Container Performance Analysis

Brendan Gregg

Sr. Performance Architect,
Netflix

Take Aways

Identify bottlenecks:

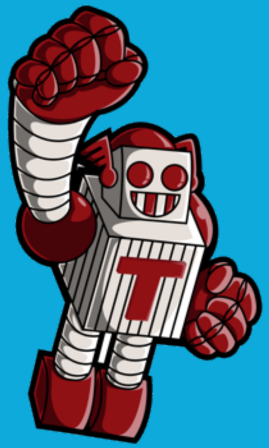
1. In the host vs container, using system metrics
2. In application code on containers, using CPU flame graphs
3. Deeper in the kernel, using tracing tools

Focus of this talk is how containers work in **Linux** (will demo on 4.9)

I will include some **Docker** specifics, and start with a **Netflix** summary (**Titus**)

1. Titus

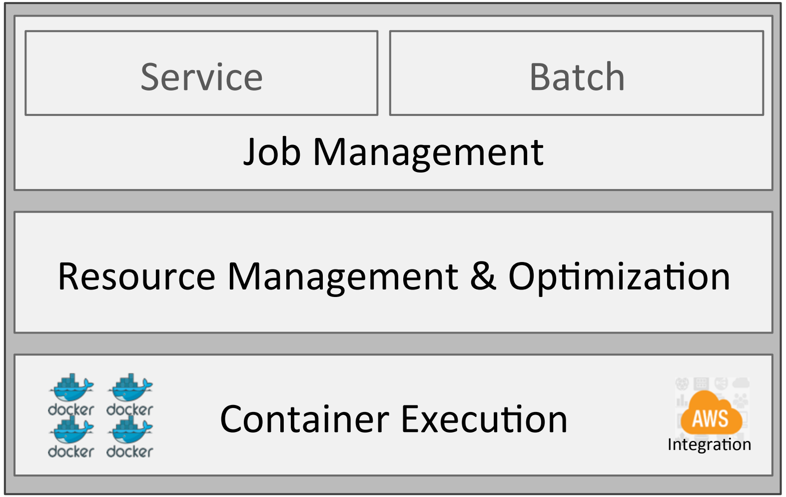
Containers at NETFLIX



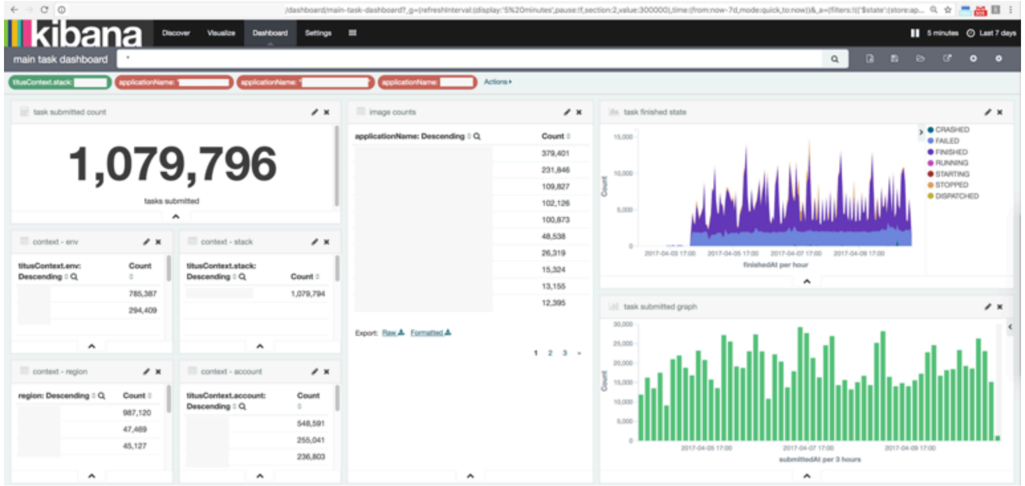
Summary slides from the Titus team

Titus

- Cloud runtime platform for container jobs
- Scheduling
  - Service & batch job management
  - Advanced resource management across elastic shared resource pool
- Container Execution
  - Docker and AWS EC2 Integration
    - Adds VPC, security groups, EC2 metadata, IAM roles, S3 logs, …
  - Integration with Netflix infrastructure
- In depth: http://techblog.netflix.com/2017/04/the-evolution-of-container-usage-at.html



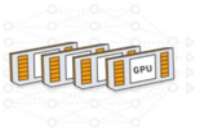
Current Titus Scale



- Deployed across multiple AWS accounts & three regions
- Over 2,500 instances (Mostly M4.4xls & R3.8xls)
- Over a week period launched over 1,000,000 containers

Titus Use Cases

- Service
  - Stream Processing (Flink)
  - UI Services (Node.JS single core)
  - Internal dashboards
- Batch
  - Algorithm training, personalization & recommendations
  - Adhoc reporting
  - Continuous integration builds
- Queued worker model
  - Media encoding



Container Performance @Netflix

- Ability to **scale** and **balance** workloads with EC2 and Titus
  - Can already solve many perf issues

- Performance needs:
  - Application analysis: using CPU flame graphs with containers
  - Host tuning: file system, networking, sysctl's, …
  - Container analysis and tuning: cgroups, GPUs, …
  - Capacity planning: reduce over provisioning

2. Container Background

And Strategy

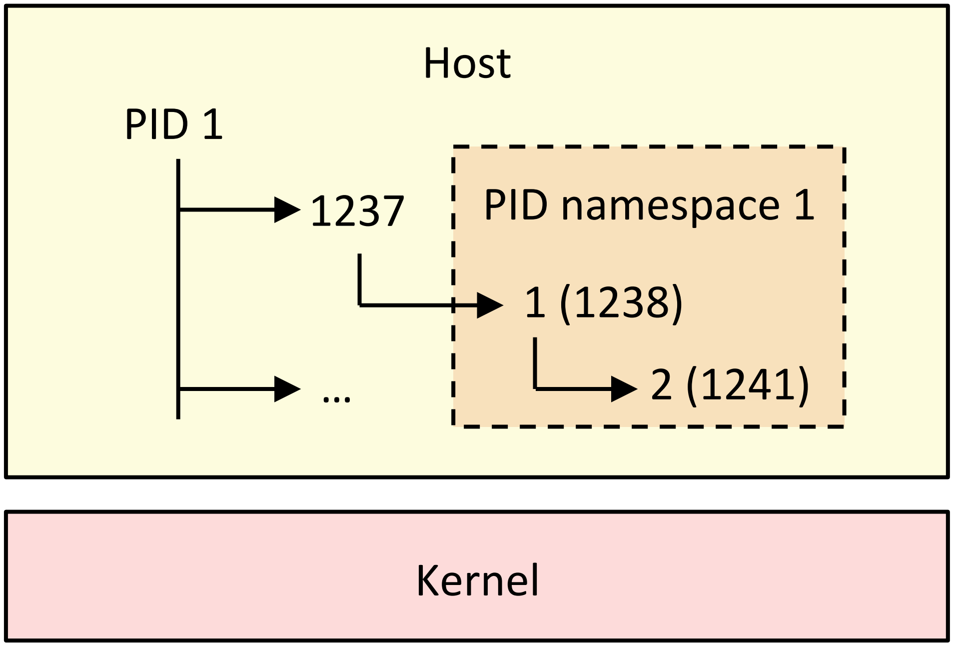
Namespaces

Restricting visibility

Namespaces:

- cgroup
- ipc
- mnt
- net
- pid
- user
- uts

PID namespaces

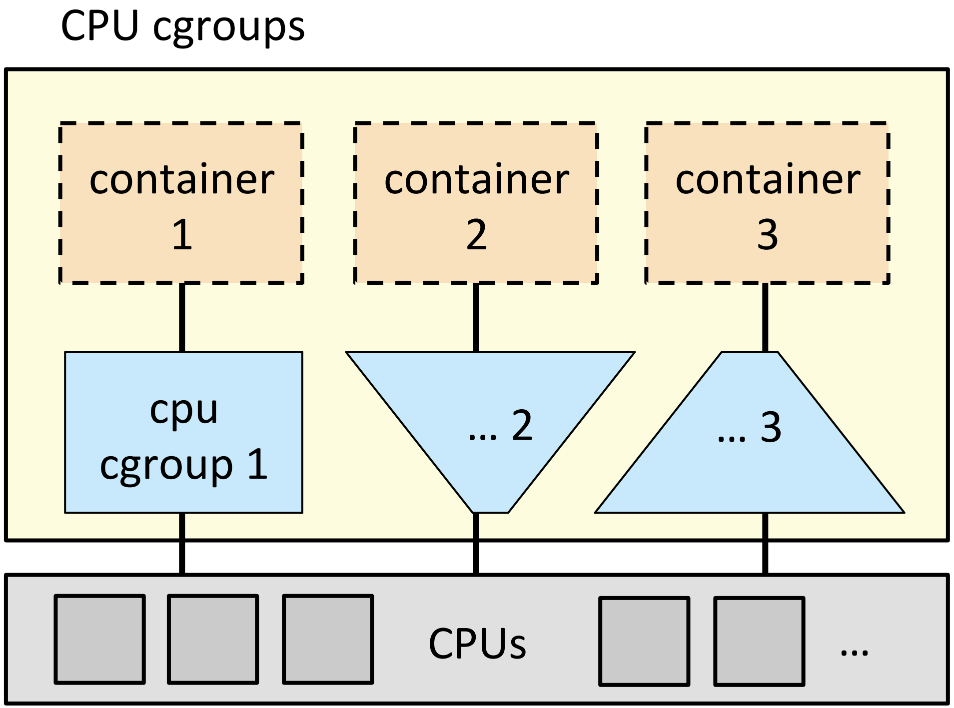


Control Groups

Restricting usage

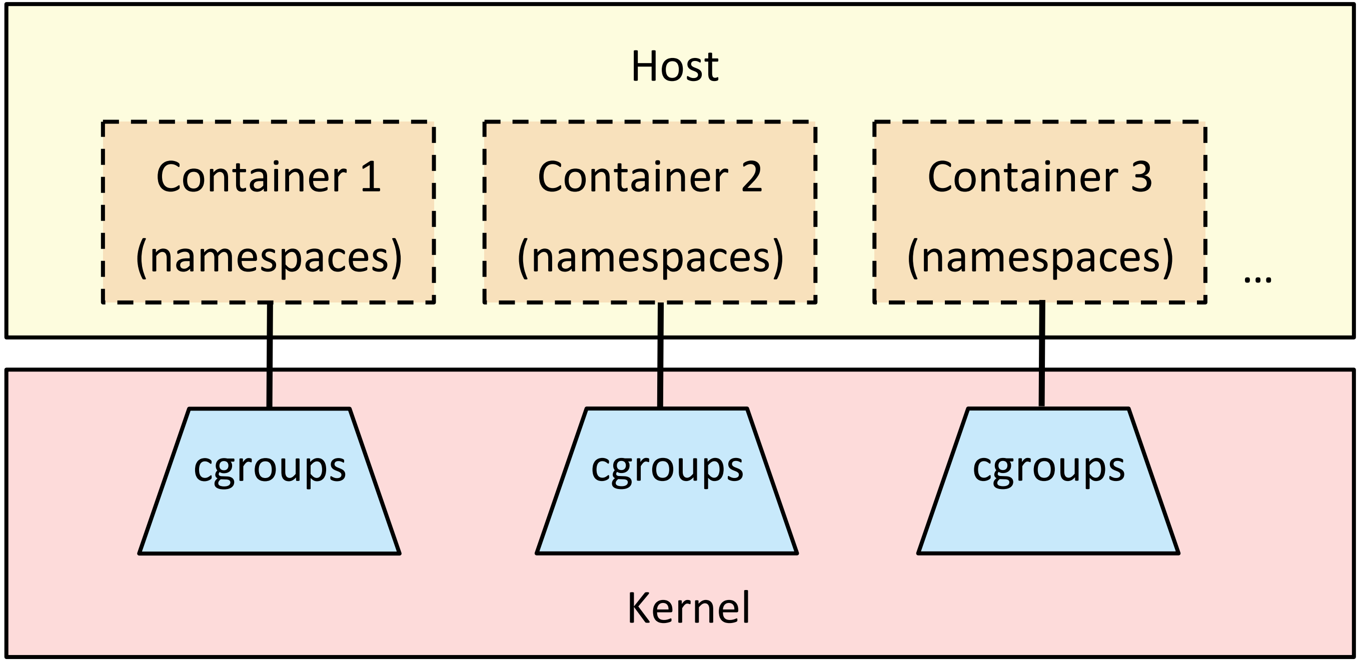
cgroups:

- blkio
- **cpu,cpuacct**
- cpuset
- devices
- hugetlb
- **memory**
- net_cls,net_prio
- pids
- …



Linux Containers

Container = combination of namespaces & cgroups



cgroup v1

cpu,cpuacct:

- **cap CPU usage** (hard limit). e.g. 1.5 CPUs.
- **CPU shares**. e.g. 100 shares.
- usage statistics (cpuacct)

Docker:

--cpus (1.13)

--cpu-shares

memory:

- **limit** and **kmem limit** (maximum bytes)
- **OOM control**: enable/disable
- usage statistics

--memory --kernel-memory

--oom-kill-disable

blkio (block I/O):

- **weights** (like shares)
- **IOPS/tput** caps per storage device
- statistics

CPU Shares

$$\text{Container's CPU limit} = 100\% \times \frac{\text{container's shares}}{\text{total busy shares}}$$

This lets a container use other tenant's idle CPU (aka "bursting"), when available.

$$\text{Container's minimum CPU limit} = 100\% \times \frac{\text{container's shares}}{\text{total allocated shares}}$$

Can make analysis tricky. Why did perf regress? Less bursting available?

cgroup v2

- Major rewrite has been happening: cgroups v2
  - Supports nested groups, better organization and consistency
  - Some already merged, some not yet (e.g. CPU)
- See docs/talks by maintainer Tejun Heo (Facebook)
- References:
  - https://www.kernel.org/doc/Documentation/cgroup-v2.txt
  - https://lwn.net/Articles/679786/

Container OS Configuration

File systems

- Containers may be setup with aufs/overlay on top of another FS
- See "in practice" pages and their performance sections from https://docs.docker.com/engine/userguide/storagedriver/

Networking

- With Docker, can be bridge, host, or overlay networks
- Overlay networks have come with significant performance cost

Analysis Strategy

Performance analysis with containers:

- One kernel
- Two perspectives
- Namespaces
- cgroups

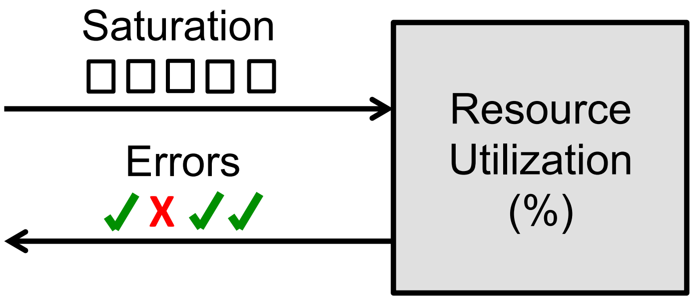
Methodologies:

- USE Method
- Workload characterization
- Checklists
- Event tracing

USE Method

For every resource, check:

1. Utilization
2. Saturation
3. Errors



For example, CPUs:

- Utilization: time busy
- Saturation: run queue length or latency
- Errors: ECC errors, etc.

Can be applied to hardware resources and software resources (cgroups)

3. Host Tools

And Container Awareness

… if you have host access

Host Analysis Challenges

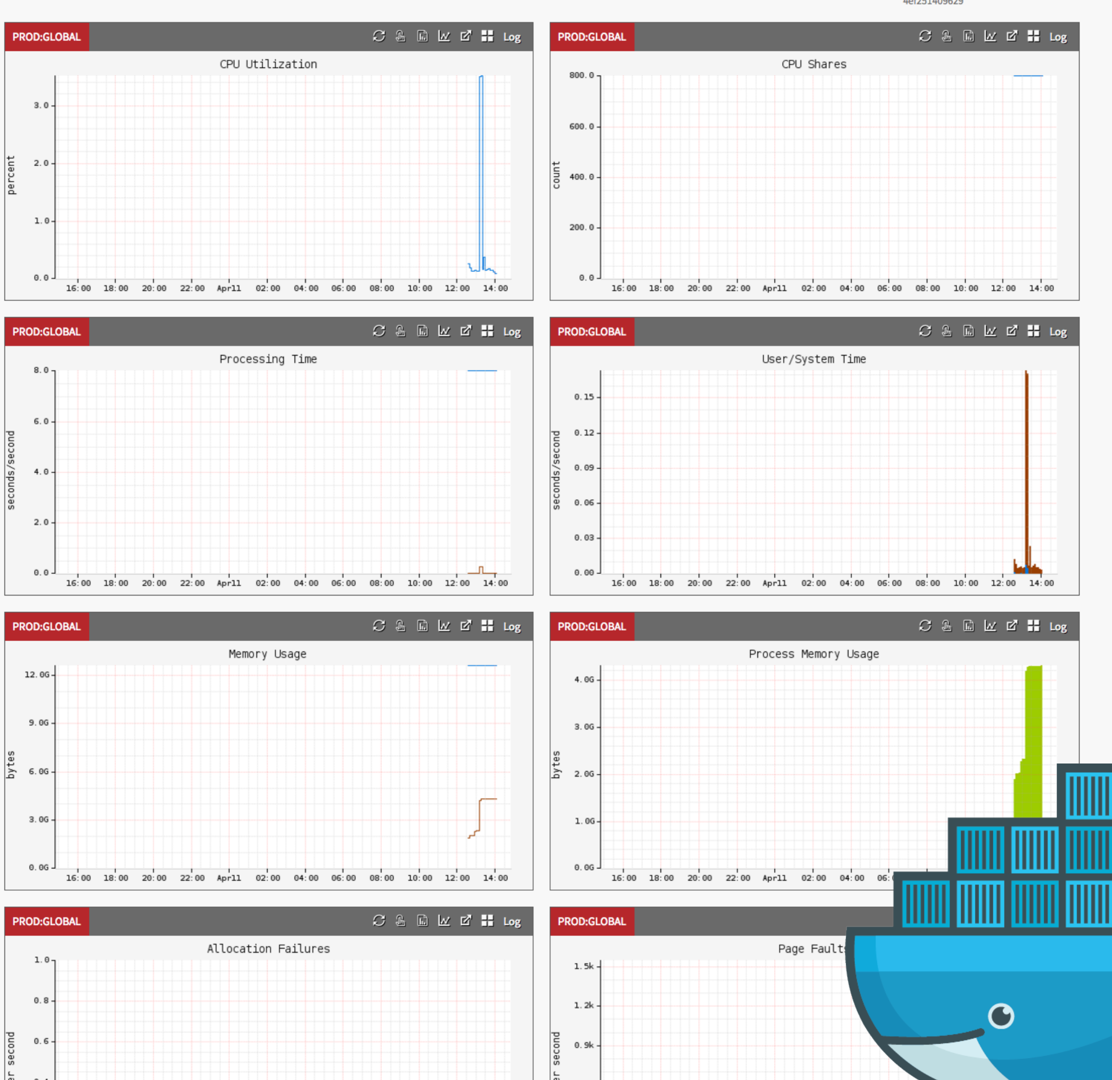
- PIDs in host don't match those seen in containers
- Symbol files aren't where tools expect them
- The kernel currently doesn't have a container ID

CLI Tool Disclaimer

I'll demo CLI tools

It's the lowest common denominator

You may usually use GUIs (like we do). They source the same metrics.



