Metrics Matter

Debugging the Kubernetes scheduler with Prometheus

(aka, 10x improvement for Kubernetes performance)

Jonathan Boulle, CoreOS

github.com/jonboulle

@baronboulle

Why metrics?

Why "instrument all the things"?

Health monitoring with metrics

Factors externally visible to your system: request/error rates, latencies

Alerting with metrics

Time-series based alerting, same language and power as monitoring

What else can we do?

Business insight with metrics

Number of page views, capacity planning, ...

Autotuning with metrics

Automatically adjust system state based on metrics

Debugging with metrics

Instrument everything, expose deep application internals

Debugging: a case study

Improving the Kubernetes scheduler performance by 10x

Debugging: a case study

How to use Prometheus

Debugging: a case study

How to ~~not~~ use Prometheus

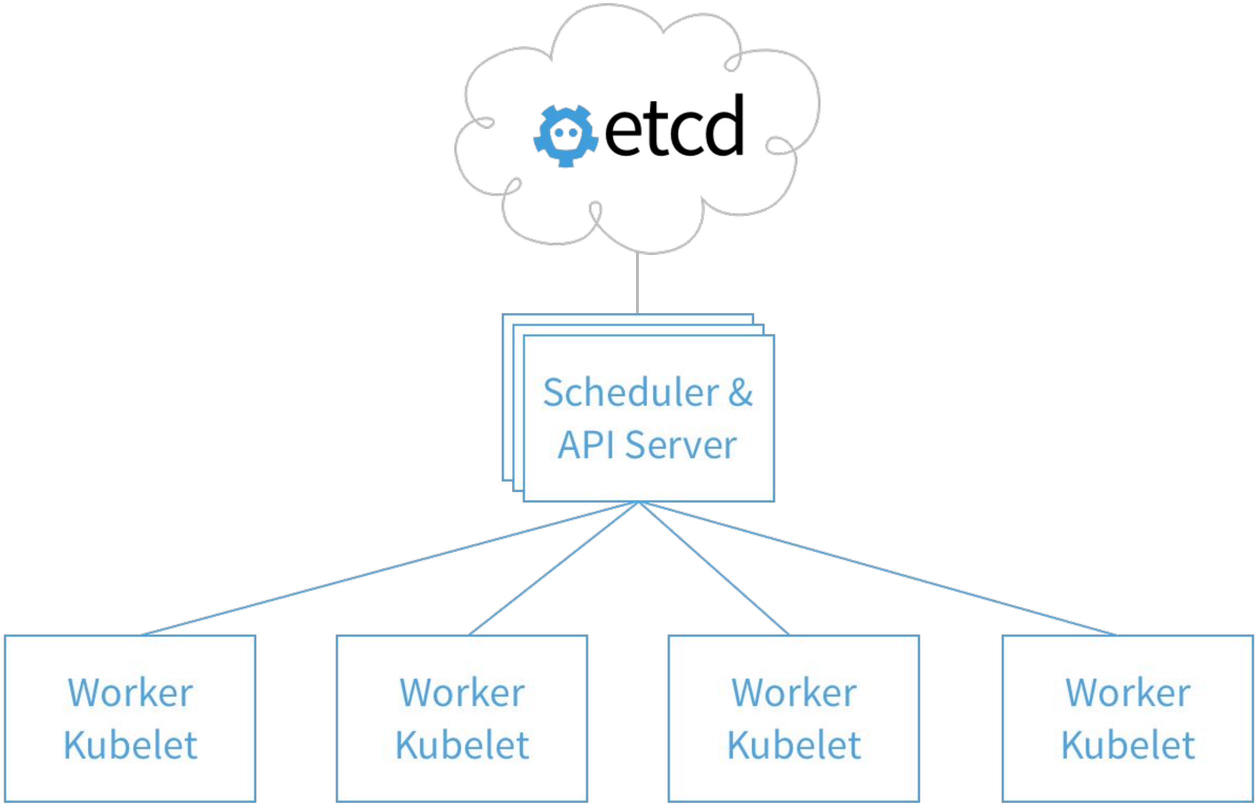
Initial goal: understand Kubernetes performance

New "Kubernetes scalability" team at CoreOS

Test and understand Kubernetes cluster performance

(ultimate goal: ensure it can scale to 10 000s of nodes)

Kubernetes architecture overview



Initial focus: API server and scheduler (control plane)

Worker nodes are relatively independent

Control plane is involved in every single pod scheduled in a cluster

Initial experiments: density tests

Kubemark: simulate the worker nodes, test large clusters

How long does it take to run 30 pods on every node in a cluster?

