Hidden Linux Metrics with ebpf_exporter

Ivan Babrou

@ibobrik

Performance team @Cloudflare

What does Cloudflare do

CDN
Moving content physically closer to visitors with our CDN.

Intelligent caching

Unlimited DDOS mitigation

Unlimited bandwidth at flat pricing with free plans

Website Optimization
Making web fast and up to date for everyone.

TLS 1.3 (with 0-RTT)

HTTP/2 + QUIC

Server push

AMP

Origin load-balancing

Smart routing

Workers

Post quantum crypto

Many more

DNS
Cloudflare is the fastest managed DNS providers in the world.

1.1.1.1

2606:4700:4700::1111

DNS over TLS

CLOUDFLARE

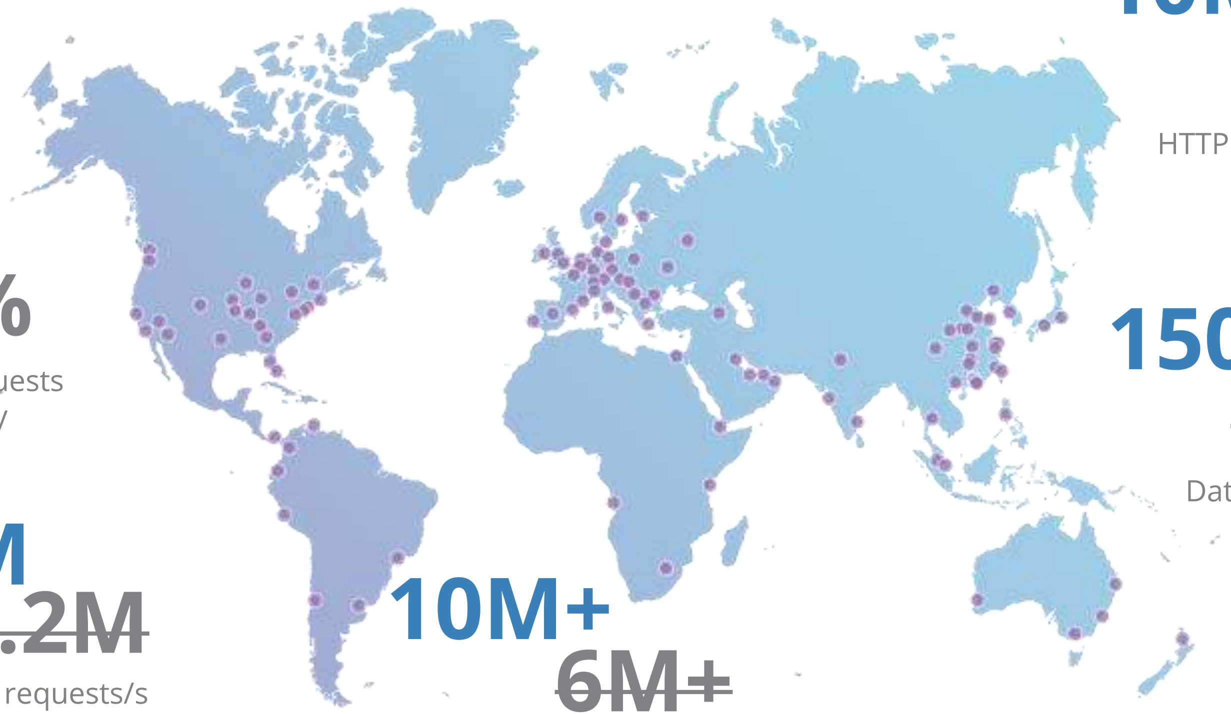
Link to slides with speaker notes

Slideshare doesn't allow links on the first 3 slides

Monitoring Cloudflare's planet-scale edge network with Prometheus

Matt Bostock from Cloudflare was here last year talking about how we use Prometheus at Cloudflare.
Check out [video] and [slides] for his presentation.

Cloudflare’s anycast network



10M
~~**5M**~~
HTTP requests/second

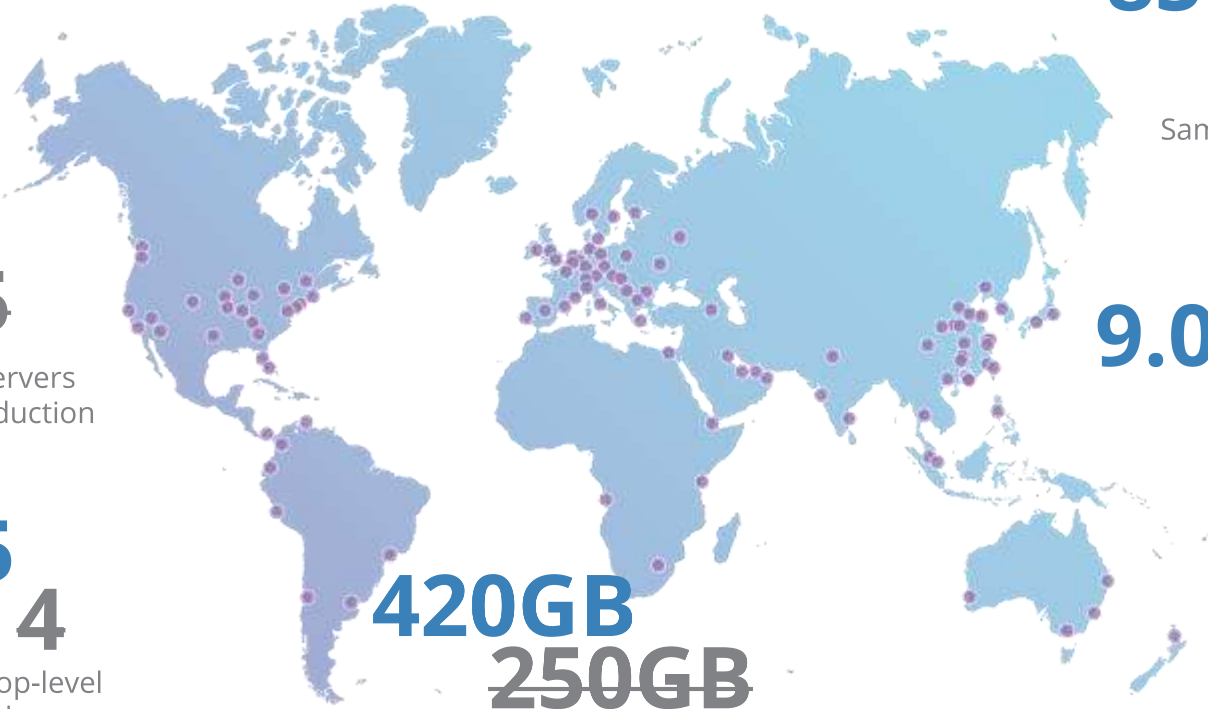
10%
Internet requests everyday

150+
~~**100+**~~
Data centers globally

2.5M
~~**1.2M**~~
DNS requests/s

10M+
~~**6M+**~~
Websites, apps & APIs in 150 countries

Cloudflare's Prometheus deployment



267
~~185~~
Prometheus servers currently in production

5
~~4~~
Top-level Prometheus servers

420GB
~~250GB~~
Max size of data on disk

85K
~~72K~~
Samples ingested per server max

9.0M
~~4.6M~~
Time-series max per server

But this is a talk about an exporter

Two main options to collect system metrics

node_exporter

Gauges and counters for system metrics with lots of plugins:

cpu, diskstats, edac, filesystem, loadavg, meminfo, netdev, etc

cAdvisor

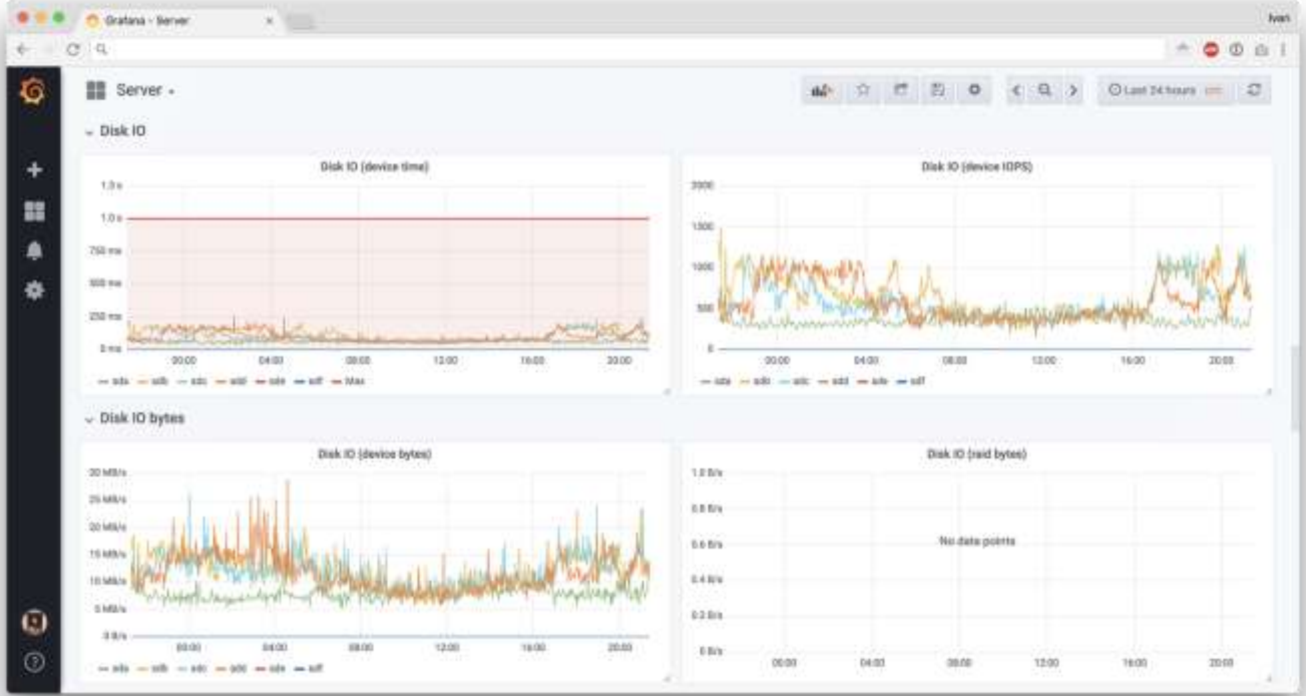
Gauges and counters for container level metrics:

cpu, memory, io, net, delay accounting, etc.

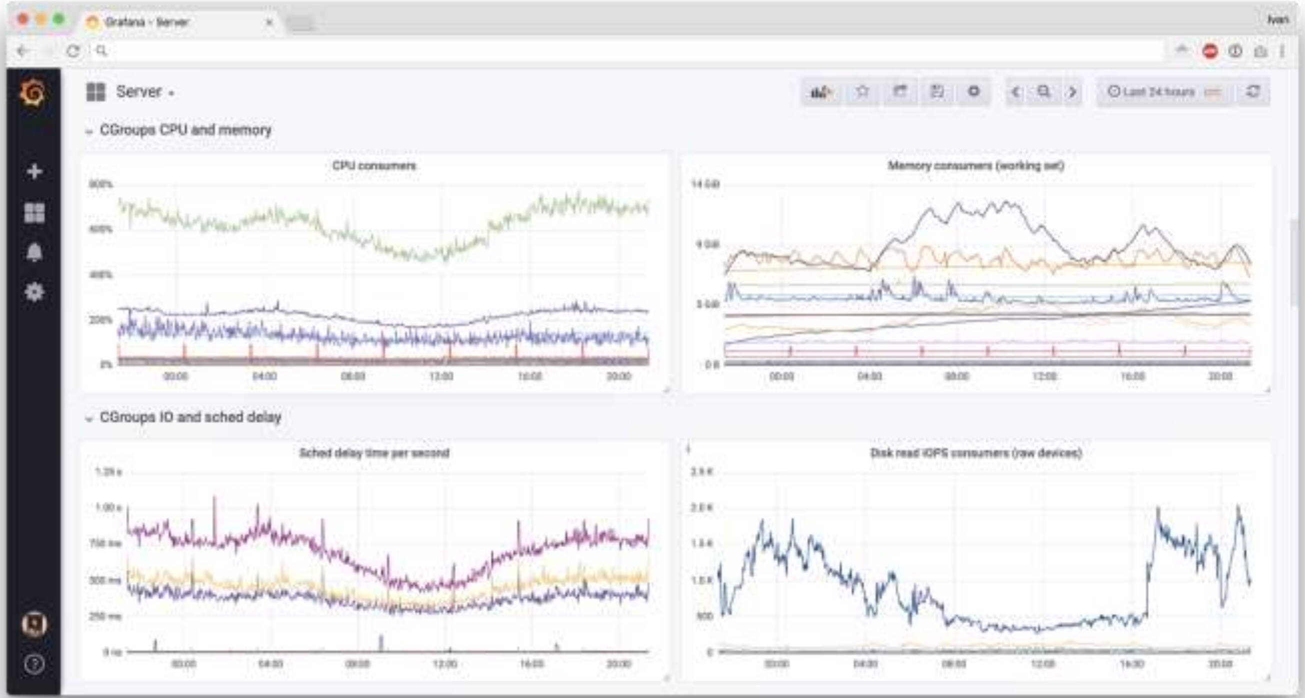
Check out [this issue] about Prometheus friendliness.