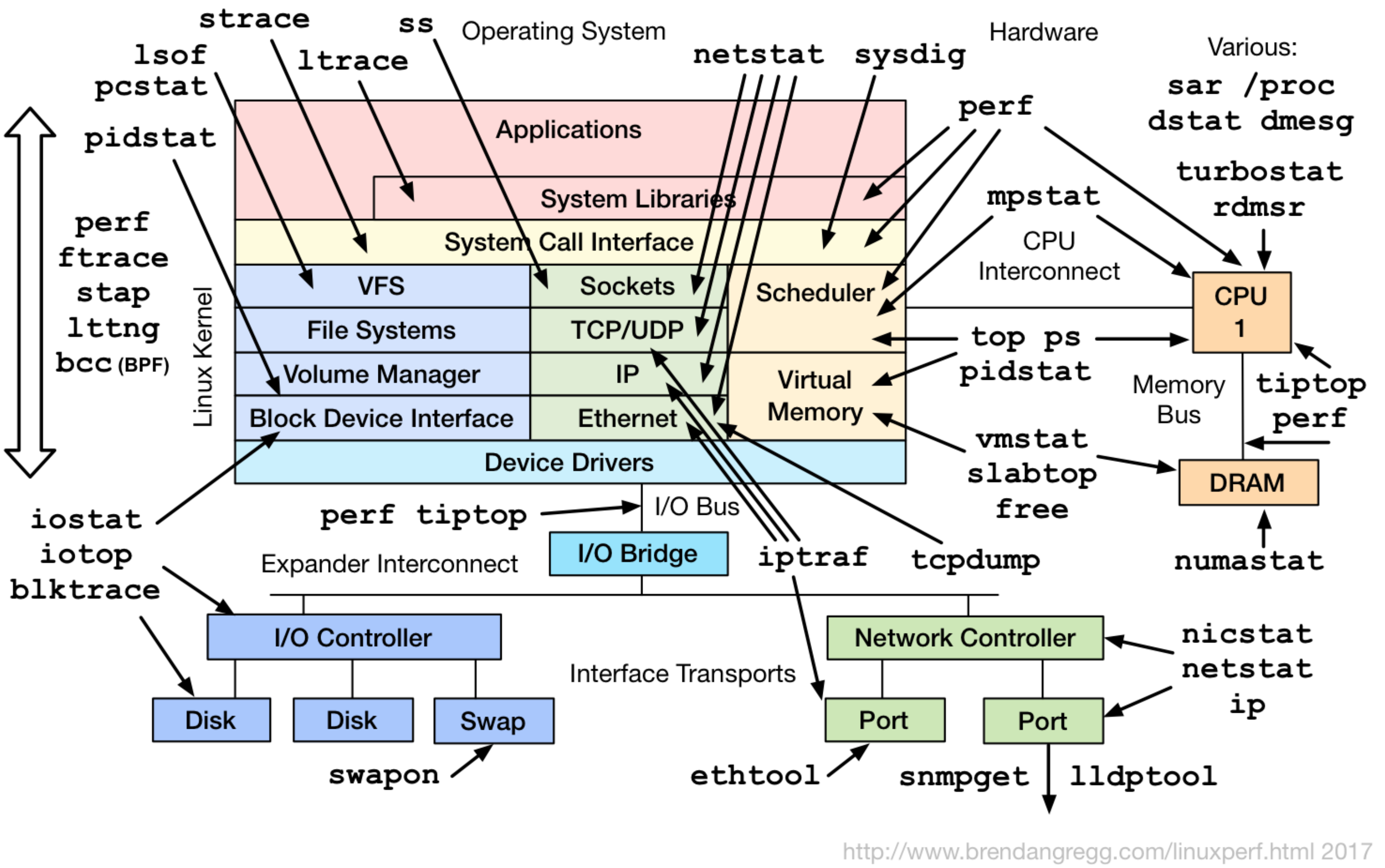
3.1. Host Physical Resources

A refresher of basics... Not container specific.

This will, however, solve many issues!

Containers are often not the problem.

Linux Perf Tools



Where can we begin?

Host Perf Analysis in 60s

1. `uptime` ----------------------------► load averages
2. `dmesg | tail` ------------------------► kernel errors
3. `vmstat 1` --------------------------► overall stats by time
4. `mpstat -P ALL 1` -------------------► CPU balance
5. `pidstat 1` -------------------------► process usage
6. `iostat -xz 1` ----------------------► disk I/O
7. `free -m` ---------------------------► memory usage
8. `sar -n DEV 1` ----------------------► network I/O
9. `sar -n TCP,ETCP 1` -----------------► TCP stats
10. `top` ---------------------------------► check overview

http://techblog.netflix.com/2015/11/linux-performance-analysis-in-60s.html

USE Method: Host Resources

Resource	Utilization	Saturation	Errors
CPU	`mpstat -P ALL 1`, sum non-idle fields	`vmstat 1`, "r"	`perf`
Memory Capacity	`free –m`, "used"/"total"	`vmstat 1`, "si"+"so"; `demsg | grep killed`	`dmesg`
Storage I/O	`iostat –xz 1`, "%util"	`iostat –xnz 1`, "avgqu-sz" > 1	/sys/…/ioerr_cnt; `smartctl`
Network	`nicstat`, "%Util"	`ifconfig`, "overrunns"; `netstat –s "retrans…"`	`ifconfig`, "errors"

These should be in your monitoring GUI. Can do other resources too (busses, ...)

Event Tracing: e.g. iosnoop

Disk I/O events with latency (from perf-tools; also in bcc/BPF as biosnoop)

```
# ./iosnoop
Tracing block I/O... Ctrl-C to end.
COMM             PID    TYPE DEV      BLOCK        BYTES     LATms
supervise        1809   W    202,1    17039968     4096       1.32
supervise        1809   W    202,1    17039976     4096       1.30
tar              14794  RM   202,1    8457608      4096       7.53
tar              14794  RM   202,1    8470336      4096      14.90
tar              14794  RM   202,1    8470368      4096       0.27
tar              14794  RM   202,1    8470784      4096       7.74
tar              14794  RM   202,1    8470360      4096       0.25
tar              14794  RM   202,1    8469968      4096       0.24
tar              14794  RM   202,1    8470240      4096       0.24
tar              14794  RM   202,1    8470392      4096       0.23
```

Event Tracing: e.g. zfsslower

```
# /usr/share/bcc/tools/zfsslower 1
Tracing ZFS operations slower than 1 ms
TIME     COMM           PID    T BYTES   OFF_KB   LAT(ms) FILENAME
23:44:40 java           31386  O 0       0           8.02 solrFeatures.txt
23:44:53 java           31386  W 8190    1812222    36.24 solrFeatures.txt
23:44:59 java           31386  W 8192    1826302    20.28 solrFeatures.txt
23:44:59 java           31386  W 8191    1826846    28.15 solrFeatures.txt
23:45:00 java           31386  W 8192    1831015    32.17 solrFeatures.txt
23:45:15 java           31386  O 0       0          27.44 solrFeatures.txt
23:45:56 dockerd        3599   S 0       0           1.03 .tmp-a66ce9aad…
23:46:16 java           31386  W 31      0          36.28 solrFeatures.txt
```

- This is from our production Titus system (Docker).
- File system latency is a better pain indicator than disk latency.
- zfsslower (and btrfs*, etc) are in bcc/BPF. Can exonerate FS/disks.

Latency Histograms: e.g. btrfsdist

```
# ./btrfsdist
Tracing btrfs operation latency... Hit Ctrl-C to end.
^C
operation = 'read'
     usecs               : count     distribution
         0 -> 1          : 192529   |****************************************|
         2 -> 3          : 72337    |***************                         |
         4 -> 7          : 5620     |*                                       |
         8 -> 15         : 1026     |                                        |
        16 -> 31         : 369      |                                        |
        32 -> 63         : 239      |                                        |
        64 -> 127        : 53       |                                        |
       128 -> 255        : 975      |                                        |
       256 -> 511        : 524      |                                        |
       512 -> 1023       : 128      |                                        |
      1024 -> 2047       : 16       |                                        |
      2048 -> 4095       : 7        |                                        |
      4096 -> 8191       : 2        |                                        |
```

probably
cache reads

probably
cache misses
(flash reads)

Latency Histograms: e.g. btrfsdist

```
[…]
operation = 'write'
     usecs               : count     distribution
         0 -> 1          : 1        |                                        |
         2 -> 3          : 276      |                                        |
         4 -> 7          : 32125    |***********                             |
         8 -> 15         : 111253   |****************************************|
        16 -> 31         : 59154    |*********************                   |
        32 -> 63         : 5463     |*                                       |
        64 -> 127        : 612      |                                        |
       128 -> 255        : 25       |                                        |
       256 -> 511        : 2        |                                        |
       512 -> 1023       : 1        |                                        |
```

- From a test Titus system (Docker).
- Histograms show modes, outliers. Also in bcc/BPF (with other FSes).
- Latency heat maps: http://queue.acm.org/detail.cfm?id=1809426

3.2. Host Containers & cgroups

Inspecting containers from the host

Namespaces

Worth checking namespace config before analysis:

```
# ./dockerpsns.sh
CONTAINER    NAME                    PID PATH              CGROUP     IPC        MNT        NET        PID        USER       UTS
host         titusagent-mainvpc-m      1 systemd           4026531835 4026531839 4026531840 4026532533 4026531836 4026531837 4026531838
b27909cd6dd1 Titus-1435830-worker 37280 svscanboot         4026531835 4026533387 4026533385 4026532931 4026533388 4026531837 4026533386
dcf3a506de45 Titus-1392192-worker 27992 /apps/spaas/spaa   4026531835 4026533354 4026533352 4026532991 4026533355 4026531837 4026533353
370a3f041f36 Titus-1243558-worker 98602 /apps/spaas/spaa   4026531835 4026533290 4026533288 4026533223 4026533291 4026531837 4026533289
af7549c76d9a Titus-1243553-worker 97972 /apps/spaas/spaa   4026531835 4026533216 4026533214 4026533149 4026533217 4026531837 4026533215
dc27769a9b9c Titus-1243546-worker 97356 /apps/spaas/spaa   4026531835 4026533142 4026533140 4026533075 4026533143 4026531837 4026533141
e18bd6189dcd Titus-1243517-worker 96733 /apps/spaas/spaa   4026531835 4026533068 4026533066 4026533001 4026533069 4026531837 4026533067
ab45227dcea9 Titus-1243516-worker 96173 /apps/spaas/spaa   4026531835 4026532920 4026532918 4026532830 4026532921 4026531837 4026532919
```