Start with a 100 node cluster => 3000 pods to schedule

Initial experiments: density tests

Kubemark: simulate the worker nodes, test large clusters

How long does it take to run 30 pods on every node in a cluster?

Start with a 100 node cluster => 3000 pods to schedule

```
Pods:  229 out of 3000 created,  211 running, 18 pending, 0 waiting
Pods:  429 out of 3000 created,  412 running, 17 pending, 0 waiting
Pods:  622 out of 3000 created,  604 running, 17 pending, 1 waiting
...
Pods: 2578 out of 3000 created, 2561 running, 17 pending, 0 waiting
Pods: 2779 out of 3000 created, 2761 running, 18 pending, 0 waiting
Pods: 2979 out of 3000 created, 2962 running, 16 pending, 1 waiting
Pods: 3000 out of 3000 created, 3000 running, 0 pending,  0 waiting
```

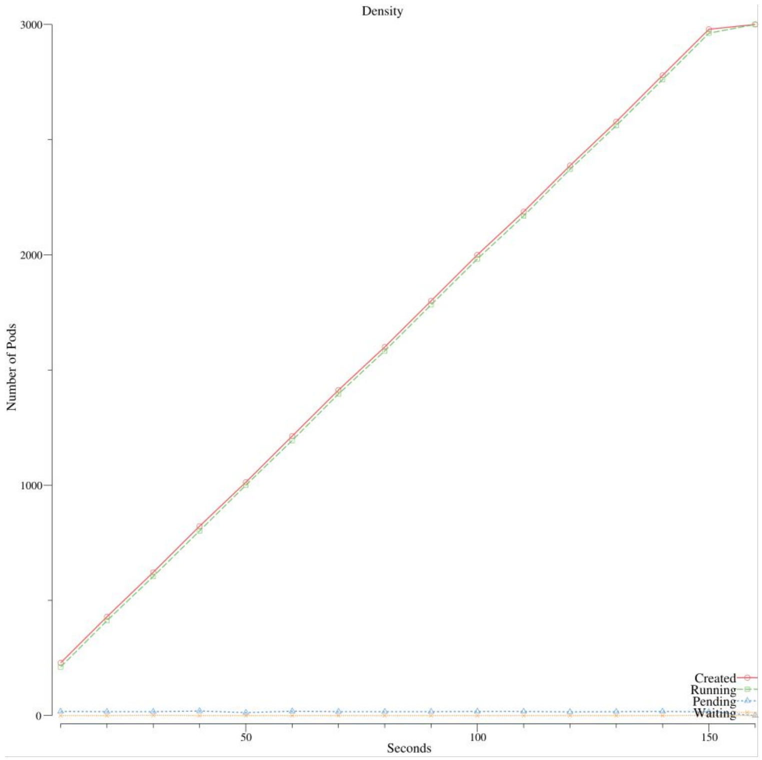
Initial results

3000 pods in ~150 seconds

Linear rate of pod **creation**
(20 pods/second)

No delay between creation+running

Suspected bottleneck in overall performance; starting point for further investigation



Initial suspect: hardware resource starvation

First place to look

Easy fix: throw more hardware at the problem!

Initial suspect: hardware resource starvation

Using 3 cores out of 16 cores

PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20	0	268m	253m	23m	S	192	0.4	4:25.43	kube-apiserver
20	0	58000	47m	15m	S	53	0.1	0:51.18	kube-scheduler
20	0	130m	118m	20m	S	17	0.2	0:27.61	kube-controller
20	0	124m	111m	10m	S	15	0.2	0:21.31	etcd

Resource usage during the test

~~Initial suspect: hardware resource starvation~~

Using 3 cores out of 16 cores

PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20	0	268m	253m	23m	S	192	0.4	4:25.43	kube-apiserver
20	0	58000	47m	15m	S	53	0.1	0:51.18	kube-scheduler
20	0	130m	118m	20m	S	17	0.2	0:27.61	kube-controller
20	0	124m	111m	10m	S	15	0.2	0:21.31	etcd

Resource usage during the test

~~Initial suspect: hardware resource starvation~~

Using 3 cores out of 16 cores

PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20	0	268m	253m	23m	S	192	0.4	4:25.43	kube-apiserver
20	0	58000	47m	15m	S	53	0.1	0:51.18	kube-scheduler
20	0	130m	118m	20m	S	17	0.2	0:27.61	kube-controller
20	0	124m	111m	10m	S	15	0.2	0:21.31	etcd

Resource usage during the test

We'll come back to this...