Example graphs from node_exporter

Example graphs from node_exporter



Example graphs from cAdvisor



Counters are easy, but lack detail: e.g. IO

What's the distribution?

- Many fast IOs?
- Few slow IOs?
- Some kind of mix?
- Read vs write speed?

Histograms to the rescue

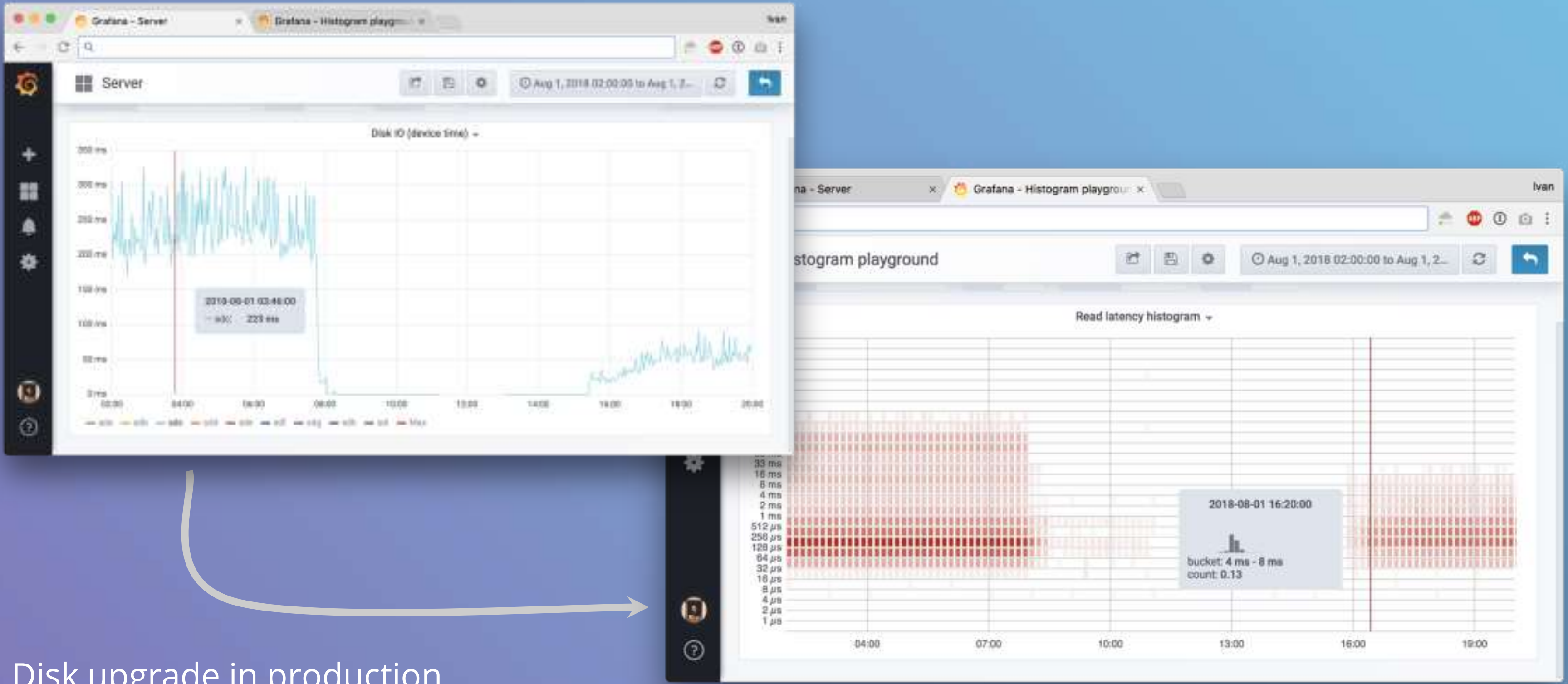
- Counter:

```
node_disk_io_time_ms{instance="foo", device="sdc"} 39251489
```

- Histogram:

```
bio_latency_seconds_bucket{instance="foo", device="sdc", le="+Inf"} 53516704
bio_latency_seconds_bucket{instance="foo", device="sdc", le="67.108864"} 53516704
...
bio_latency_seconds_bucket{instance="foo", device="sdc", le="0.001024"} 51574285
bio_latency_seconds_bucket{instance="foo", device="sdc", le="0.000512"} 46825073
bio_latency_seconds_bucket{instance="foo", device="sdc", le="0.000256"} 33208881
bio_latency_seconds_bucket{instance="foo", device="sdc", le="0.000128"} 9037907
bio_latency_seconds_bucket{instance="foo", device="sdc", le="6.4e-05"} 239629
bio_latency_seconds_bucket{instance="foo", device="sdc", le="3.2e-05"} 132
bio_latency_seconds_bucket{instance="foo", device="sdc", le="1.6e-05"} 42
bio_latency_seconds_bucket{instance="foo", device="sdc", le="8e-06"} 29
bio_latency_seconds_bucket{instance="foo", device="sdc", le="4e-06"} 2
bio_latency_seconds_bucket{instance="foo", device="sdc", le="2e-06"} 0
```