- A POC "docker ps --namespaces" tool. NS shared with root in red.
- https://github.com/docker/docker/issues/32501

systemd-cgtop

A "top" for cgroups:

```
# systemd-cgtop
Control Group                                   Tasks   %CPU   Memory  Input/s Output/s
/                                                   -  798.2    45.9G        -        -
/docker                                          1082  790.1    42.1G        -        -
/docker/dcf3a...9d28fc4a1c72bbaff4a24834          200  610.5    24.0G        -        -
/docker/370a3...e64ca01198f1e843ade7ce21          170  174.0     3.0G        -        -
/system.slice                                     748    5.3     4.1G        -        -
/system.slice/daemontools.service                 422    4.0     2.8G        -        -
/docker/dc277...42ab0603bbda2ac8af67996b          160    2.5     2.3G        -        -
/user.slice                                         5    2.0    34.5M        -        -
/user.slice/user-0.slice                            5    2.0    15.7M        -        -
/user.slice/u....slice/session-c26.scope            3    2.0    13.3M        -        -
/docker/ab452...c946f8447f2a4184f3ccff2a          174    1.0     6.3G        -        -
/docker/e18bd...26ffdd7368b870aa3d1deb7a          156    0.8     2.9G        -        -
[...]
```

docker stats

A "top" for containers. Resource utilization. Workload characterization.

```
# docker stats
CONTAINER     CPU %     MEM USAGE / LIMIT       MEM %     NET I/O     BLOCK I/O         PIDS
353426a09db1  526.81%   4.061 GiB / 8.5 GiB     47.78%    0 B / 0 B   2.818 MB / 0 B    247
6bf166a66e08  303.82%   3.448 GiB / 8.5 GiB     40.57%    0 B / 0 B   2.032 MB / 0 B    267
58dcf8aed0a7  41.01%    1.322 GiB / 2.5 GiB     52.89%    0 B / 0 B   0 B / 0 B         229
61061566ffe5  85.92%    220.9 MiB / 3.023 GiB   7.14%     0 B / 0 B   43.4 MB / 0 B     61
bdc721460293  2.69%     1.204 GiB / 3.906 GiB   30.82%    0 B / 0 B   4.35 MB / 0 B     66
6c80ed61ae63  477.45%   557.7 MiB / 8 GiB       6.81%     0 B / 0 B   9.257 MB / 0 B    19
337292fb5b64  89.05%    766.2 MiB / 8 GiB       9.35%     0 B / 0 B   5.493 MB / 0 B    19
b652ede9a605  173.50%   689.2 MiB / 8 GiB       8.41%     0 B / 0 B   6.48 MB / 0 B     19
d7cd2599291f  504.28%   673.2 MiB / 8 GiB       8.22%     0 B / 0 B   12.58 MB / 0 B    19
05bf9f3e0d13  314.46%   711.6 MiB / 8 GiB       8.69%     0 B / 0 B   7.942 MB / 0 B    19
09082f005755  142.04%   693.9 MiB / 8 GiB       8.47%     0 B / 0 B   8.081 MB / 0 B    19
bd45a3e1ce16  190.26%   538.3 MiB / 8 GiB       6.57%     0 B / 0 B   10.6 MB / 0 B     19
[...]
```

Loris Degioanni demoed a similar sysdigcloud view yesterday (needs the sysdig kernel agent)

top

In the host, top shows all processes. Currently doesn't show a container ID.

```
# top - 22:46:53 up 36 days, 59 min,  1 user,  load average: 5.77, 5.61, 5.63
Tasks: 1067 total,   1 running, 1046 sleeping,   0 stopped,  20 zombie
%Cpu(s): 34.8 us,  1.8 sy,  0.0 ni, 61.3 id,  0.0 wa,  0.0 hi,  1.9 si,  0.1 st
KiB Mem : 65958552 total, 12418448 free, 49247988 used,  4292116 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 13101316 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 28321 root      20   0 33.126g 0.023t  37564 S 621.1 38.2  35184:09 java
 97712 root      20   0 11.445g 2.333g  37084 S   3.1  3.7 404:27.90 java
 98306 root      20   0 12.149g 3.060g  36996 S   2.0  4.9 194:21.10 java
 96511 root      20   0 15.567g 6.313g  37112 S   1.7 10.0 168:07.44 java
  5283 root      20   0 1643676 100092  94184 S   1.0  0.2 401:36.16 mesos-slave
  2079 root      20   0    9512    132     12 S   0.7  0.0 220:07.75 rngd
  5272 titusag+  20   0 10.473g 1.611g  23488 S   0.7  2.6   1934:44 java
[…]
```

… remember, there is no container ID in the kernel yet.

htop

htop can add a CGROUP field, but, can truncate important info:

```
 CGROUP       PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
:pids:/docker/  28321 root       20   0 33.1G 24.0G 37564 S 524. 38.2    672h /apps/java
:pids:/docker/   9982 root       20   0 33.1G 24.0G 37564 S 44.4 38.2 17h00:41 /apps/java
:pids:/docker/   9985 root       20   0 33.1G 24.0G 37564 R 41.9 38.2 16h44:51 /apps/java
:pids:/docker/   9979 root       20   0 33.1G 24.0G 37564 S 41.2 38.2 17h01:35 /apps/java
:pids:/docker/   9980 root       20   0 33.1G 24.0G 37564 S 39.3 38.2 16h59:17 /apps/java
:pids:/docker/   9981 root       20   0 33.1G 24.0G 37564 S 39.3 38.2 17h01:32 /apps/java
:pids:/docker/   9984 root       20   0 33.1G 24.0G 37564 S 37.3 38.2 16h49:03 /apps/java
:pids:/docker/   9983 root       20   0 33.1G 24.0G 37564 R 35.4 38.2 16h54:31 /apps/java
:pids:/docker/   9986 root       20   0 33.1G 24.0G 37564 S 35.4 38.2 17h05:30 /apps/java
:name=systemd:/user.slice/user-0.slice/session-c31.scope?  74066 root       20   0 27620
:pids:/docker/   9998 root       20   0 33.1G 24.0G 37564 R 28.3 38.2 11h38:03 /apps/java
:pids:/docker/  10001 root       20   0 33.1G 24.0G 37564 S 27.7 38.2 11h38:59 /apps/java
:name=systemd:/system.slice/daemontools.service?   5272 titusagen  20   0 10.5G 1650M 23
:pids:/docker/  10002 root       20   0 33.1G 24.0G 37564 S 25.1 38.2 11h40:37 /apps/java
```

Can fix, but that would be Docker + cgroup-v1 specific. Still need a kernel CID.

Host PID -> Container ID

… who does that (CPU busy) PID 28321 belong to?

```
# grep 28321 /sys/fs/cgroup/cpu,cpuacct/docker/*/tasks | cut -d/ -f7
dcf3a506de453107715362f6c9ba9056fcfc6e769d28fc4a1c72bbaff4a24834
```

- Only works for Docker, and that cgroup v1 layout. Some Linux commands:

```
# ls -l /proc/27992/ns/*
lrwxrwxrwx 1 root root 0 Apr 13 20:49 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Apr 13 20:49 ipc -> ipc:[4026533354]
lrwxrwxrwx 1 root root 0 Apr 13 20:49 mnt -> mnt:[4026533352]
[…]
# cat /proc/27992/cgroup
11:freezer:/docker/dcf3a506de453107715362f6c9ba9056fcfc6e769d28fc4a1c72bbaff4a24834
10:blkio:/docker/dcf3a506de453107715362f6c9ba9056fcfc6e769d28fc4a1c72bbaff4a24834
9:perf_event:/docker/dcf3a506de453107715362f6c9ba9056fcfc6e769d28fc4a1c72bbaff4a24834
[…]
```

nsenter Wrapping

… what hostname is PID 28321 running on?

```
# nsenter -t 28321 -u hostname
titus-1392192-worker-14-16
```

- Can namespace enter:
  - -m: mount -u: uts -i: ipc -n: net -p: pid -U: user
- Bypasses cgroup limits, and seccomp profile (allowing syscalls)
  - For Docker, you can enter the container more completely with: docker exec -it CID command
- Handy nsenter one-liners:
  - `nsenter -t PID -u hostname` container hostname
  - `nsenter -t PID -n netstat -i` container netstat
  - `nsenter -t PID –m -p df -h` container file system usage
  - `nsenter -t PID -p top` container top

nsenter: Host -> Container top

… Given PID 28321, running top for its container by entering its namespaces:

```
# nsenter -t 28321 -m -p top

top - 18:16:13 up 36 days, 20:28,  0 users,  load average: 5.66, 5.29, 5.28
Tasks:   6 total,   1 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu(s): 30.5 us,  1.7 sy,  0.0 ni, 65.9 id,  0.0 wa,  0.0 hi,  1.8 si,  0.1 st
KiB Mem:  65958552 total, 54664124 used, 11294428 free,   164232 buffers
KiB Swap:        0 total,        0 used,        0 free.  1592372 cached Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
   301 root      20   0 33.127g 0.023t  37564 S 537.3 38.2  40269:41 java
     1 root      20   0   21404   2236   1812 S   0.0  0.0   4:15.11 bash
 87888 root      20   0   21464   1720   1348 R   0.0  0.0   0:00.00 top
```

Note that it is PID 301 in the container. Can also see this using:

```
# grep NSpid /proc/28321/status
NSpid:    28321     301
```

perf: CPU Profiling

Can run system-wide (-a), match a pid (-p), or cgroup (-G, if it works)

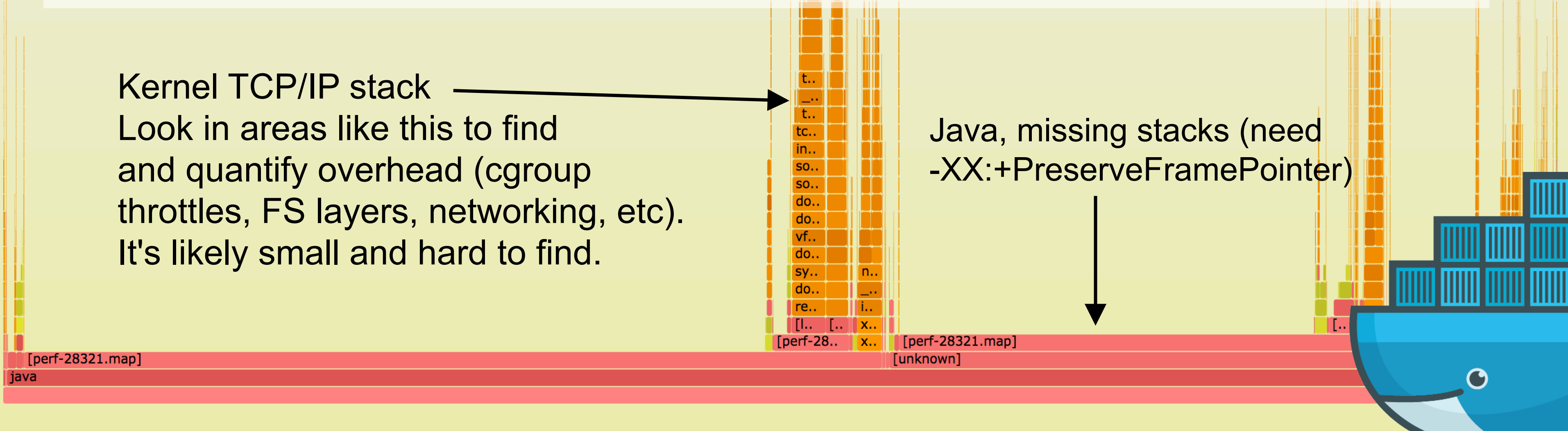
```
# perf record -F 49 -a -g -- sleep 30
# perf script
Failed to open /lib/x86_64-linux-gnu/libc-2.19.so, continuing without symbols
Failed to open /tmp/perf-28321.map, continuing without symbols
```