~~Initial suspect: hardware resource starvation~~

Bottleneck somewhere in the software

But... finding bottlenecks in a complex codebase is never trivial

(and Kubernetes is large: ~1.3M LOC)

Prometheus to the rescue!

Kubernetes provides metrics for most end-to-end API calls

Use Prometheus to explore suspicious areas - narrow down the focus of the investigation

Narrowing it down

Re-ran the Kubemark suite, monitoring scheduler's metrics

```
# HELP scheduler_e2e_scheduling_latency_microseconds E2e scheduling latency (scheduling algorithm + binding)
# TYPE scheduler_e2e_scheduling_latency_microseconds summary
scheduler_e2e_scheduling_latency_microseconds{quantile="0.5"} 23207
scheduler_e2e_scheduling_latency_microseconds{quantile="0.9"} 35112
scheduler_e2e_scheduling_latency_microseconds{quantile="0.99"} 40268
scheduler_e2e_scheduling_latency_microseconds_sum 7.1321295e+07
```

Narrowing it down

Re-ran the Kubemark suite, monitoring scheduler's metrics

```
# HELP scheduler_e2e_scheduling_latency_microseconds E2e scheduling latency (scheduling algorithm + binding)
# TYPE scheduler_e2e_scheduling_latency_microseconds summary
scheduler_e2e_scheduling_latency_microseconds{quantile="0.5"} 23207
scheduler_e2e_scheduling_latency_microseconds{quantile="0.9"} 35112
scheduler_e2e_scheduling_latency_microseconds{quantile="0.99"} 40268
scheduler_e2e_scheduling_latency_microseconds_sum 7.1321295e+07
```

Scheduler end-to-end latency ~7ms => ~140 pods/second

Implied that scheduler was not the (primary) bottleneck

Narrowing it down

Explore more metrics, logs (not enough exposed metrics!)

Thanks to log debugging, isolated contention to Replication Controller code

Iterative improvements

Removed rate limiters (protecting the API from being overwhelmed)

Fixed an inefficient code path....

Inefficient code path

```
+func init() {
+        metrics.Register()
+}
+
 // HTTPClient is an interface for testing a request objec
 type HTTPClient interface {
         Do(req *http.Request) (*http.Response, error)
@@ -109,7 +113,6 @@ type Request struct {
 // NewRequest creates a new request helper object for acce
 func NewRequest(client HTTPClient, verb string, baseURL *
         codec runtime.Codec) *Request {
-        metrics.Register()
```