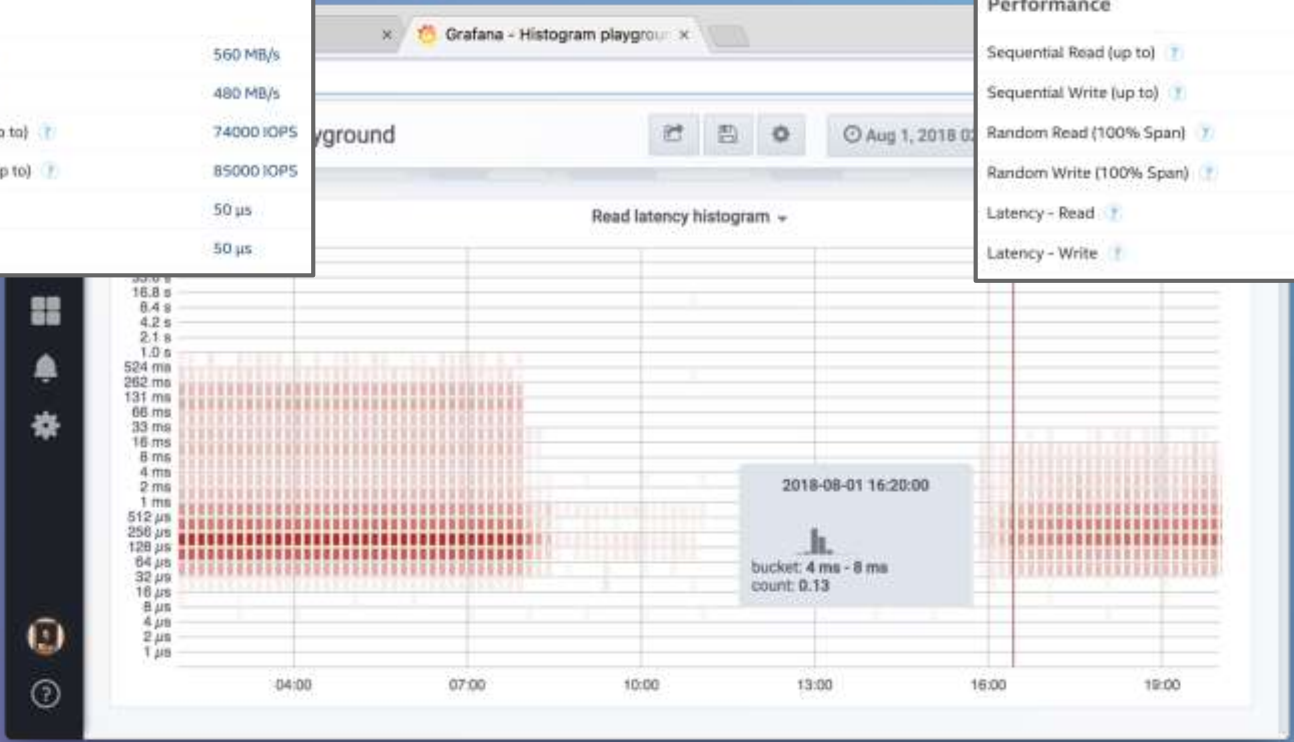
Can be nicely visualized with new Grafana



Disk upgrade in production

Larger view to see in detail

So much for holding up to spec



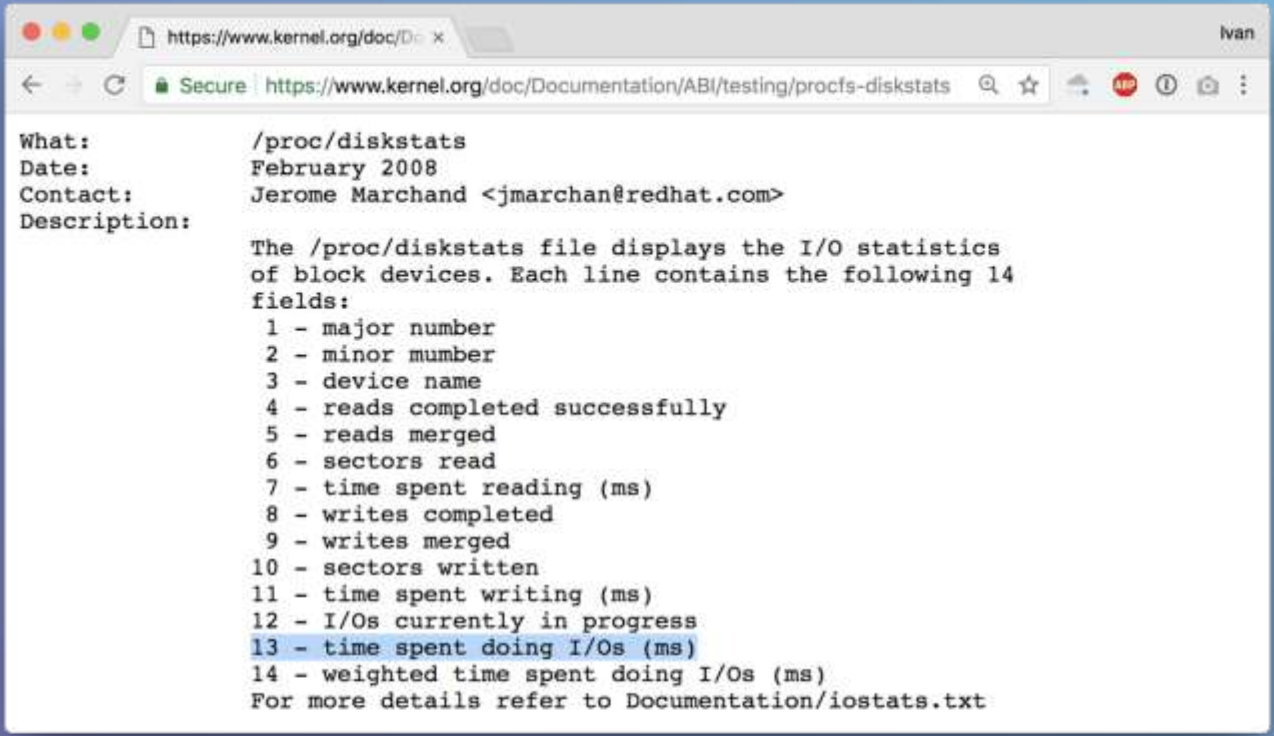
Performance

Sequential Read (up to)	560 MB/s
Sequential Write (up to)	480 MB/s
Random Read (8GB Span) (up to)	74000 IOPS
Random Write (8GB Span) (up to)	85000 IOPS
Latency - Read	50 µs
Latency - Write	50 µs

