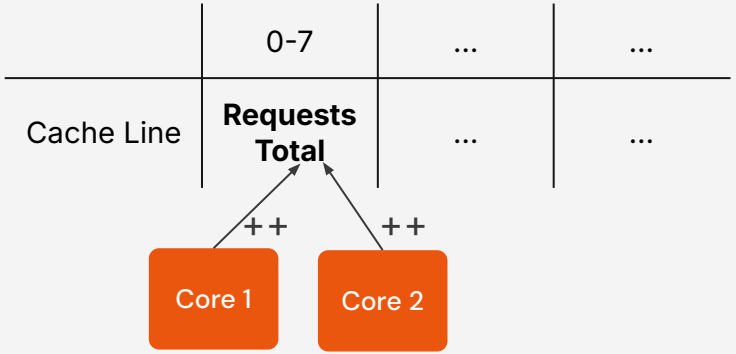
Parallelism and sharing of cache lines

- When a particular area of memory is shared between CPU cores, cache coherence protocols are used to ensure caches are up-to-date
- Sharing between cores might happen
  - Intentionally through concurrent access (true sharing)
  - Unintentionally through being located on the same cache line

False sharing



True sharing



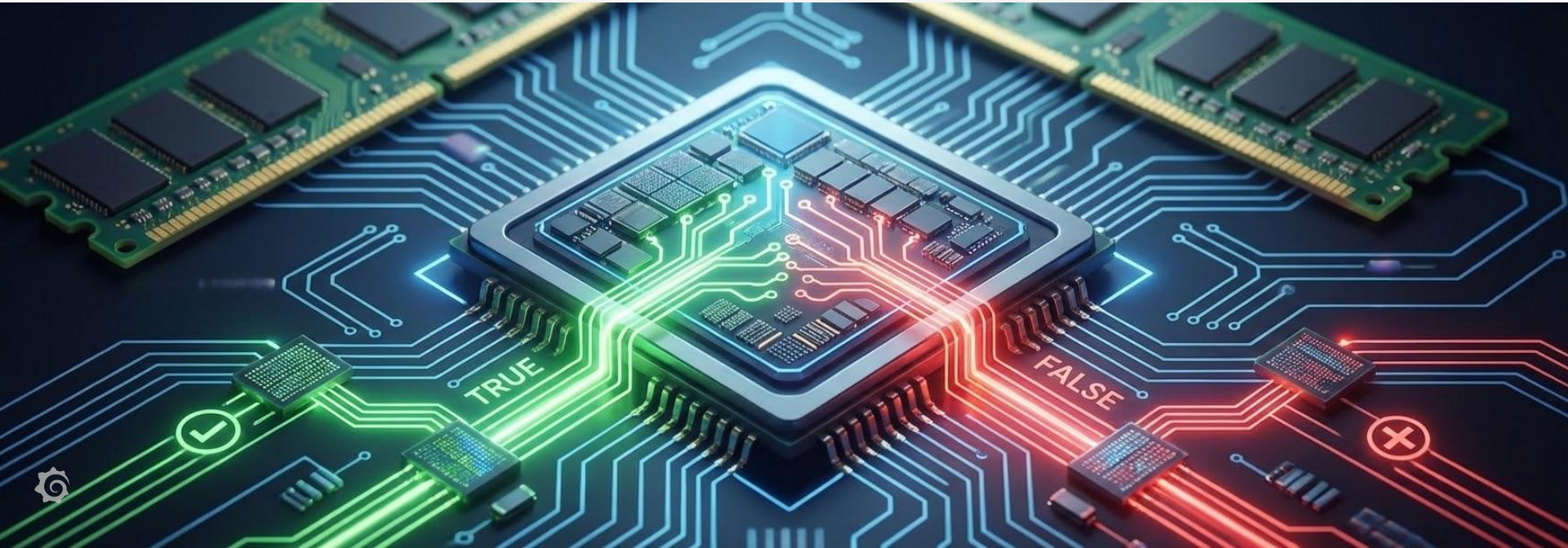
In Go: Avoid false sharing through padding