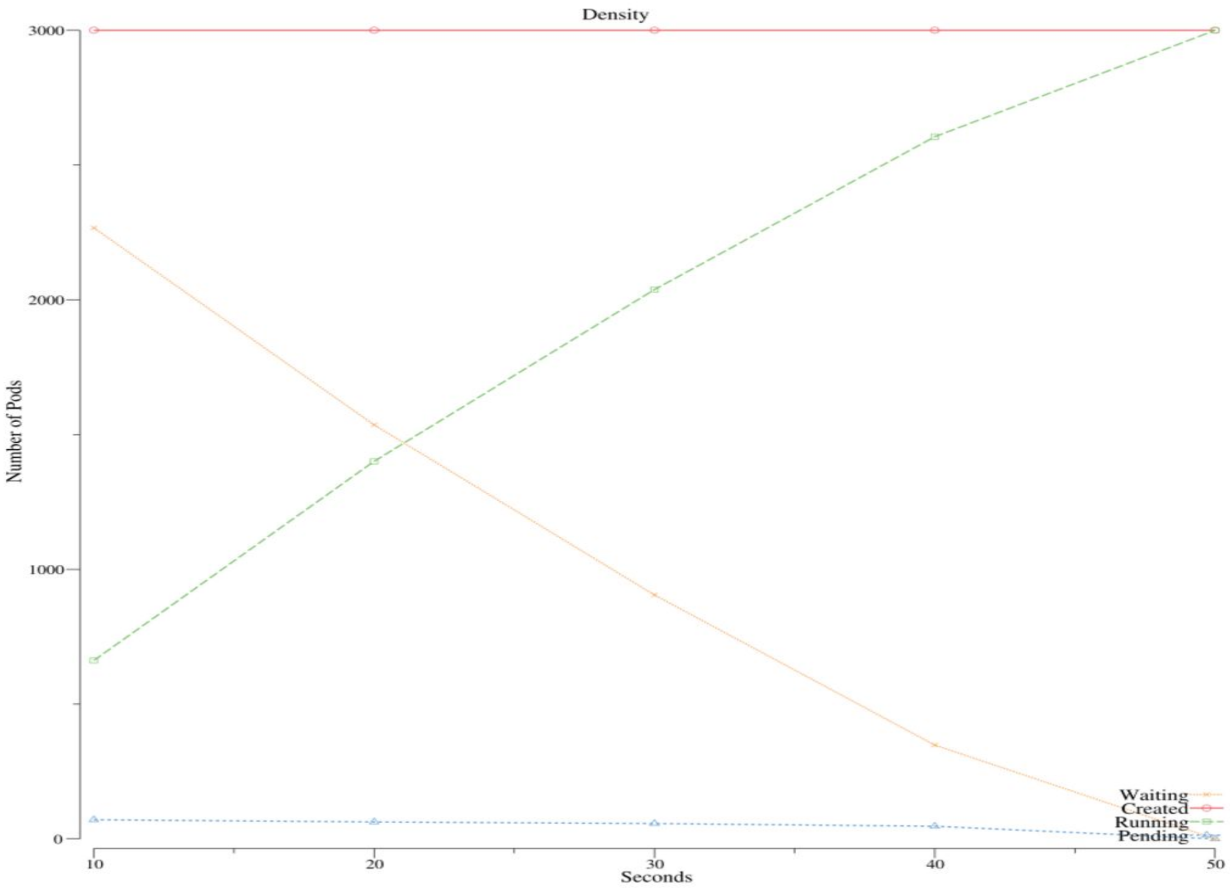
Iterative improvements

Removed rate limiters (protecting the API from being overwhelmed)

Fixed an inefficient code path....

Early success!

From **20 pods/sec** to **300+ pods/second** (100-node cluster)