Performance

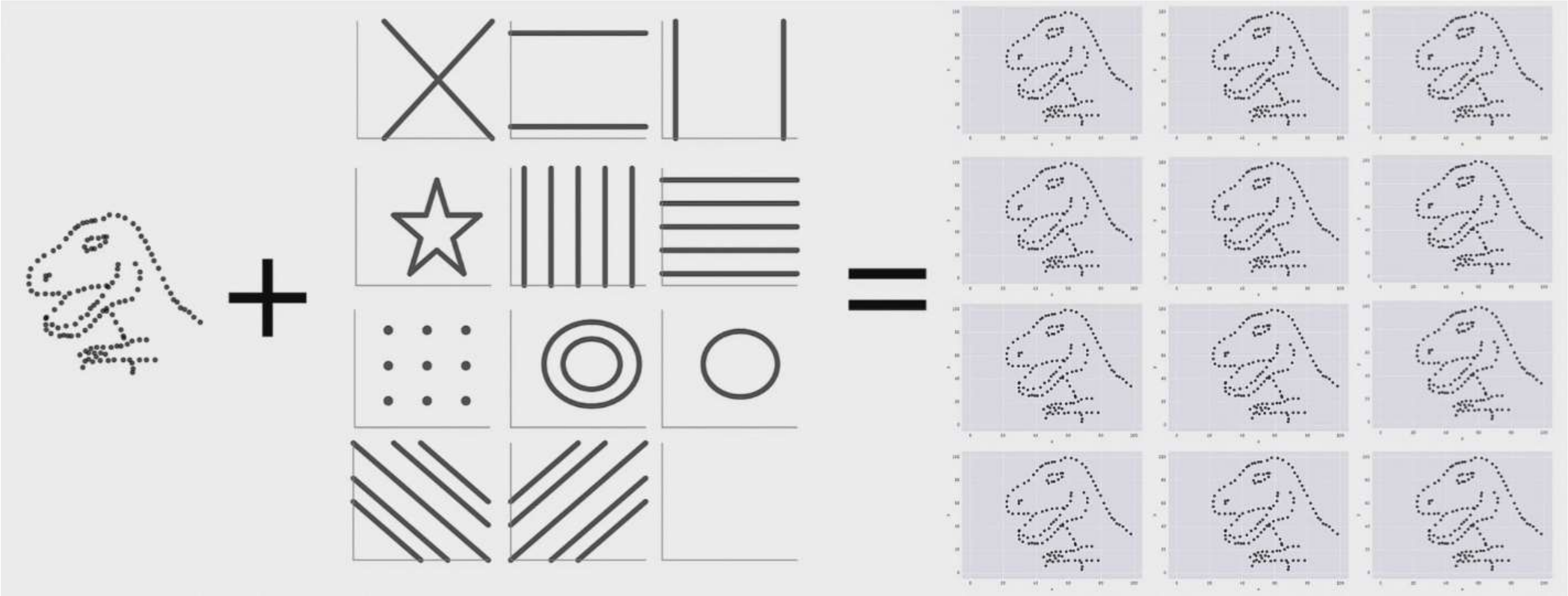
Sequential Read (up to)	500 MB/s
Sequential Write (up to)	330 MB/s
Random Read (100% Span)	72000 IOPS
Random Write (100% Span)	20000 IOPS
Latency - Read	36 µs
Latency - Write	36 µs

Linux kernel only gives you counters



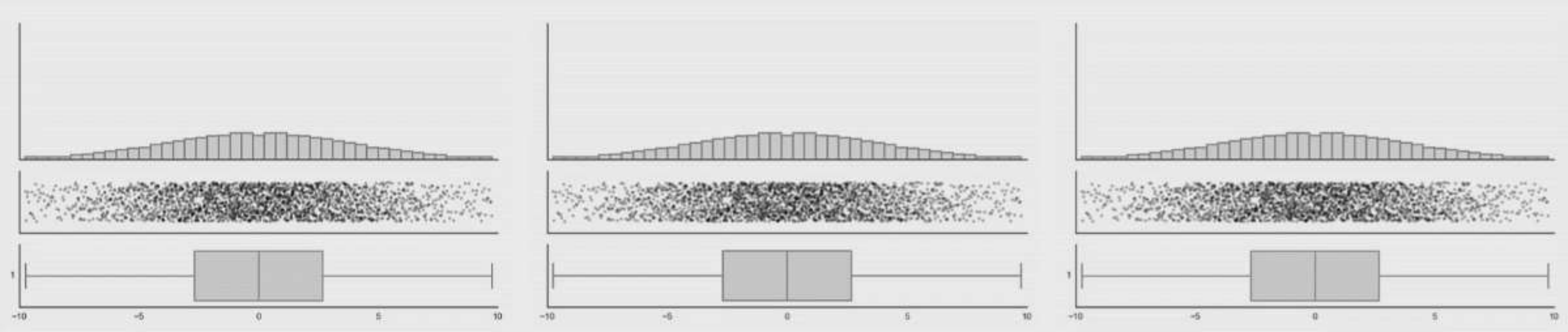

```
What:           /proc/diskstats
Date:           February 2008
Contact:        Jerome Marchand <jmarchan@redhat.com>
Description:
                The /proc/diskstats file displays the I/O statistics
                of block devices. Each line contains the following 14
                fields:
                 1 - major number
                 2 - minor mumber
                 3 - device name
                 4 - reads completed successfully
                 5 - reads merged
                 6 - sectors read
                 7 - time spent reading (ms)
                 8 - writes completed
                 9 - writes merged
                10 - sectors written
                11 - time spent writing (ms)
                12 - I/Os currently in progress
                13 - time spent doing I/Os (ms)
                14 - weighted time spent doing I/Os (ms)
                For more details refer to Documentation/iostats.txt
```

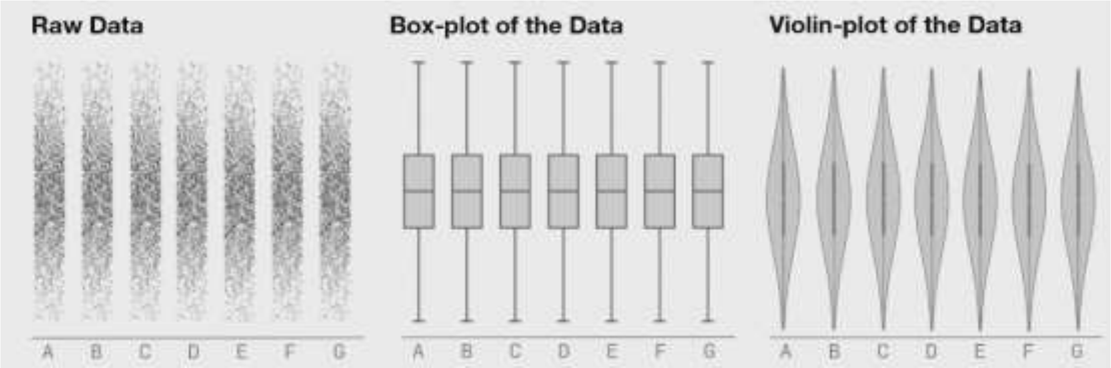
Autodesk research: Datasaurus (animated)



CLOUDFLARE

Autodesk research: Datasaurus (animated)





CLOUDFLARE

You need individual events for histograms

- Solution has to be low overhead (no blktrace)
- Solution has to be universal (not just IO tracing)
- Solution has to be supported out of the box (no modules or patches)
- Solution has to be safe (no kernel crashes or loops)

Enter eBPF

Low overhead sandboxed user-defined bytecode running in kernel.

It can never crash, hang or interfere with the kernel negatively.

If you run Linux 4.1 (June 2015) or newer, you already have it.