- Through padding it possible to split both Metrics on different cache lines.
- Increases memory use, so only do this after confirming the problem with measuring

```
// Metrics demonstrates false sharing
// counters are adjacent in memory
type Metrics struct {
      RequestsTotal int64
      CronJobRuns   int64
}

// PaddedMetrics uses padding to avoid
// false sharing between counters.
type PaddedMetrics struct {
      RequestsTotal int64
      _             [56]byte // Padding to
fill cache line
      CronJobRuns   int64
}
```

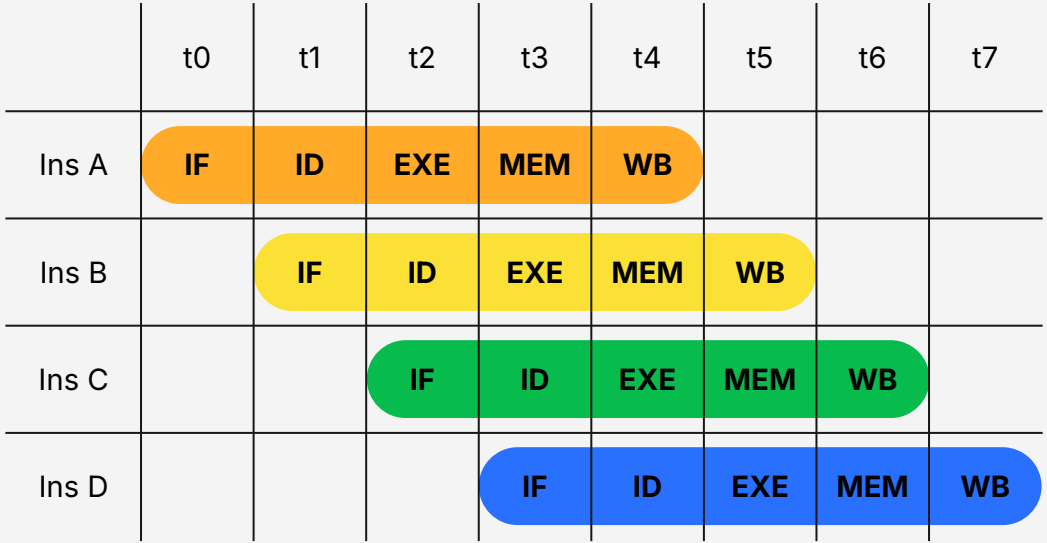
Example false-sharing

Branch prediction



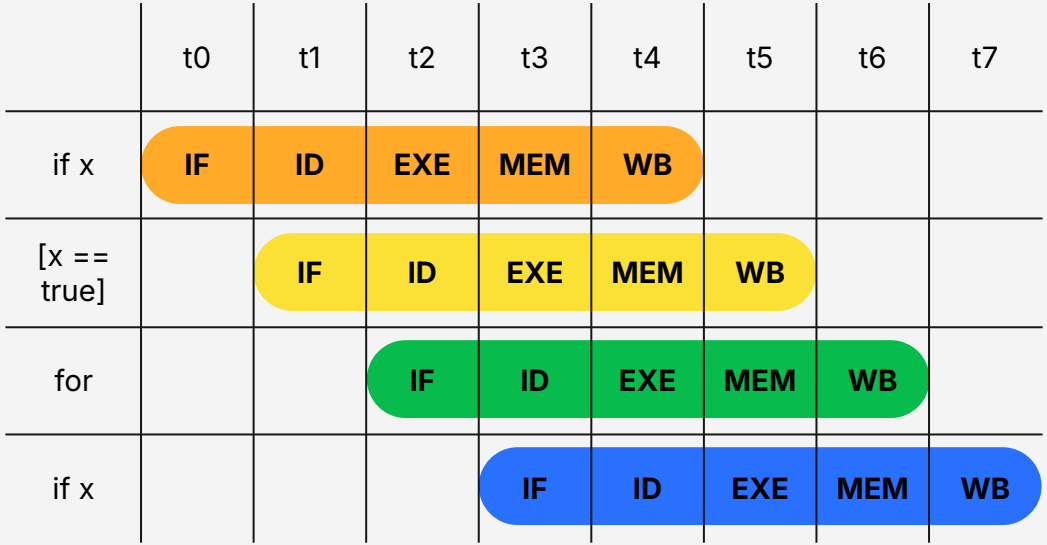
CPU pipelining

- Instructions are processed in stages
  - IF = Instruction fetch
  - ID = Instruction decode
  - EXE = Execute
  - MEM= Memory access
  - WB = Register write-back
- Better utilization of hardware
- Superscalar: Multiple processing circuits per stage
- Modern CPUs have 10-20 stages



Branch prediction

- Code with branches, limits effectiveness of pipelining
- Result not know when the instruction would have to go into the first stage
- CPUs predict which branch is likely taken
  - If prediction was wrong, revert clear pipeline and start from last good state (pipeline depth + x cycles penalty)
  - Modern CPU are quite good at predicting repetitive patterns



Branches

- Now some actual Go code that has branches