Increased pod scheduling throughput from our changes

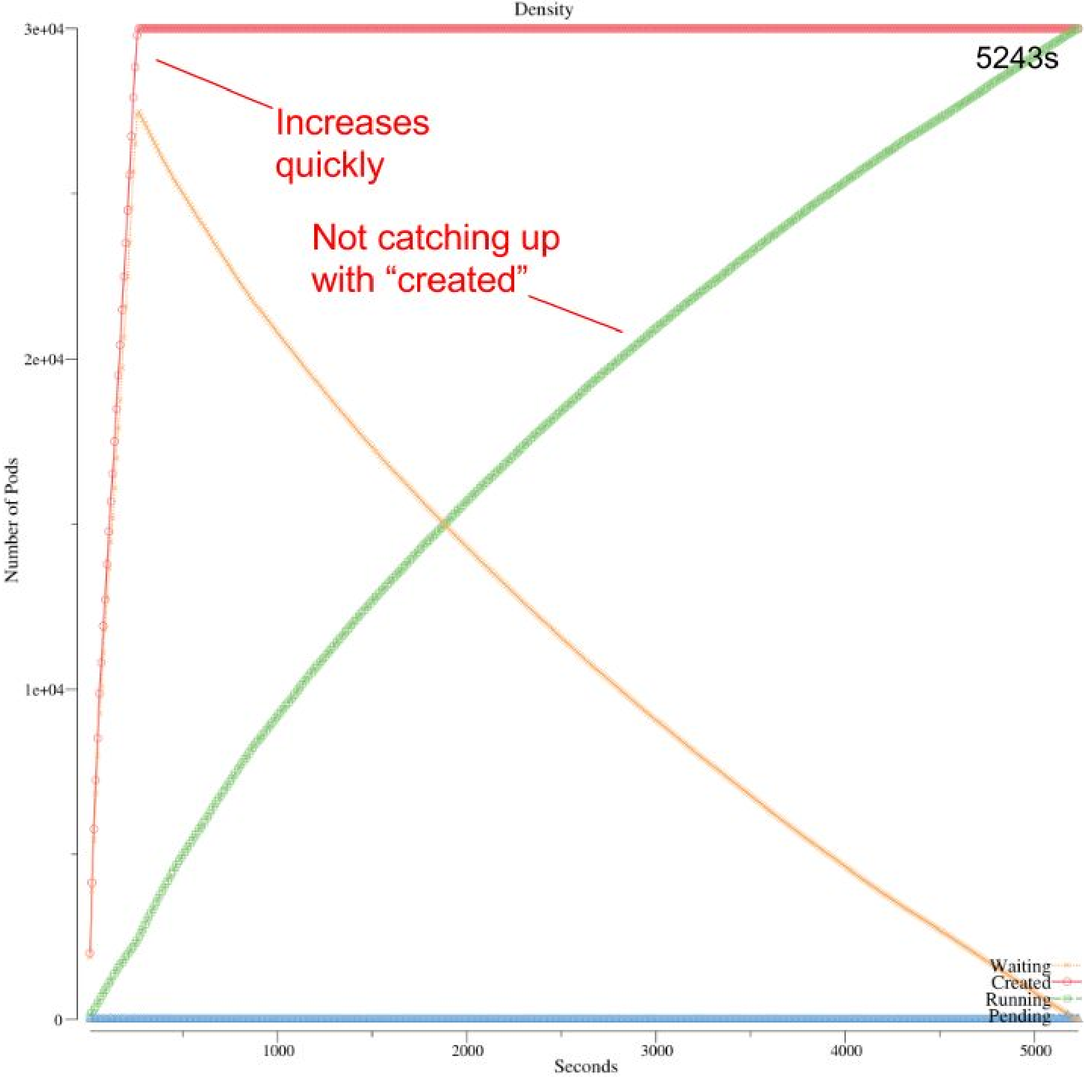
Expanding horizons

Retried tests on a 1 000-node cluster

Not so lucky: **5 243 seconds** to run **30 000 pods**

Average pod throughput of **5.72 pods/sec**

Creation rate was good, but running rate remained low



Density
5243s
Increases
quickly
Not catching up
with “created”
3e+04
2e+04
1e+04
0
Number of Pods
1000
2000
3000
4000
5000
Seconds
Waiting
Created
Running
Pending

Changing focus: the scheduler

Scheduler metrics suspicious: high initial latency (60ms), deterioriating over time (~200ms)

Scheduling latency increasing with total number of scheduled pods

Isolated to scheduler, but still in a complex codebase...

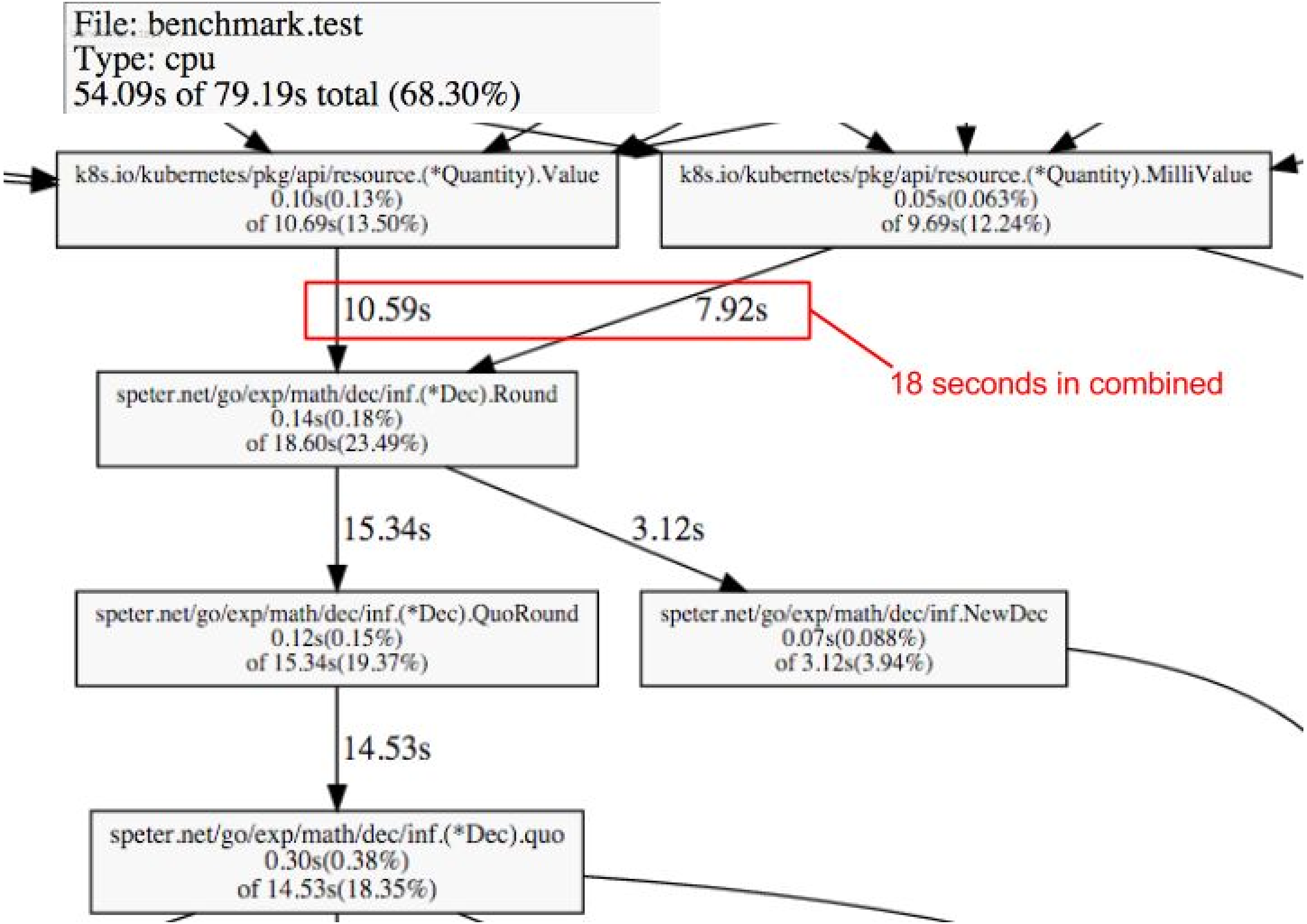
Fine-grained profiling required!

