Container Performance Analysis

Brendan Gregg

bgregg@netflix.com
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 Linux 4.9)
Containers at Netflix: summary slides from the Titus team.

1. TITUS
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
Current Titus Scale

- Used for ad hoc reporting, media encoding, stream processing, ...
- Over 2,500 instances (Mostly m4.16xsls & r3.8xsls) across three regions
- Over a week period launched over 1,000,000 containers
Container Performance @Netflix

• Ability to scale and balance workloads with EC2 and Titus
• 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
Namespaces: Restricting Visibility

Current Namespaces:
- cgroup
- ipc
- mnt
- net
- pid
- user
- uts
Control Groups: Restricting Usage

Current cgroups:
- blkio
- cpu,cpuacct
- cpuset
- devices
- hugetlb
- memory
- net_cls,net_prio
- pids
- ...

CPU cgroups:
- container 1
- container 2
- container 3
- cpu cgroup 1
- ...
- CPUs
- ...
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)

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

blkio (block I/O):
- **weights** (like shares)
- **IOPS/tput** caps per storage device
- statistics
CPU Shares

Container's CPU limit = 100% x \( \frac{\text{container's shares}}{\text{total busy shares}} \)

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

Container's minimum CPU limit = 100% x \( \frac{\text{container's shares}}{\text{total allocated shares}} \)

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://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)
And Container Awareness

3. HOST TOOLS
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
3.1. Host Physical Resources

A refresher of basics... Not container specific.

This will, however, solve many issues!

Containers are often not the problem.

I will demo CLI tools. GUIs source the same metrics.
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

## USE Method: Host Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Saturation</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td><code>mpstat -P ALL 1, sum non-idle fields</code></td>
<td><code>vmstat 1,&quot;r&quot;</code></td>
<td><code>perf</code></td>
</tr>
<tr>
<td>Memory Capacity</td>
<td><code>free -m, &quot;used&quot;/&quot;total&quot;</code></td>
<td>`vmstat 1,&quot;si&quot;+&quot;so&quot;; dmesg</td>
<td>grep killed</td>
</tr>
<tr>
<td>Storage I/O</td>
<td><code>iostat -xz 1, &quot;%util&quot;</code></td>
<td>`iostat -xnz 1, &quot;avgqu-sz&quot; &gt; 1</td>
<td><code>/sys/*/ioerr_cnt; smartctl</code></td>
</tr>
<tr>
<td>Network</td>
<td><code>nicstat,&quot;%Util&quot;</code></td>
<td><code>ifconfig,&quot;overrunns&quot;; netstat -s &quot;retrans...&quot;</code></td>
<td><code>ifconfig,&quot;errors&quot;</code></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>COMM</th>
<th>PID</th>
<th>TYPE</th>
<th>DEV</th>
<th>BLOCK</th>
<th>BYTES</th>
<th>LATms</th>
</tr>
</thead>
<tbody>
<tr>
<td>supervise</td>
<td>1809</td>
<td>W</td>
<td>202,1</td>
<td>17039968</td>
<td>4096</td>
<td>1.32</td>
</tr>
<tr>
<td>supervise</td>
<td>1809</td>
<td>W</td>
<td>202,1</td>
<td>17039976</td>
<td>4096</td>
<td>1.30</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8457608</td>
<td>4096</td>
<td>7.53</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8470336</td>
<td>4096</td>
<td>14.90</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8470368</td>
<td>4096</td>
<td>0.27</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8470784</td>
<td>4096</td>
<td>7.74</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8470360</td>
<td>4096</td>
<td>0.25</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8469968</td>
<td>4096</td>
<td>0.24</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8470240</td>
<td>4096</td>
<td>0.24</td>
</tr>
<tr>
<td>tar</td>
<td>14794</td>
<td>RM</td>
<td>202,1</td>
<td>8470392</td>
<td>4096</td>
<td>0.23</td>
</tr>
</tbody>
</table>
```
Event Tracing: e.g. zfsslower

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 Histogram: e.g. btrfsdist

### Latency Histogram