- Depending how predictable the numbers are the CPU might miss predict quite a lot branches
- Not all ifs will necessarily result in branches, so check the compilers work [example]

```
func countSignedness(numbers []int64) (pos int, neg int) {
	for _, v := range numbers {
		if v < 0 {
			neg += 1
		} else {
			pos += 1
		}
	}
	return pos, neg
}
```

Avoid branch prediction misses

- Make data more predictable (e.g. sort inputs)
- Avoid branching
  - For simple problems there might be an arithmetic solution
  - Fewer Branch misses, but extra instructions & code complexity

```
func countSignBranchless(numbers []int64) (pos int, neg int) {
	var isNeg int
	for _, v := range numbers {
		isNeg = int((v >> 63) & 1)
		neg += isNeg
		pos += 1 - isNeg
	}
	return pos, neg
}
```

Example branch-prediction

How can I measure that?

Performance Monitoring Unit (PMU)

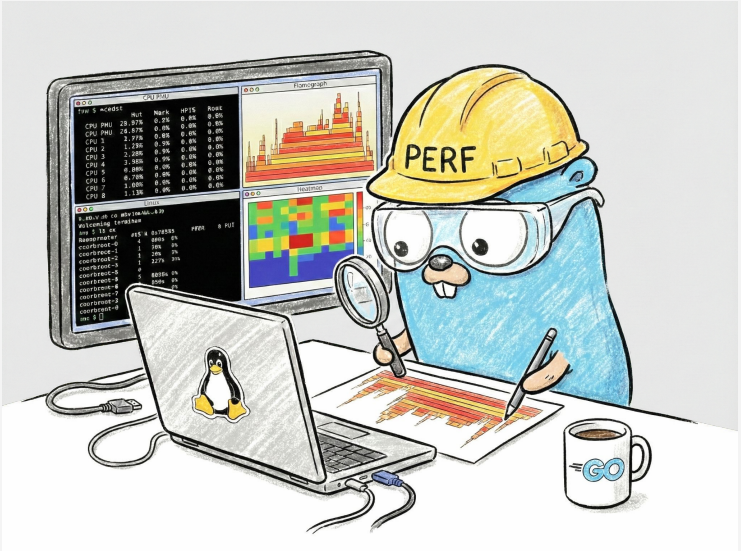
- Modern CPU have dedicated hardware counters
- Typically supported events are
  - Cycles, Instructions
  - L1/L2/L3 Cache ref and misses, read/write
  - Branches and misses
  - And many more (consult your CPU vendors technical documentation)
- Events count can also be associated with the IP
  - Depending on CPU this is subject to skid (Events attributed to nearby instruction)
  - Intel PEBS, AMD IBS
- Assume they are not available in VMs
  - Exceptions: AWS GCP