Taking a shortcut

Kubemark too slow: hours to run each test

Wrote a lightweight custom benchmarking tool

Analysed the go runtime performance with pprof


File: benchmark.test
Type: cpu
54.09s of 79.19s total (68.30%)
k8s.io/kubernetes/pkg/api/resource.(*Quantity).Value
0.10s(0.13%)
of 10.69s(13.50%)
k8s.io/kubernetes/pkg/api/resource.(*Quantity).MilliValue
0.05s(0.063%)
of 9.69s(12.24%)
10.59s
7.92s
18 seconds in combined
speter.net/go/exp/math/dec/inf.(*Dec).Round
0.14s(0.18%)
of 18.60s(23.49%)
15.34s
3.12s
speter.net/go/exp/math/dec/inf.(*Dec).QuoRound
0.12s(0.15%)
of 15.34s(19.37%)
speter.net/go/exp/math/dec/inf.NewDec
0.07s(0.088%)
of 3.12s(3.94%)
14.53s
speter.net/go/exp/math/dec/inf.(*Dec).quo
0.30s(0.38%)
of 14.53s(18.35%)

Taking a shortcut

Several issues identified, fixes proposed:

- https://github.com/kubernetes/kubernetes/issues/18126
- https://github.com/kubernetes/kubernetes/pull/18170
- https://github.com/kubernetes/kubernetes/issues/18255
- https://github.com/kubernetes/kubernetes/pull/18413
- https://github.com/kubernetes/kubernetes/issues/18831

Case study: end results

- Average scheduling latency reduced from **53 seconds** to **23 seconds** for 1 000 pods on 1 000 nodes
- Time to schedule 30 000 pods onto 1 000 nodes reduced from **8 780** seconds to **587** seconds
- 4 merged Kubernetes pull requests
- Prometheus, top, logplot, pprof, and more...

Case study: what went wrong?