Great intro from Brendan Gregg: http://www.brendangregg.com/ebpf.html

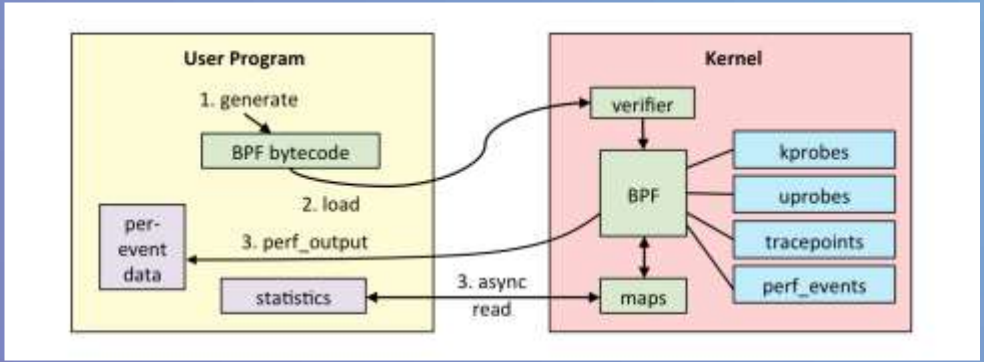
BPF and XDP reference: https://cilium.readthedocs.io/en/v1.1/bpf/

It’s a bytecode you don’t have to write

```
 0:   79 12 20 00 00 00 00 00   r2 = *(u64 *)(r1 + 32)
 1:   15 02 03 00 57 00 00 00   if r2 == 87 goto +3
 2:   b7 02 00 00 00 00 00 00   r2 = 0
 3:   79 11 28 00 00 00 00 00   r1 = *(u64 *)(r1 + 40)
 4:   55 01 01 00 57 00 00 00   if r1 != 87 goto +1
 5:   b7 02 00 00 01 00 00 00   r2 = 1
 6:   7b 2a f8 ff 00 00 00 00   *(u64 *)(r10 - 8) = r2
 7:   18 11 00 00 03 00 00 00 00 00 00 00 00 00 00 00   ld_pseudo   r1, 1, 3
 9:   bf a2 00 00 00 00 00 00   r2 = r10
10:   07 02 00 00 f8 ff ff ff   r2 += -8
11:   85 00 00 00 01 00 00 00   call 1
12:   15 00 04 00 00 00 00 00   if r0 == 0 goto +4
13:   79 01 00 00 00 00 00 00   r1 = *(u64 *)(r0 + 0)
14:   07 01 00 00 01 00 00 00   r1 += 1
15:   7b 10 00 00 00 00 00 00   *(u64 *)(r0 + 0) = r1
16:   05 00 0a 00 00 00 00 00   goto +10
17:   b7 01 00 00 01 00 00 00   r1 = 1
18:   7b 1a f0 ff 00 00 00 00   *(u64 *)(r10 - 16) = r1
```

eBPF in a nutshell

- You can write small C programs that attach to kernel functions
  - Max 4096 instructions, 512B stack, in-kernel JIT for opcodes
  - Verified and guaranteed to terminate
  - No crossing of kernel / user space boundary
- You can use maps to share data with these programs (extract metrics)



BCC takes care of compiling C (dcstat)

```
int count_lookup(struct pt_regs *ctx) {                         // runs after d_lookup kernel function
    struct key_t key = { .op = S_SLOW };

    bpf_get_current_comm(&key.command, sizeof(key.command));  // helper function to get current command

    counts.increment(&key);                                 // update map you can read from userspace

    if (PT_REGS_RC(ctx) == 0) {
        key.op = S_MISS;
        val = counts.increment(&key);                         // update another key if it's a miss
    }

    return 0;
}
```

CLOUDFLARE

BCC has bundled tools: biolatency

```
$ sudo /usr/share/bcc/tools/biolatency
Tracing block device I/O... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 3        |                                        |
        32 -> 63         : 14       |*                                       |
        64 -> 127        : 107      |********                                |
       128 -> 255        : 525      |****************************************|
       256 -> 511        : 68       |*****                                   |
       512 -> 1023       : 10       |                                        |
```

CLOUDFLARE

BCC has bundled tools: execsnoop

```
# execsnoop
PCOMM        PID    RET ARGS
bash         15887    0 /usr/bin/man ls
preconv      15894    0 /usr/bin/preconv -e UTF-8
man          15896    0 /usr/bin/tbl
man          15897    0 /usr/bin/nroff -mandoc -rLL=169n -rLT=169n -Tutf8
man          15898    0 /usr/bin/pager -s
nroff        15900    0 /usr/bin/locale charmap
nroff        15901    0 /usr/bin/groff -mtty-char -Tutf8 -mandoc -rLL=169n -rLT=169n
groff        15902    0 /usr/bin/troff -mtty-char -mandoc -rLL=169n -rLT=169n -Tutf8
groff        15903    0 /usr/bin/grotty
```

CLOUDFLARE

BCC has bundled tools: ext4slower

```
# ext4slower 1
Tracing ext4 operations slower than 1 ms
TIME     COMM           PID    T BYTES   OFF_KB   LAT(ms) FILENAME
06:49:17 bash           3616   R 128     0           7.75 cksum
06:49:17 cksum          3616   R 39552   0           1.34 [
06:49:17 cksum          3616   R 96      0           5.36 2to3-2.7
06:49:17 cksum          3616   R 96      0          14.94 2to3-3.4
06:49:17 cksum          3616   R 10320   0           6.82 411toppm
06:49:17 cksum          3616   R 65536   0           4.01 a2p
06:49:17 cksum          3616   R 55400   0           8.77 ab
06:49:17 cksum          3616   R 36792   0          16.34 aclocal-1.14
06:49:17 cksum          3616   R 15008   0          19.31 acpi_listen
06:49:17 cksum          3616   R 6123    0          17.23 add-apt-repository
06:49:17 cksum          3616   R 6280    0          18.40 addpart
```

Making use of all that with ebpf_exporter

- Many BCC tools make sense as metrics, so let’s use that
- Exporter compiles user-defined BCC programs and loads them
- Programs run in the kernel and populate maps
- During scrape exporter pulls all maps and transforms them:
    - Map keys to labels (disk name, function name, cpu number)
    - Map values to metric values
    - There are no float values in eBPF

Getting timer counters into Prometheus

```
code: |
  BPF_HASH(counts, u64);

  // Generates tracepoint__timer__timer_start
  TRACEPOINT_PROBE(timer, timer_start) {
      counts.increment((u64) args->function);
      return 0;
  }
```

Code to run in the kernel
and populate the map

```
metrics:
  counters:
    - name: timer_start_total
      help: Timers fired in the kernel
      table: counts
      labels:
        - name: function
          size: 8
          decoders:
            - name: ksym
tracepoints:
  timer:timer_start: tracepoint__timer__timer_start
```