```
# ./btrfsdist
Tracing btrfs operation latency... Hit Ctrl-C to end.
^C
operation = 'read'

<table>
<thead>
<tr>
<th>usecs</th>
<th>count</th>
<th>distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -&gt; 1</td>
<td>192529</td>
<td>**********************</td>
</tr>
<tr>
<td>2 -&gt; 3</td>
<td>72337</td>
<td>*******************</td>
</tr>
<tr>
<td>4 -&gt; 7</td>
<td>5620</td>
<td>*</td>
</tr>
<tr>
<td>8 -&gt; 15</td>
<td>1026</td>
<td></td>
</tr>
<tr>
<td>16 -&gt; 31</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>32 -&gt; 63</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>64 -&gt; 127</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>128 -&gt; 255</td>
<td>975</td>
<td></td>
</tr>
<tr>
<td>256 -&gt; 511</td>
<td>524</td>
<td></td>
</tr>
<tr>
<td>512 -&gt; 1023</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>1024 -&gt; 2047</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2048 -&gt; 4095</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
```

From a test Titus system

** probable cache reads

** probably cache misses (flash reads)

- Histograms show modes, outliers. Also in bcc/BPF (with other FSes).
- Latency heat maps: [http://queue.acm.org/detail.cfm?id=1809426](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:

| 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 |
| dc0f3e506de45  | Titus-1392192-worker    | 27992 | /apps/spaa/spaa | 4026531835 | 4026533354 | 4026533352 | 4026532991 | 4026533355 | 4026531837 | 4026533353 |
| 370a3f041f36   | Titus-1243558-worker    | 98602 | /apps/spaa/spaa | 4026531835 | 4026533290 | 4026533288 | 4026533223 | 4026533291 | 4026533289 |
| af7549c76d9a   | Titus-1243553-worker    | 97972 | /apps/spaa/spaa | 4026531835 | 4026533216 | 4026533214 | 4026533149 | 4026533217 | 4026531837 | 4026533215 |
| dc27769a9bhc   | Titus-1243546-worker    | 97356 | /apps/spaa/spaa | 4026531835 | 4026533142 | 4026533140 | 4026533075 | 4026533143 | 4026531837 | 4026533141 |
| e18bd6198dcd   | Titus-1243517-worker    | 96733 | /apps/spaa/spaa | 4026531835 | 4026533068 | 4026533066 | 4026533001 | 4026533069 | 4026531837 | 4026533067 |
| ab45227dceae9  | Titus-1243516-worker    | 96173 | /apps/spaa/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
- https://github.com/kubernetes-incubator/cri-o/issues/868