Taking full(er) advantage of metrics and Prometheus

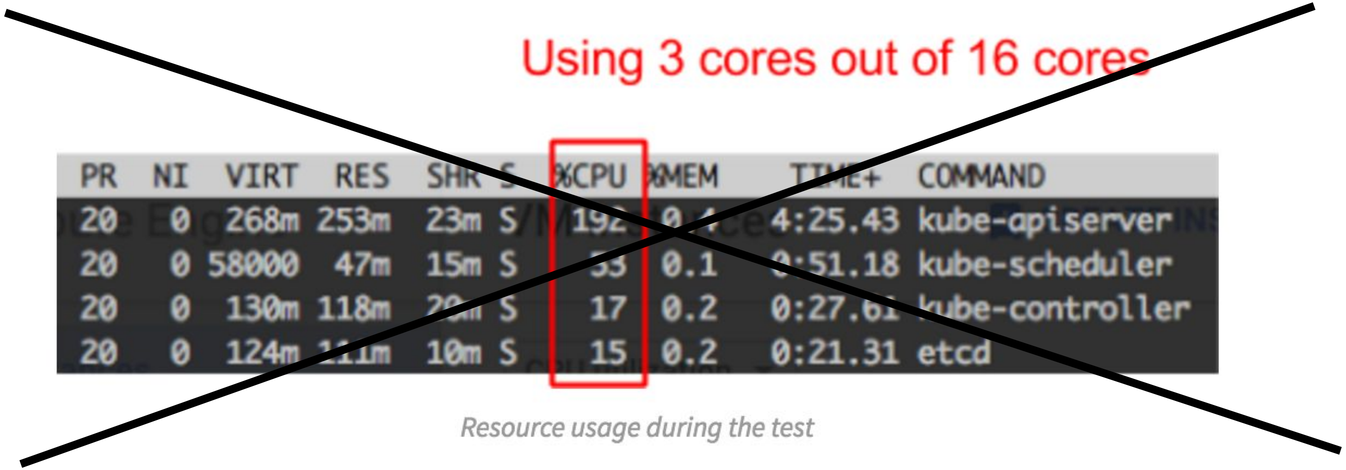
Case study: what went wrong?

Using 3 cores out of 16 cores

PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20	0	268m	253m	23m	S	192	0.4	4:25.43	kube-apiserver
20	0	58000	47m	15m	S	53	0.1	0:51.18	kube-scheduler
20	0	130m	118m	20m	S	17	0.2	0:27.61	kube-controller
20	0	124m	111m	10m	S	15	0.2	0:21.31	etcd

Resource usage during the test

Case study: what went wrong?

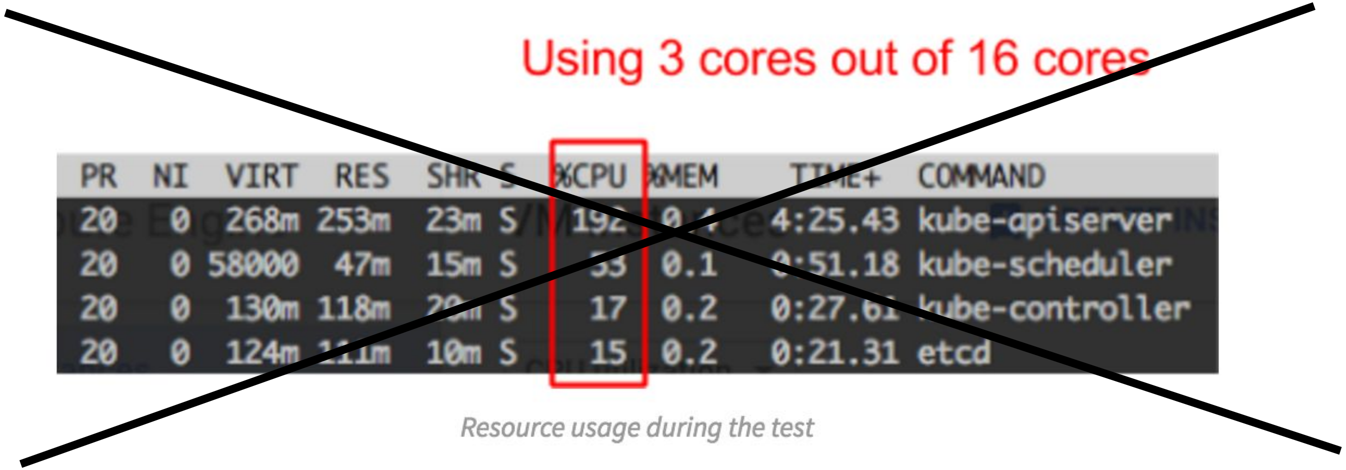


PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20	0	268m	253m	23m	S	192	[illegible]	4:25.43	kube-apiserver
20	0	58000	47m	15m	S	53	0.1	0:51.18	kube-scheduler
20	0	130m	118m	20m	S	17	0.2	0:27.61	kube-controller
20	0	124m	111m	10m	S	15	0.2	0:21.31	etcd

Resource usage during the test

Not a good example of metrics debugging!