How to turn map into metrics
readable by Prometheus

Getting timer counters into Prometheus

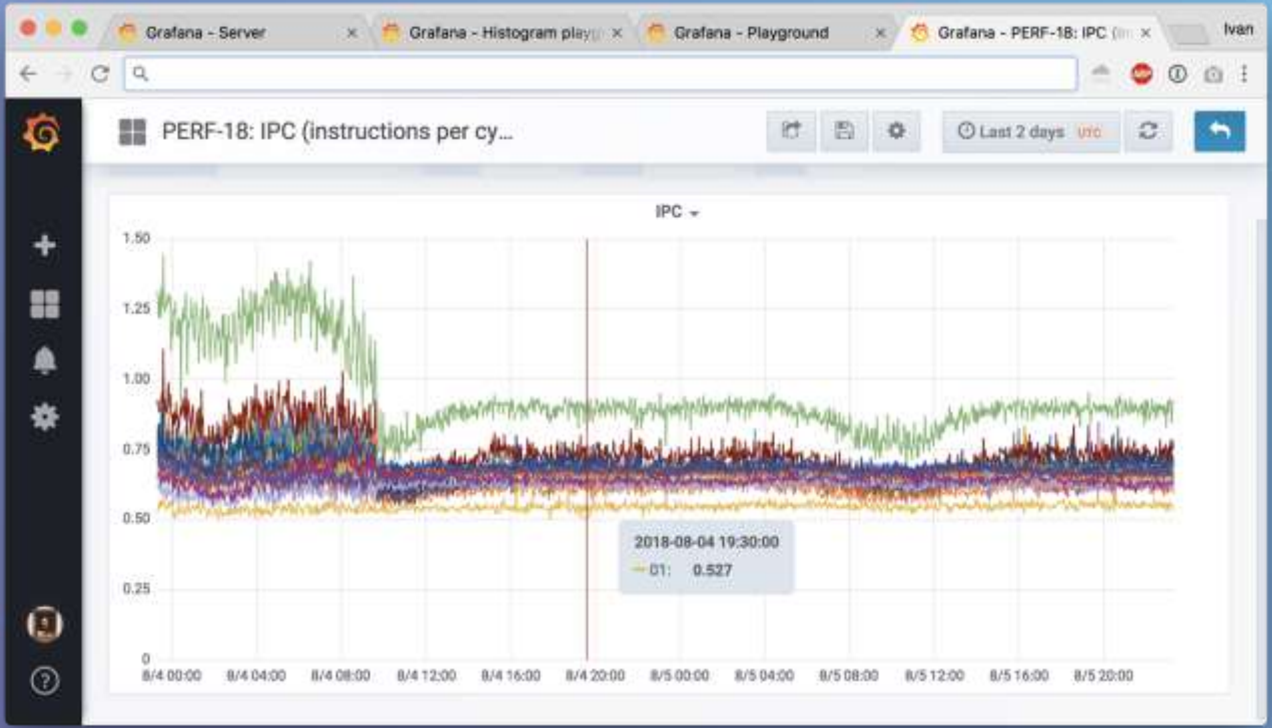
Why can timers be useful?

See Cloudflare blog post: “Tracing System CPU on Debian Stretch”.

TL;DR: Debian upgrade triggered systemd bug where it broke TCP segmentation offload, which increased CPU load 5x and introduced lots of interesting side effects up to memory allocation stalls.

If we had timer metrics enabled, we would have seen this sooner.

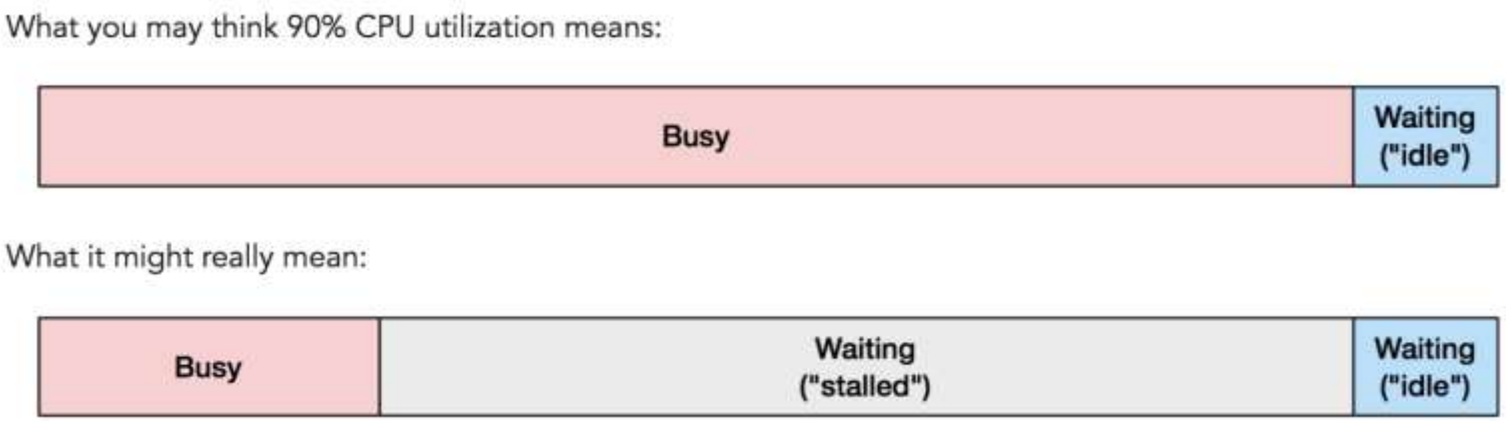
Other bundled examples: IPC



Why can instructions per cycle be useful?

See Brendan Gregg's blog post: "[CPU Utilization is Wrong]".

TL;DR: same CPU% may mean different throughput in terms of CPU work done. IPC helps to understand the workload better.



CLOUDFLARE

Other bundled examples: LLC (L3 Cache)