after-hashing1-v2.pb.gz:
1.7x speedup
after-hashing1-v3.pb.gz
20x speedup


WHAT SOURCE CODE
DID I USE WITH THIS
JAKE-CLARK.TUMBLR
gflip.com

Why PerfGo

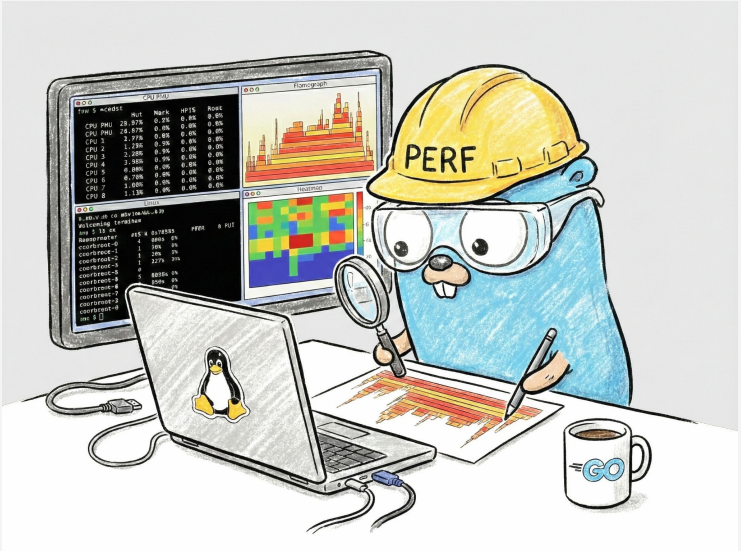
- Automate my benchmark workflow (modify, run, compare)
  - Strong link between source code - profile
  - Review
  - Easy comparison
- perf not available on my laptop, setting up addr2line properly can be painful
- pprof to visualize and compare profiles



github.com/perfgo/perfgo

PerfGo v0.1

- Test mode
  - Compiles Go test/benchmarks
  - Run them
    - PMU counter/profile collection
    - On a remote target (via SSH)
  - Store results
- Attach mode
  - Run a privileged pod with perf and collect another pods counters/profiles



github.com/perfgo/perfgo

DEMO TIME

Let’s look at perfgo in practice

Conclusion

Ensure you are CPU bound

If you applications is spending a lot of time on IO or memory allocations, fix those first

Verify your measurements

Always doubt the correctness of your assumptions and measurements

Locality of Reference

CPUs are complex, but they are optimized for locality of reference.

Evidence >> Assumptions

Performance metrics tightens the harness around AI agents (and your own assumptions)

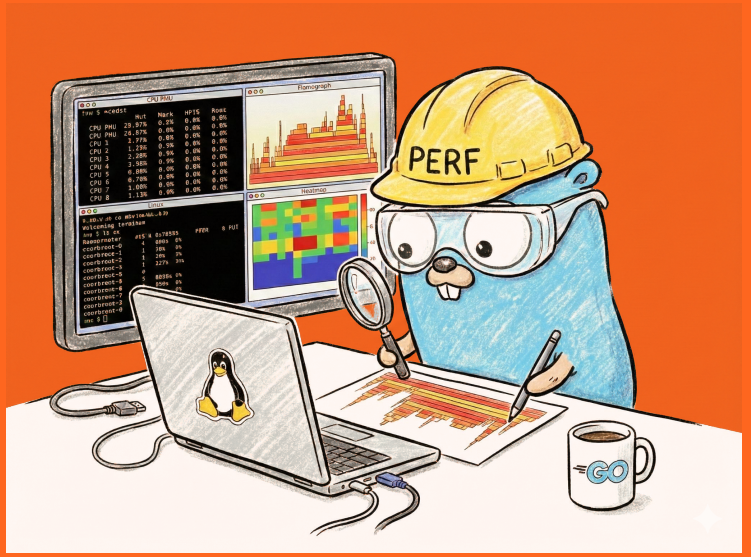
Trade offs

A lot of optimization is about trade offs, measuring performance can help to validate if past trade offs are still valid.

Try PerfGo

PerfGo might also help you bring additional insight for you benchmarks/workloads

- Learn more about this
- Try the examples



github.com/perfgo/perfgo