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/dfc3a...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


<table>
<thead>
<tr>
<th>CONTAINER</th>
<th>CPU %</th>
<th>MEM USAGE / LIMIT</th>
<th>MEM %</th>
<th>NET I/O</th>
<th>BLOCK I/O</th>
<th>PIDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>353426ea09db1</td>
<td>526.81%</td>
<td>4.061 GiB / 8.5 GiB</td>
<td>47.78%</td>
<td>0 B / 0 B</td>
<td>2.818 MB / 0 B</td>
<td>247</td>
</tr>
<tr>
<td>6bf166a66e08</td>
<td>303.82%</td>
<td>3.448 GiB / 8.5 GiB</td>
<td>40.57%</td>
<td>0 B / 0 B</td>
<td>2.032 MB / 0 B</td>
<td>267</td>
</tr>
<tr>
<td>58dcf8aedd0a7</td>
<td>41.01%</td>
<td>1.322 GiB / 2.5 GiB</td>
<td>52.89%</td>
<td>0 B / 0 B</td>
<td>0 B / 0 B</td>
<td>229</td>
</tr>
<tr>
<td>61061566ffe5</td>
<td>85.92%</td>
<td>220.9 MiB / 3.023 GiB</td>
<td>7.14%</td>
<td>0 B / 0 B</td>
<td>43.4 MB / 0 B</td>
<td>61</td>
</tr>
<tr>
<td>bdc721460293</td>
<td>2.69%</td>
<td>1.204 GiB / 3.906 GiB</td>
<td>30.82%</td>
<td>0 B / 0 B</td>
<td>4.35 MB / 0 B</td>
<td>66</td>
</tr>
<tr>
<td>6c80ed61ae63</td>
<td>477.45%</td>
<td>557.7 MiB / 8 GiB</td>
<td>6.81%</td>
<td>0 B / 0 B</td>
<td>9.257 MB / 0 B</td>
<td>19</td>
</tr>
<tr>
<td>337292fb5b64</td>
<td>89.05%</td>
<td>766.2 MiB / 8 GiB</td>
<td>9.35%</td>
<td>0 B / 0 B</td>
<td>5.493 MB / 0 B</td>
<td>19</td>
</tr>
<tr>
<td>b652ede9a605</td>
<td>173.50%</td>
<td>689.2 MiB / 8 GiB</td>
<td>8.41%</td>
<td>0 B / 0 B</td>
<td>6.48 MB / 0 B</td>
<td>19</td>
</tr>
<tr>
<td>d7cd2599291f</td>
<td>504.28%</td>
<td>673.2 MiB / 8 GiB</td>
<td>8.22%</td>
<td>0 B / 0 B</td>
<td>12.58 MB / 0 B</td>
<td>19</td>
</tr>
<tr>
<td>05bf9f3e0d13</td>
<td>314.46%</td>
<td>711.6 MiB / 8 GiB</td>
<td>8.69%</td>
<td>0 B / 0 B</td>
<td>7.942 MB / 0 B</td>
<td>19</td>
</tr>
<tr>
<td>09082f005755</td>
<td>142.04%</td>
<td>693.9 MiB / 8 GiB</td>
<td>8.47%</td>
<td>0 B / 0 B</td>
<td>8.081 MB / 0 B</td>
<td>19</td>
</tr>
<tr>
<td>bd45a3e1ce16</td>
<td>190.26%</td>
<td>538.3 MiB / 8 GiB</td>
<td>6.57%</td>
<td>0 B / 0 B</td>
<td>10.6 MB / 0 B</td>
<td>19</td>
</tr>
</tbody>
</table>

[...]
In the host, top shows all processes, but currently no container IDs.

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

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

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

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?