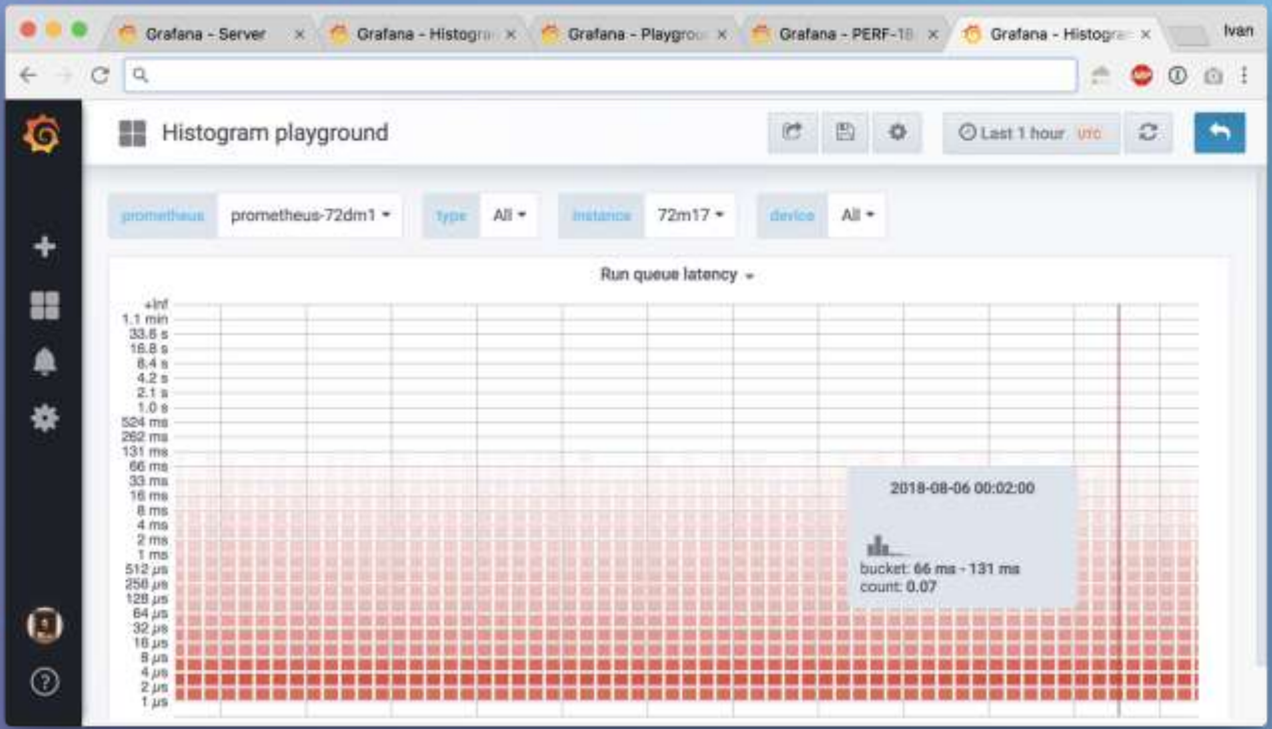
Why can LLC hit rate be useful?

You can answer questions like:

- Do I need to pay more for a CPU with bigger L3 cache?
- How does having more cores affect my workload?

LLC hit rate usually follows IPC patterns as well.

Other bundled examples: run queue delay



Why can run queue latency be useful?

See: "perf sched for Linux CPU scheduler analysis" by Brendan G.

You can see how contended your system is, how effective is the scheduler and how changing sysctls can affect that.

It's surprising how high delay is by default.

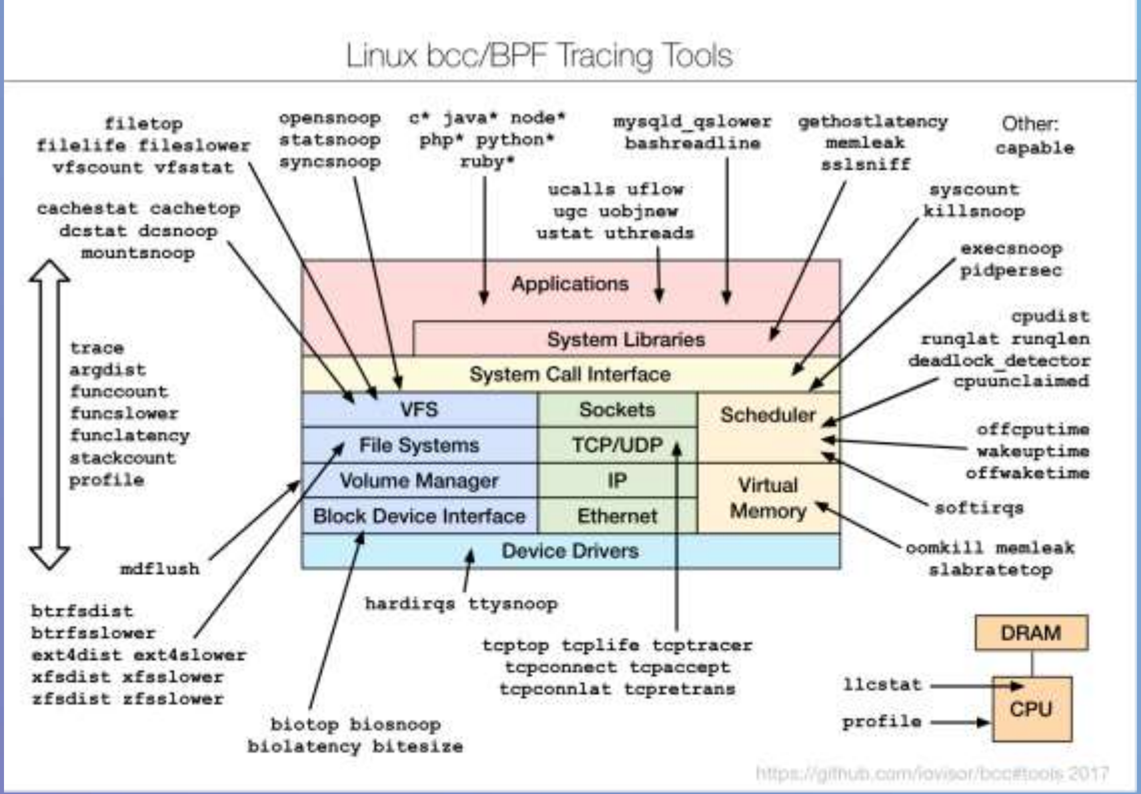
From Scylla: "Reducing latency spikes by tuning the CPU scheduler".

How many things can you measure?

Numbers are from a production Cloudflare machine running Linux 4.14:

- 501 hardware events and counters:
  - sudo perf list hw cache pmu | grep '^ [a-z]' | wc -l
- 1853 tracepoints:
  - sudo perf list tracepoint | grep '^ [a-z]' | wc -l
- Any non-inlined kernel function (there’s like a bazillion of them)
- No support for USDT or uprobes yet

Tools bundled with BCC



eBPF overhead numbers for kprobes

You should always measure yourself (system CPU is the metric).

Here's what we've measured for getpid() syscall:

Case	ns/op	overhead ns/op	ops/s	overhead percent
no probe	316	0	3,164,556	0%
simple	424	108	2,358,490	34%
complex	647	331	1,545,595	105%

Where should you run ebpf_exporter

Anywhere where overhead is worth it.

- Simple programs can run anywhere
- Complex programs (run queue latency) can be gated to:
  - Canary machines where you test upgrades
  - Timeline around updates

At Cloudflare we do exactly this, except we use canary datacenters.

Thank you

Run it: https://github.com/cloudflare/ebpf_exporter (there are docs!)

Reading materials on eBPF:

- https://iovisor.github.io/bcc/
- https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md
- http://www.brendangregg.com/ebpf.html
- http://docs.cilium.io/en/latest/bpf/

Ivan on twitter: @ibobrik

Questions?

CLOUDFLARE