- Current symbol translation gotchas (up to 4.10-ish):
  - perf can't find /tmp/perf-PID.map files in the host, and the PID is different
  - perf can't find container binaries under host paths (what /usr/bin/java?)
- Can copy files to the host, map PIDs, then run perf script/report:
  - http://blog.alicegoldfuss.com/making-flamegraphs-with-containerized-java/
  - http://batey.info/docker-jvm-flamegraphs.html
- Can nsenter (-m -u -i -n -p) a "power" shell, and then run "perf -p PID"
- perf should be fixed to be namespace aware (like bcc was, PR#1051)

CPU Flame Graphs

```
git clone --depth 1 https://github.com/brendangregg/FlameGraph
cd FlameGraph
perf record –F 49 -a –g -- sleep 30
perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf.svg
```

- See previous slide for getting perf symbols to work
- From the host, can study all containers, as well as container overheads



/sys/fs/cgroups (raw)

The best source for per-cgroup metrics. e.g. CPU:

```
# cd /sys/fs/cgroup/cpu,cpuacct/docker/02a7cf65f82e3f3e75283944caa4462e82f8f6ff5a7c9a...
# ls
cgroup.clone_children  cpuacct.usage_all          cpuacct.usage_sys   cpu.shares
cgroup.procs           cpuacct.usage_percpu       cpuacct.usage_user  cpu.stat
cpuacct.stat           cpuacct.usage_percpu_sys   cpu.cfs_period_us   notify_on_release
cpuacct.usage          cpuacct.usage_percpu_user  cpu.cfs_quota_us    tasks
# cat cpuacct.usage
1615816262506
# cat cpu.stat
nr_periods 507
nr_throttled 74
throttled_time 3816445175
```

total time throttled (nanoseconds). saturation metric.
average throttle time = throttled_time / nr_throttled

- https://www.kernel.org/doc/Documentation/cgroup-v1/, ../scheduler/sched-bwc.txt
- https://blog.docker.com/2013/10/gathering-lxc-docker-containers-metrics/

Note: `grep cgroup /proc/mounts` to check where these are mounted

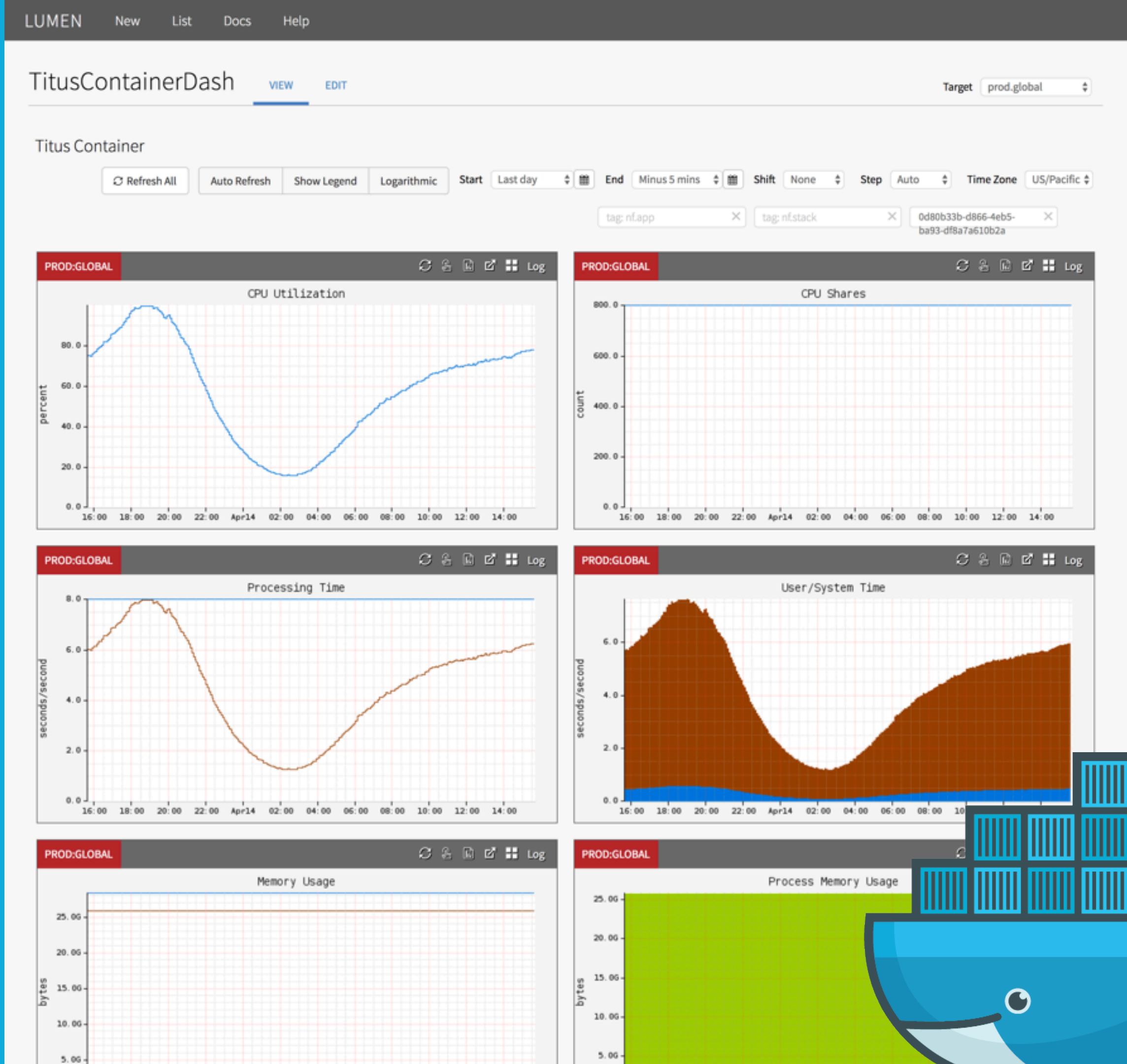
These metrics should be included in performance monitoring GUIs

Netflix Atlas

Cloud-wide monitoring of containers (and instances)

Fetches cgroup metrics via Intel snap

https://github.com/netflix/Atlas



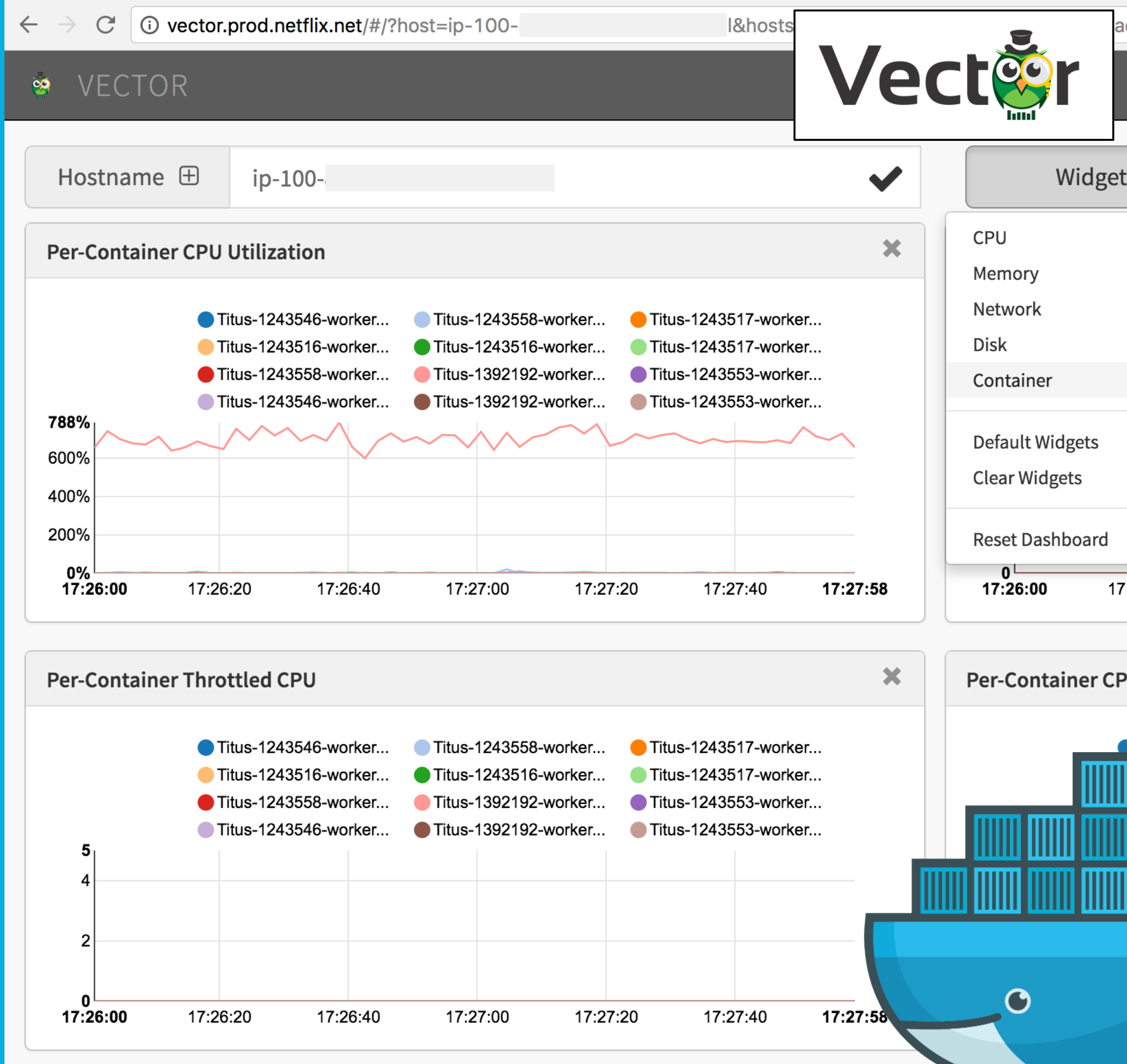
Netflix Vector

Our per-instance analyzer

Has per-container metrics

https://github.com/Netflix/vector

- Per-Container CPU Utilization
- Per-Container Memory Usage (Mb)
- Total Container Memory Usage (Mb)
- Per-Container Memory Headroom (Mb)
- Container Disk IOPS
- Container Disk Throughput (Bytes)
- Container Disk IOPS (Throttled)
- Container Disk Throughput (Throttled) (Bytes)
- Per-Container CPU Scheduler
- Per-Container CPU Headroom
- Per-Container Throttled CPU
- Per-Container Memory Utilization