```bash
# 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:

```bash
# 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]
[..]
```

```bash
# 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?

```bash
# 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, 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: total, 54664124 used, 11294428 free, 164232 buffers
KiB Swap: 0 total, 0 used, 0 free. 1592372 cached Mem

```

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>33.127</td>
<td>0.023</td>
<td>37564</td>
<td>S</td>
<td>537.3</td>
<td>38.2</td>
<td>40269:41 java</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>21404</td>
<td>2236</td>
<td>1812</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>4:15.11 bash</td>
</tr>
<tr>
<td>87888</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>21464</td>
<td>1720</td>
<td>1348</td>
<td>R</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00 top</td>
</tr>
</tbody>
</table>

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
```

- Symbol translation gotchas on Linux 4.13 and earlier
  - 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://batey.info/docker-jvm-flamegraphs.html](http://batey.info/docker-jvm-flamegraphs.html)
  - Can nsenter (-m -u -i -n -p) a "power" shell, and then run "perf -p PID"

- Linux 4.14 perf checks namespaces for symbol files
  - Thanks Krister Johansen
CPU Flame Graphs

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

```
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
```

Kernel TCP/IP stack
Look in areas like this to find and quantify overhead (cgroup throttles, FS layers, networking, etc). It's likely small and hard to find.

Java, missing stacks (need -XX:+PreserveFramePointer)
The best source for per-cgroup metrics. e.g. CPU:

```bash
# 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

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

These metrics should be included in performance monitoring GUIs

- https://blog.docker.com/2013/10/gathering-lxc-docker-containers-metrics/
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/NetIlix/vector
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
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.

For example, CPU performance outcomes:
A. physical CPU throttled
B. cap throttled
C. shares throttled (assumes physical CPU limited as well)
D. not throttled
CPU Bottleneck Identification

Differential Diagnosis

- **throttled_time increasing?**
  - **Y** -> **cap throttled**
  - **N**

- **nonvol...switches increasing?**
  - **N**
  - **Y** -> **host has idle CPU?**
    - **Y** (but dig further) -> **not throttled**
    - **N**

- **all other tenants idle?**
  - **N** -> **share throttled**
  - **Y** -> **physical CPU throttled**
And Container Awareness

4. GUEST TOOLS

... 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:

```bash
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
[...]
```

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

- `uptime` shows the system uptime and load average.
- `mpstat` output indicates the percentage of CPU usage for each process.
- `pidstat` shows the CPU usage for each process identified by their `PID`.

The load average is high (5.08, 3.69, 2.22), suggesting that the system is busy. 

Busy CPUs are evident, but this container is running nothing (we saw CPU usage from neighbors).
Memory

Can see host's memory:

```bash
container# free -m

Mem:
  total  used  free
   15040  1019  8381
  shared  buff/cache  available
     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

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 permitted)
perf_event_open(..., 0) failed unexpectedly with error 1 (Operation not permitted)
Error: You may not have permission to collect system-wide stats.
```

Consider tweaking `/proc/sys/kernel/perf_event_paranoid`, ...

Although tweaking perf_event_paranoid (to -1) doesn't fix it. The real problem is:

<table>
<thead>
<tr>
<th>open_by_handle_at</th>
<th>Cause of an old container breakout. Also gated by <code>CAP_DAC_READ_SEARCH</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>perf_event_open</td>
<td>Tracing/profiling syscall, which could leak a lot of information on the host.</td>
</tr>
<tr>
<td>personality</td>
<td>Prevent container from enabling BSD emulation. Not inherently dangerous, but poorly tested, potential for a lot of kernel vulns.</td>
</tr>
</tbody>
</table>
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)
  – e.g. Netflix Vector -> CPU Flame Graph
5. TRACING

... a few more examples
(iosnoop, zfsslower, and btrfsdist shown earlier)
Built-in Linux Tracers

- ftrace (2008+)
- perf_events (2009+)
- eBPF (aka BPF) (2014+)

Some front-ends:
- ftrace: [https://github.com/brendangregg/perf-tools](https://github.com/brendangregg/perf-tools)
- perf_events: used for CPU flame graphs
- eBPF (aka BPF): [https://github.com/iovisor/bcc](https://github.com/iovisor/bcc) (Linux 4.4+)
Using ftrace via my perf-tools to count function calls in-kernel context:

```
# funccount '*ovl'*
Tracing '*ovl*'... Ctrl-C to end.
^C

<table>
<thead>
<tr>
<th>FUNC</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ovl_cache_free</td>
<td>3</td>
</tr>
<tr>
<td>ovl_xattr_get</td>
<td>3</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
</tr>
<tr>
<td>ovl_fill_merge</td>
<td>339</td>
</tr>
<tr>
<td>ovl_path_real</td>
<td>617</td>
</tr>
<tr>
<td>ovl_path_upper</td>
<td>777</td>
</tr>
<tr>
<td>ovl_update_time</td>
<td>777</td>
</tr>
<tr>
<td>ovl_permission</td>
<td>1408</td>
</tr>
<tr>
<td>ovl_d_real</td>
<td>1434</td>
</tr>
<tr>
<td>ovl_override_creds</td>
<td>1804</td>
</tr>
</tbody>
</table>
```

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:

```bash
# 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

**Observability Program**
- BPF program
- BPF bytecode
- event config
- per-event data
- statistics
- output

**Kernel**
- verifier
- BPF
- maps
- static tracing
  - tracepoints
- dynamic tracing
  - kprobes
  - uprobes
- sampling, PMCs
- perf_events