Case study: what went wrong?



PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20	0	268m	253m	23m	S	192	0.4	4:25.43	kube-apiserver
20	0	58000	47m	15m	S	53	0.1	0:51.18	kube-scheduler
20	0	130m	118m	20m	S	17	0.2	0:27.61	kube-controller
20	0	124m	111m	10m	S	15	0.2	0:21.31	etcd

Resource usage during the test

Better: use Prometheus for system level metrics

Case study: what went wrong?

Kubemark test used log scraping + custom graphing tool

```
Pods:  229 out of 3000 created,  211 running, 18 pending, 0 waiting
Pods:  429 out of 3000 created,  412 running, 17 pending, 0 waiting
Pods:  622 out of 3000 created,  604 running, 17 pending, 1 waiting
...
Pods: 2578 out of 3000 created, 2561 running, 17 pending, 0 waiting
Pods: 2779 out of 3000 created, 2761 running, 18 pending, 0 waiting
Pods: 2979 out of 3000 created, 2962 running, 16 pending, 1 waiting
Pods: 3000 out of 3000 created, 3000 running, 0 pending,  0 waiting
```

Case study: what went wrong?

Kubemark test used log scraping + custom graphing tool

```
Pods:  229 out of 3000 created,  211 running, 18 pending, 0 waiting
Pods:  429 out of 3000 created,  412 running, 17 pending, 0 waiting
Pods:  622 out of 3000 created,  604 running, 17 pending, 1 waiting
...
Pods: 2578 out of 3000 created, 2561 running, 17 pending, 0 waiting
Pods: 2779 out of 3000 created, 2761 running, 18 pending, 0 waiting
Pods: 2979 out of 3000 created, 2962 running, 16 pending, 1 waiting
Pods: 3000 out of 3000 created, 3000 running, 0 pending,  0 waiting
```

Better: use Prometheus + Grafana (or similar)

Why are metrics better?

Simplify the lower level analysis

- Access to meaningful internal information without spelunking through code
- No need to reproduce the problem
    - No need to attach debugger
- Dynamic power of a query language
    - No static SVG graph (pprof)

Why are metrics better?

More powerful higher level analysis

- Fleet-wide view and aggregation
- Historic view
- Easy correlation with other internal and external factors over time

tl;dr

- Metrics don't provide all the answers (e.g. profiling a single function call), but in any complex system they help understand and debug behaviour and guide further investigation.

CoreOS <3

Join us!

Hiring Prometheus developers in Berlin

coreos.com/careers

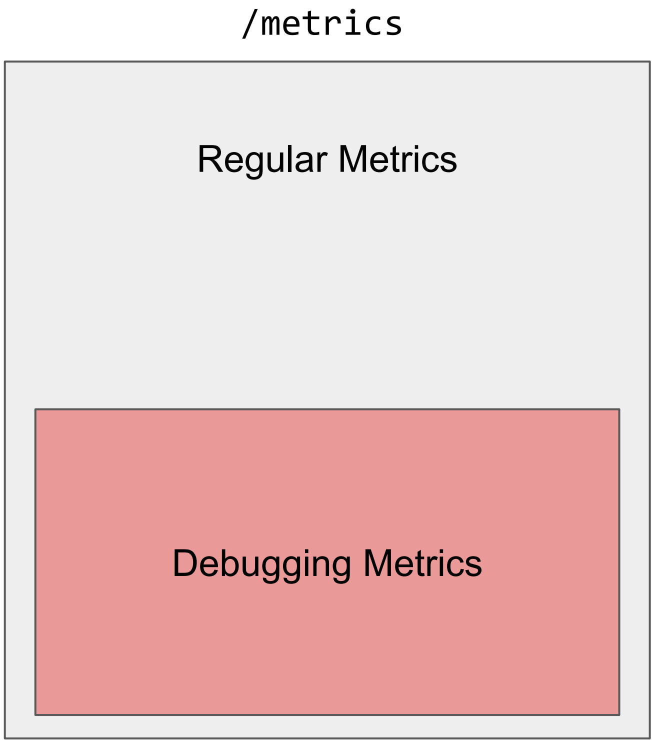
`/metrics`

Regular Metrics

`/debug_metrics`

Debugging Metrics

ingest optionally



ingest by default,

ignore optionally