Intel snap

A metric collector used by monitoring GUIs

https://github.com/intelsdi-x/snap

Has a Docker plugin to read cgroup stats

There's also a collectd plugin:

https://github.com/bobrik/collectd-docker

cpu_stats/cpu_usage/kernel_mode	uint64	CPU time consumed by tasks in system (kernel) mode
cpu_stats/cpu_usage/user_mode	uint64	CPU time consumed by tasks in user mode
cpu_stats/cpu_usage/per_cpu/<N>/value	uint64	CPU time consumed on each N-th CPU by all tasks
cpu_stats/throttling_data/nr_periods	uint64	The number of period intervals that have elapsed
cpu_stats/throttling_data/nr_throttled	uint64	The number of times tasks in a cgroup have been throttled
cpu_stats/throttling_data/throttled_time	uint64	The total time duration for which tasks in a cgroup have been throttled
cpu_stats/cpu_shares	uint64	The relative share of CPU time available to the tasks in a cgroup
cpuset_stats/cpu_exclusive	uint64	Flag (0 or 1) that specifies whether cpusets other than this one and its parents and children can share the CPUs specified for this cpuset
cpuset_stats/memory_exclusive	uint64	Flag (0 or 1) that specifies whether other cpusets can share the memory nodes specified for the cpuset
cpuset_stats/memory_migrate	uint64	Flag (0 or 1) that specifies whether a page in memory should migrate to a new node if the values in cpuset.mems change
cpuset_stats/cpus	string	CPUs numbers that tasks in this cgr permitted to access
cpuset_stats/mems	string	Memory nodes that tasks in th permitted to access
pids_stats/current	uint64	The current number of PI
pids_stats/limit	uint64	The maximum number of PI

3.3. Let's Play a Game

Host or Container?

(or neither?)

Game Scenario 1

Container user claims they have a CPU performance issue

- Container has a CPU cap and CPU shares configured
- There is idle CPU on the host
- Other tenants are CPU busy
- /sys/fs/cgroup/.../cpu.stat -> throttled_time is increasing
- /proc/PID/status nonvoluntary_ctxt_switches is increasing
- Container CPU usage equals its cap (clue: this is not really a clue)

Game Scenario 2

Container user claims they have a CPU performance issue

- Container has a CPU cap and CPU shares configured
- There is no idle CPU on the host
- Other tenants are CPU busy
- /sys/fs/cgroup/.../cpu.stat -> throttled_time is not increasing
- /proc/PID/status nonvoluntary_ctxt_switches is increasing

Game Scenario 3

Container user claims they have a CPU performance issue

- Container has CPU shares configured
- There is no idle CPU on the host
- Other tenants are CPU busy
- /sys/fs/cgroup/.../cpu.stat -> throttled_time is not increasing
- /proc/PID/status nonvoluntary_ctxt_switches is not increasing much

Experiments to confirm conclusion?

Methodology: Reverse Diagnosis

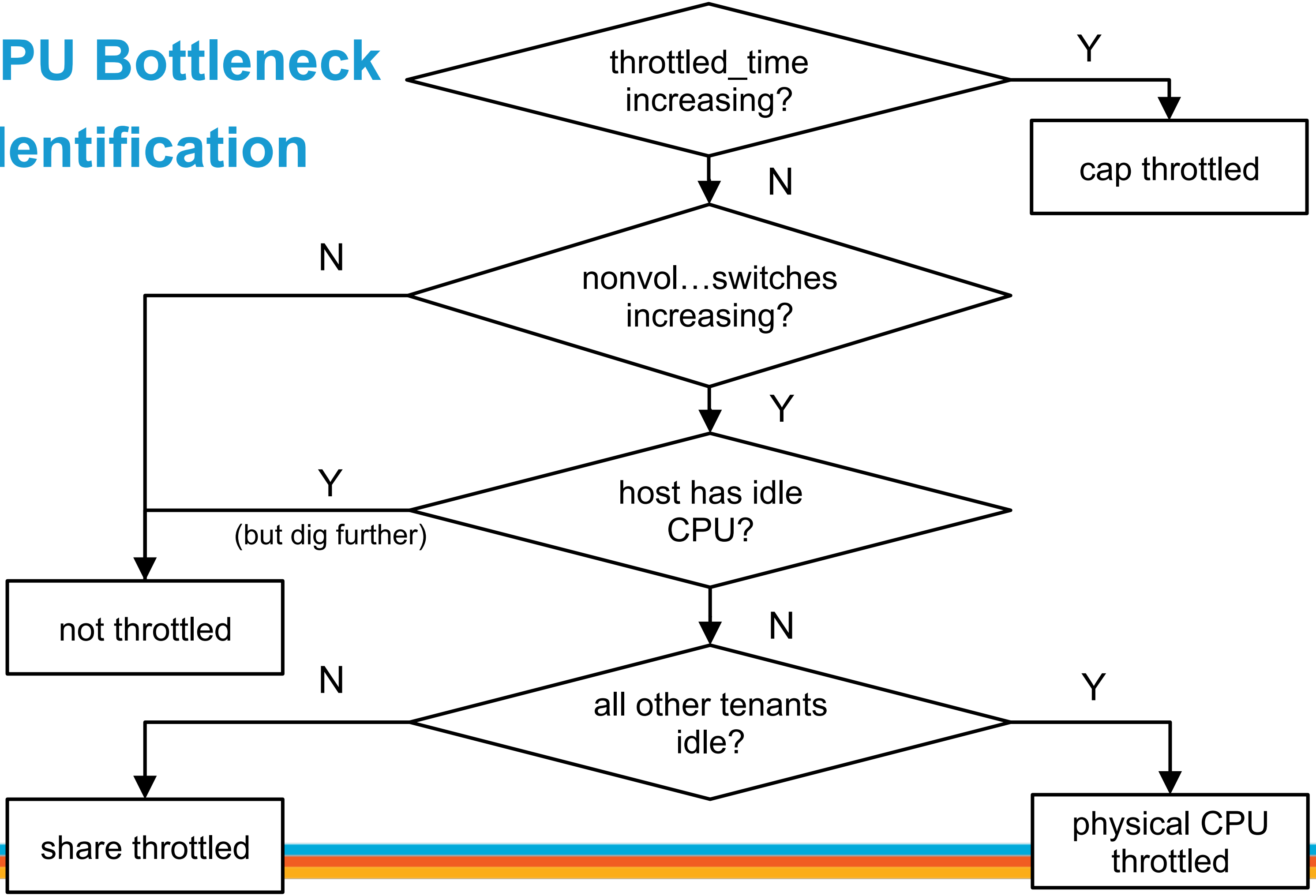
Enumerate possible outcomes, and work backwards to the metrics needed for diagnosis.

e.g. CPU performance outcomes:

A. physical CPU throttled
B. cap throttled
C. shares throttled (assumes physical CPU limited as well)
D. not throttled

Game answers: 1. B, 2. C, 3. D

CPU Bottleneck Identification



docker 17 con

4. Guest Tools

And Container Awareness

… if you only have guest access

Guest Analysis Challenges

- Some resource metrics are for the container, some for the host. Confusing!
- May lack system capabilities or syscalls to run profilers and tracers

CPU

Can see host's CPU devices, but only container (pid namespace) processes:

```
container# uptime
 20:17:19 up 45 days, 21:21,  0 users,  load average: 5.08, 3.69, 2.22
container# mpstat 1
Linux 4.9.0 (02a7cf65f82e) 04/14/17   _x86_64_   (8 CPU)

20:17:26     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
20:17:27     all   51.00    0.00   12.28    0.00    0.00    0.00    0.00    0.00    0.00   36.72
20:17:28     all   50.88    0.00   12.31    0.00    0.00    0.00    0.00    0.00    0.00   36.81
^C
Average:     all   50.94    0.00   12.30    0.00    0.00    0.00    0.00    0.00    0.00   36.76
container# pidstat 1
Linux 4.9.0 (02a7cf65f82e) 04/14/17   _x86_64_   (8 CPU)

20:17:33      UID       PID    %usr %system  %guest    %CPU   CPU  Command

20:17:34      UID       PID    %usr %system  %guest    %CPU   CPU  Command

20:17:35      UID       PID    %usr %system  %guest    %CPU   CPU  Command
[...]
```

load!

busy CPUs

but this container is running nothing (we saw CPU usage from neighbors)

Memory

Can see host's memory:

```
container# free -m
              total        used        free      shared  buff/cache   available
Mem:          15040        1019        8381         153        5639       14155
Swap:             0           0           0

container# perl -e '$a = "A" x 1_000_000_000'
Killed
```

host memory (this container is --memory=1g)

tries to consume ~2 Gbytes

Disks

Can see host's disk devices:

```
container# iostat -xz 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          52.57    0.00   16.94    0.00    0.00   30.49

Device:   rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
xvdap1      0.00     7.00    0.00    2.00     0.00    36.00    36.00     0.00    2.00    0.00    2.00   2.00   0.40
xvdb        0.00     0.00  200.00    0.00  3080.00     0.00    30.80     0.04    0.20    0.20    0.00   0.20   4.00
xvdc        0.00     0.00  185.00    0.00  2840.00     0.00    30.70     0.04    0.24    0.24    0.00   0.24   4.40
md0         0.00     0.00  385.00    0.00  5920.00     0.00    30.75     0.00    0.00    0.00    0.00   0.00   0.00
[...]
container# pidstat -d 1
Linux 4.9.0 (02a7cf65f82e)  04/18/17    _x86_64_    (8 CPU)

22:41:13      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command

22:41:14      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command

22:41:15      UID       PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
[...]
```

host disk I/O

but no container I/O

Network

Can't see host's network interfaces (network namespace):

```
container# sar -n DEV,TCP 1
Linux 4.9.0 (02a7cf65f82e)  04/14/17    _x86_64_    (8 CPU)

21:45:07        IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s   %ifutil
21:45:08           lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
21:45:08         eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

21:45:07     active/s passive/s    iseg/s    oseg/s
21:45:08         0.00      0.00      0.00      0.00

21:45:08        IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s   %ifutil
21:45:09           lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
21:45:09         eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

21:45:08     active/s passive/s    iseg/s    oseg/s
21:45:09         0.00      0.00      0.00      0.00
[...]
```

host has heavy network I/O,
container sees itself (idle)