**Flow**
- load
- attach
- async copy
bcc/BPF
Perf
Tools

- filetop
- filelifeslower
- vfscount
- vfsstat
- cachestat
- cachetop
- dcsnoop
dcnsnoop
- mountsnoop
- opensnoop
- statsnoop
- syncsnoop
- c* java* node*
- php* python*
ruby*
- mysqlqslower
- bashreadline
- ucalls
- uflow
- ugc
- uobjnew
- uthreads
- gethostlatency
- memleak
- sslsniff
- syscount
- killsnprintf
- execsnoop
- pidpersec
- cpudist
- runqlat
- runqlen
deadlock_detector
- cpuunclaimed
- offcputime
- wakeuptime
- offwaketime
- softirqs
- oomkill
- memleak
- slbatetop
- btrfsdist
- btrfs
- ext4dist
- ext4slower
- xfsdist
- xfs
- zfsslower
- zfs
- biotop
- biosnoop
- biolatency
- bitesize
- tcptop
tcplife
tcptracer
tcpconnect
tcpcaccept
tcpconnlat
tcpcretrans
- hardirqs
ttysnoop
- DRAM
- llcstat
- profile
- CPU
- Other: capable
BPF: Scheduler Latency

```
host# runqlat --pidns -m
Tracing run queue latency... Hit Ctrl-C to end.
^C
pidns = 4026532382
  msecs : count  distribution
  0 -> 1 :  646 |****************************************|
  2 -> 3 :   18 |*                                       |
  4 -> 7 :   48 |**                                      |
  8 -> 15 :   17 |*                                       |
 16 -> 31 :  150 |*********                               |
 32 -> 63 :  134 |********|

[...]
```

```
[...]
```

```
pidns = 4026532870
  msecs : count  distribution
  0 -> 1 :  264 |****************************************|
  2 -> 3 :    0 |                                        |
[...]`

- Shows CPU share throttling when present (eg, 8 - 65 ms)
- Currently using `task_struct->nsproxy->pid_ns_for_children->ns.inum` for `pidns`. We could add a stable `bpf_get_current_pidns()` call to BPF.

summarized in-kernel for efficiency

Per-PID namespace histograms

```
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":

```bash
# funccount '/usr/bin/dockerd:*docker*get'
Tracing 463 functions for "'/usr/bin/dockerd:*docker*get"... Hit Ctrl-C to end.
^C

<table>
<thead>
<tr>
<th>FUNC</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>github.com/docker/docker/daemon.(*statsCollector).getSystemCPUUsage</td>
<td>3</td>
</tr>
<tr>
<td>github.com/docker/docker/daemon.(*Daemon).getNetworkSandboxID</td>
<td>3</td>
</tr>
<tr>
<td>github.com/docker/docker/daemon.(*Daemon).getNetworkStats</td>
<td>3</td>
</tr>
<tr>
<td>github.com/docker/docker/daemon.(*statsCollector).getSystemCPUUsage.func1</td>
<td>3</td>
</tr>
<tr>
<td>github.com/docker/docker/pkg/ioutils.getBuffer</td>
<td>6</td>
</tr>
<tr>
<td>github.com/docker/docker/vendor/golang.org/x/net/trace.getFamily</td>
<td>9</td>
</tr>
<tr>
<td>github.com/docker/docker/vendor/google.golang.org/grpc.(*ClientConn).getTransport</td>
<td>10</td>
</tr>
<tr>
<td>github.com/docker/docker/vendor/github.com/golang/protobuf/proto.getbase</td>
<td>20</td>
</tr>
<tr>
<td>github.com/docker/docker/vendor/google.golang.org/grpc/transport.(*http2Client).getStream</td>
<td>30</td>
</tr>
</tbody>
</table>

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:

```bash
# 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).Write
  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

- 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://www.brendangregg.com/USEmethod/use-linux.html full USE method list
- http://queue.acm.org/detail.cfm?id=1809426 latency heat maps
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 …