Metrics Namespace

This confuses apps too: trying to bind on all CPUs, or using 25% of memory

- Including the JDK, which is unaware of container limits, covered yesterday by Fabiane Nardon

We could add a "metrics" namespace so the container only sees itself

- Or enhance existing namespaces to do this

If you add a metrics namespace, please consider adding an option for:

- /proc/host/stats: maps to host's /proc/stats, for CPU stats
- /proc/host/diskstats: maps to host's /proc/diskstats, for disk stats

As those host metrics can be useful, to identify/exonerate neighbor issues

perf: CPU Profiling

Needs capabilities to run from a container:

```
container# ./perf record -F 99 -a -g -- sleep 10
perf_event_open(..., PERF_FLAG_FD_CLOEXEC) failed with unexpected error 1 (Operation not permited)
perf_event_open(..., 0) failed unexpectedly with error 1 (Operation not permited)
Error: You may not have permission to collect system-wide stats.

Consider tweaking /proc/sys/kernel/perf_event_paranoid,
which controls use of the performance events system by
unprivileged users (without CAP_SYS_ADMIN).

The current value is 2:

  -1: Allow use of (almost) all events by all users
>= 0: Disallow raw tracepoint access by users without CAP_IOC_LOCK
>= 1: Disallow CPU event access by users without CAP_SYS_ADMIN
>= 2: Disallow kernel profiling by users without CAP_SYS_ADMIN
```

Helpful message

Although, after setting perf_event_paranoid to -1, it prints the same error...

perf & Container Debugging

Debugging using strace from the host (as `ptrace()` is also blocked):

```
host# strace -fp 26450
[...]
[pid 27426] perf_event_open(0x2bfe498, -1, 0, -1, 0) = -1 EPERM (Operation not permitted)
[pid 27426] perf_event_open(0x2bfe498, -1, 0, -1, 0) = -1 EPERM (Operation not permitted)
[pid 27426] perf_event_open(0x2bfc1a8, -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = -1 EPERM (Operation not permitted)
```

bash PID, from which I then ran perf

Many different ways to debug this.

https://docs.docker.com/engine/security/seccomp/#**significant-syscalls-blocked-by-the-default-profile**:

...

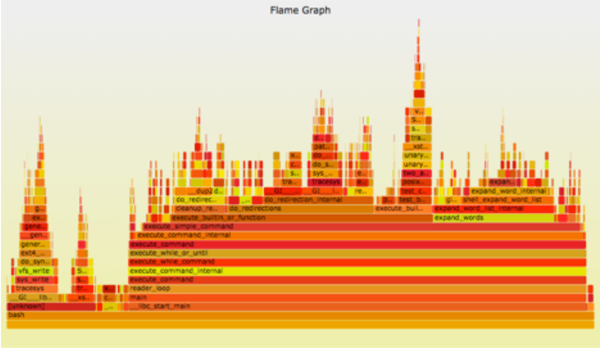
`open_by_handle_at`	Cause of an old container breakout. Also gated by `CAP_DAC_READ_SEARCH` .
`perf_event_open`	Tracing/profiling syscall, which could leak a lot of information on the host.
`personality`	Prevent container from enabling BSD emulation. Not inherently dangerous, but poorly tested, potential for a lot of kernel vulns.

...

perf, cont.

- Can enable perf_event_open() with: `docker run --cap-add sys_admin`
  - Also need (for kernel symbols): `echo 0 > /proc/sys/kernel/kptr_restrict`
- perf then "works", and you can make **flame graphs**. But it sees all CPUs!?
  - perf needs to be "container aware", and only see the container's tasks.

    patch pending: https://lkml.org/lkml/2017/1/12/308
- Currently easier to run perf from the host (or secure "monitoring" container)
  - Via a secure monitoring agent,
    e.g. Netflix Vector -> CPU Flame Graph
  - See earlier slides for steps



5. Tracing

Advanced Analysis

… a few more examples
(iosnoop, zfsslower, and btrfsdist shown earlier)

Built-in Linux Tracers

ftrace
(2008+)

perf_events
(2009+)

eBPF
(2014+)

Some front-ends:

- ftrace: https://github.com/brendangregg/**perf-tools**
- perf_events: used for **CPU flame graphs**
- eBPF (aka BPF): https://github.com/iovisor/**bcc** (Linux 4.4+)

ftrace: Overlay FS Function Calls

Using ftrace via my perf-tools to count function calls in-kernel context:

```
# funccount '*ovl*'
Tracing "*ovl*"... Ctrl-C to end.
^C
FUNC                              COUNT
ovl_cache_free                        3
ovl_xattr_get                         3
[...]
ovl_fill_merge                      339
ovl_path_real                       617
ovl_path_upper                      777
ovl_update_time                     777
ovl_permission                     1408
ovl_d_real                         1434
ovl_override_creds                 1804

Ending tracing...
```

Each can be a target for further study with kprobes

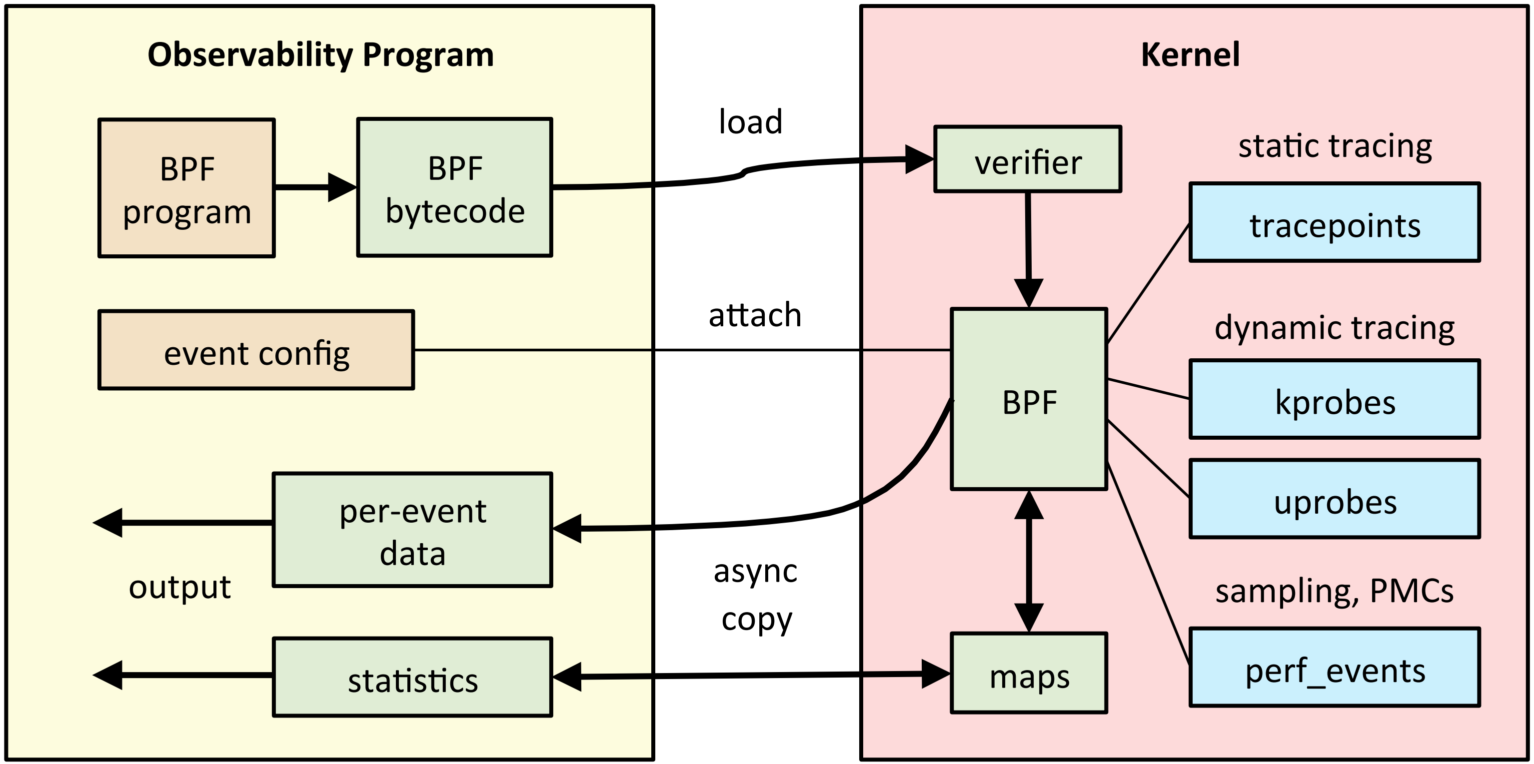
ftrace: Overlay FS Function Tracing

Using kprobe (perf-tools) to trace ovl_fill_merg() args and stack trace

```
# kprobe -s 'p:ovl_fill_merge ctx=%di name=+0(%si):string'
Tracing kprobe ovl_fill_merge. Ctrl-C to end.
            bash-16633 [000] d... 14390771.218973: ovl_fill_merge: (ovl_fill_merge+0x0/0x1f0
[overlay]) ctx=0xffffc90042477db0 name="iostat"
            bash-16633 [000] d... 14390771.218981: <stack trace>
 => ovl_fill_merge
 => ext4_readdir
 => iterate_dir
 => ovl_dir_read_merged
 => ovl_iterate
 => iterate_dir
 => SyS_getdents
 => do_syscall_64
 => return_from_SYSCALL_64
[…]
```

Good for debugging, although dumping all events can cost too much overhead. ftrace has some solutions to this, BPF has more…

Enhanced BPF Tracing Internals



BPF: Scheduler Latency 1

```
host# runqlat -p 20228 10 1
Tracing run queue latency... Hit Ctrl-C to end.

     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 4        |                                        |
         4 -> 7          : 368      |****************************************|
         8 -> 15         : 151      |****************                        |
        16 -> 31         : 22       |**                                      |
        32 -> 63         : 14       |*                                       |
        64 -> 127        : 19       |**                                      |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 2        |                                        |
       512 -> 1023       : 1        |                                        |
```

This is an app in a Docker container on a system with idle CPU

Tracing scheduler events can be costly (high rate), but this BPF program reduces cost by using in-kernel maps to summarize data, and only emits the "count" column to user space.

BPF: Scheduler Latency 2

```
host# runqlat -p 20228 10 1
Tracing run queue latency... Hit Ctrl-C to end.

     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 7        |**                                      |
         8 -> 15         : 14       |*****                                   |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 0        |                                        |
       512 -> 1023       : 0        |                                        |
      1024 -> 2047       : 0        |                                        |
      2048 -> 4095       : 5        |**                                      |
      4096 -> 8191       : 6        |**                                      |
      8192 -> 16383      : 28       |***********                             |
     16384 -> 32767      : 59       |***********************                 |
     32768 -> 65535      : 99       |****************************************|
     65536 -> 131071     : 6        |**                                      |
    131072 -> 262143     : 2        |                                        |
    262144 -> 524287     : 1        |                                        |
```

Now other tenants are using more CPU, and this PID is throttled via CPU shares

8 - 65ms delays

BPF: Scheduler Latency 3

```
host# runqlat --pidnss -m
Tracing run queue latency... Hit Ctrl-C to end.
^C
pidns = 4026532870
     msecs               : count     distribution
         0 -> 1          : 264      |****************************************|
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 2        |                                        |

[…]
pidns = 4026532382
     msecs               : count     distribution
         0 -> 1          : 646      |****************************************|
         2 -> 3          : 18       |*                                       |
         4 -> 7          : 48       |**                                      |
         8 -> 15         : 17       |*                                       |
        16 -> 31         : 150      |*********                               |
        32 -> 63         : 134      |********                                |
```

Per-PID namespace histograms
(I added this yesterday)

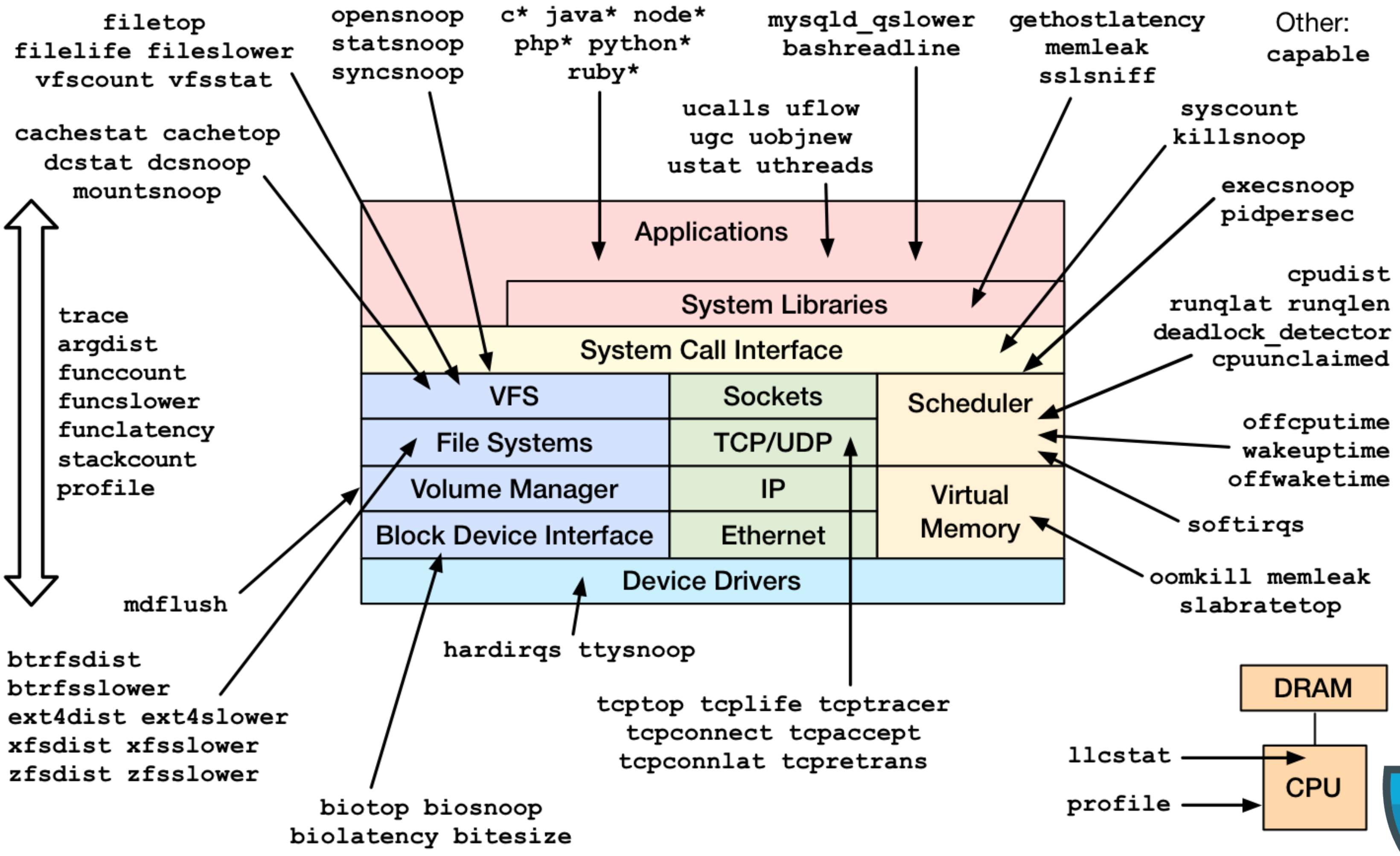
BPF: Namespace-ing Tools

Walking from the task_struct to the PID namespace ID:

```
task_struct->nsproxy->pid_ns_for_children->ns.inum
```

- This is unstable, and could break between kernel versions. If it becomes a problem, we'll add a bpf_get_current_pidns()
- Does needs a *task, or bpf_get_current_task() (added in 4.8)
- Can also pull out cgroups, but gets tricker…

bcc (BPF) Perf Tools



https://github.com/iovisor/bcc#tools 2017

Docker Analysis & Debugging

If needed, dockerd can also be analyzed using:

- go execution tracer
- GODEBUG with gctrace and schedtrace
- gdb and Go runtime support
- perf profiling
- bcc/BPF and uprobes

Each has pros/cons. bcc/BPF can trace user & kernel events.

BPF: dockerd Go Function Counting

Counting dockerd Go calls in-kernel using BPF that match "*docker*get":

```
# funccount '/usr/bin/dockerd:*docker*get*'
Tracing 463 functions for "/usr/bin/dockerd:*docker*get*"... Hit Ctrl-C to end.
^C
FUNC                                          COUNT
github.com/docker/docker/daemon.(*statsCollector).getSystemCPUUsage        3
github.com/docker/docker/daemon.(*Daemon).getNetworkSandboxID        3
github.com/docker/docker/daemon.(*Daemon).getNetworkStats        3
github.com/docker/docker/daemon.(*statsCollector).getSystemCPUUsage.func1        3
github.com/docker/docker/pkg/ioutils.getBuffer        6
github.com/docker/docker/vendor/golang.org/x/net/trace.getBucket        9
github.com/docker/docker/vendor/golang.org/x/net/trace.getFamily        9
github.com/docker/docker/vendor/google.golang.org/grpc.(*ClientConn).getTransport        10
github.com/docker/docker/vendor/github.com/golang/protobuf/proto.getbase        20
github.com/docker/docker/vendor/google.golang.org/grpc/transport.(*http2Client).getStream        30
Detaching...
# objdump -tTj .text /usr/bin/dockerd | wc -l
35859
```

35,859 functions can be traced!

Uses uprobes, and needs newer kernels. Warning: will cost overhead at high function rates.

BPF: dockerd Go Stack Tracing

Counting stack traces that led to this ioutils.getBuffer() call:

```
# stackcount 'p:/usr/bin/dockerd:*/ioutils.getBuffer'
Tracing 1 functions for "p:/usr/bin/dockerd:*/ioutils.getBuffer"... Hit Ctrl-C to end.
^C
  github.com/docker/docker/pkg/ioutils.getBuffer
  github.com/docker/docker/pkg/broadcaster.(*Unbuffered).Write
  bufio.(*Reader).writeBuf
  bufio.(*Reader).WriteTo
  io.copyBuffer
  io.Copy
  github.com/docker/docker/pkg/pools.Copy
  github.com/docker/docker/container/stream.(*Config).CopyToPipe.func1.1
  runtime.goexit
    dockerd [18176]
    110
Detaching...
```

means this stack was seen 110 times

Can also trace function arguments, and latency (with some work)

http://www.brendangregg.com/blog/2017-01-31/golang-bcc-bpf-function-tracing.html

Summary

Identify bottlenecks:

1. In the host vs container, using system metrics
2. In application code on containers, using CPU flame graphs
3. Deeper in the kernel, using tracing tools

References

- http://techblog.netflix.com/2017/04/the-evolution-of-container-usage-at.html
- http://techblog.netflix.com/2016/07/distributed-resource-scheduling-with.html
- https://www.slideshare.net/aspyker/netflix-and-containers-titus
- https://docs.docker.com/engine/admin/runmetrics/#tips-for-high-performance-metric-collection
- https://blog.docker.com/2013/10/gathering-lxc-docker-containers-metrics/
- https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon
- https://www.youtube.com/watch?v=sK5i-N34im8 Cgroups, namespaces, and beyond
- https://jvns.ca/blog/2016/10/10/what-even-is-a-container/
- https://blog.jessfraz.com/post/containers-zones-jails-vms/
- http://blog.alicegoldfuss.com/making-flamegraphs-with-containerized-java/
- http://www.brendangregg.com/USEmethod/use-linux.html full USE method list
- http://www.brendangregg.com/blog/2017-01-31/golang-bcc-bpf-function-tracing.html
- http://techblog.netflix.com/2015/11/linux-performance-analysis-in-60s.html
- http://queue.acm.org/detail.cfm?id=1809426 latency heat maps
- https://github.com/brendangregg/perf-tools ftrace tools, https://github.com/iovisor/bcc BPF tools

Thank You!

http://techblog.netflix.com
http://slideshare.net/brendangregg
http://www.brendangregg.com
bgregg@netflix.com
@brendangregg

Titus team: @aspyker @anwleung @fabiokung @tomaszbak1974 @amit_joshee @sargun @corindwyer …