Who stole my CPU?

Leonid Podolny
leonid@digitalocean.com
Vineeth Remanan Pillai
vineeth@digitalocean.com

Systems Engineering @ DigitalOcean

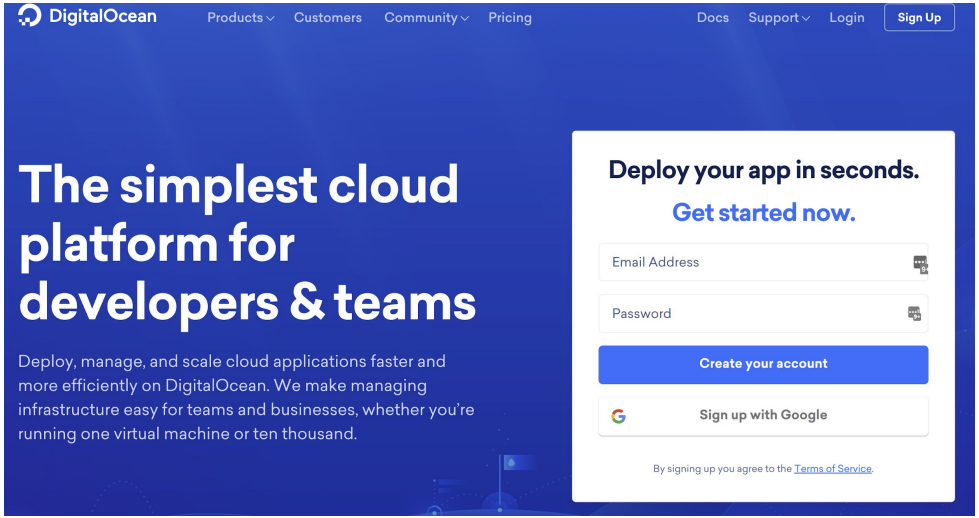

DigitalOcean

Introduction

- DigitalOcean

  Providing developers and businesses a reliable, easy-to-use cloud computing platform of virtual servers (Droplets), object storage (Spaces), and more.

- Systems Engineering
  - Responsible for the Hypervisor and its software stack
  - Host Operating System and kernel, KVM, qemu, libvirt and misc services to facilitate VM hosting



digitalocean.com

Agenda

- Steal
  - Definitions
  - Causes
  - Analysis
  - Mitigation strategies
- Mitigation approach in DigitalOcean
  - Octopus: Implementation
  - Octopus: Issues and their resolutions
    - NUMA migrations problem
    - Swapoff enhancements

What is steal?

steal, *n.* :

The fraction of time a vCPU had to wait for a physical CPU in a runqueue.

```
top - 02:09:40 up 173 days,  9:55,  2 users,  load average: 14.27, 5.04, 1.82
Tasks: 381 total,  16 running, 365 sleeping,   0 stopped,   0 zombie
%Cpu(s): 69.1 us,  9.6 sy,  0.0 ni,  3.9 id,  0.1 wa,  0.0 hi,  0.0 si, 17.4 st
KiB Mem:  32946088 total, 22525292 used, 10420796 free,   921740 buffers
KiB Swap:        0 total,        0 used,        0 free. 18607000 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
29496 root      20   0  102284  77852  18256 R  21.6  0.2   0:00.65 cc1
29596 root      20   0   84536  62332  18256 R  12.6  0.2   0:00.38 cc1
29609 root      20   0   84668  62544  18032 R  12.3  0.2   0:00.37 cc1
29636 root      20   0   73828  46388  12872 R  12.3  0.1   0:00.37 cc1
29679 root      20   0   74208  46908  14160 R   9.0  0.1   0:00.27 cc1
29676 root      20   0   72196  44344  13884 R   8.6  0.1   0:00.26 cc1
```

What is steal? *(cntd.)*

steal, *n.* :

- a vCPU is just another host task
- for a guest, the time stops
- exists only within the VM
- reported by the hypervisor

Why Steal?

- Obvious reason: More runnable vCPUs than physical CPUs
- How can we explain steal when the hypervisor has enough CPU resources to satisfy the runnable but waiting vCPUs?

Steal analysis

Steal can be analysed from two different perspectives

- VMs
  - Steal as observed from within the VM
  - Useful for determining if the steal is impacting the VM.
- Hypervisor
  - Sum of steal of individual VMs
  - Useful for determining mitigation approaches

Steal as seen from the VM

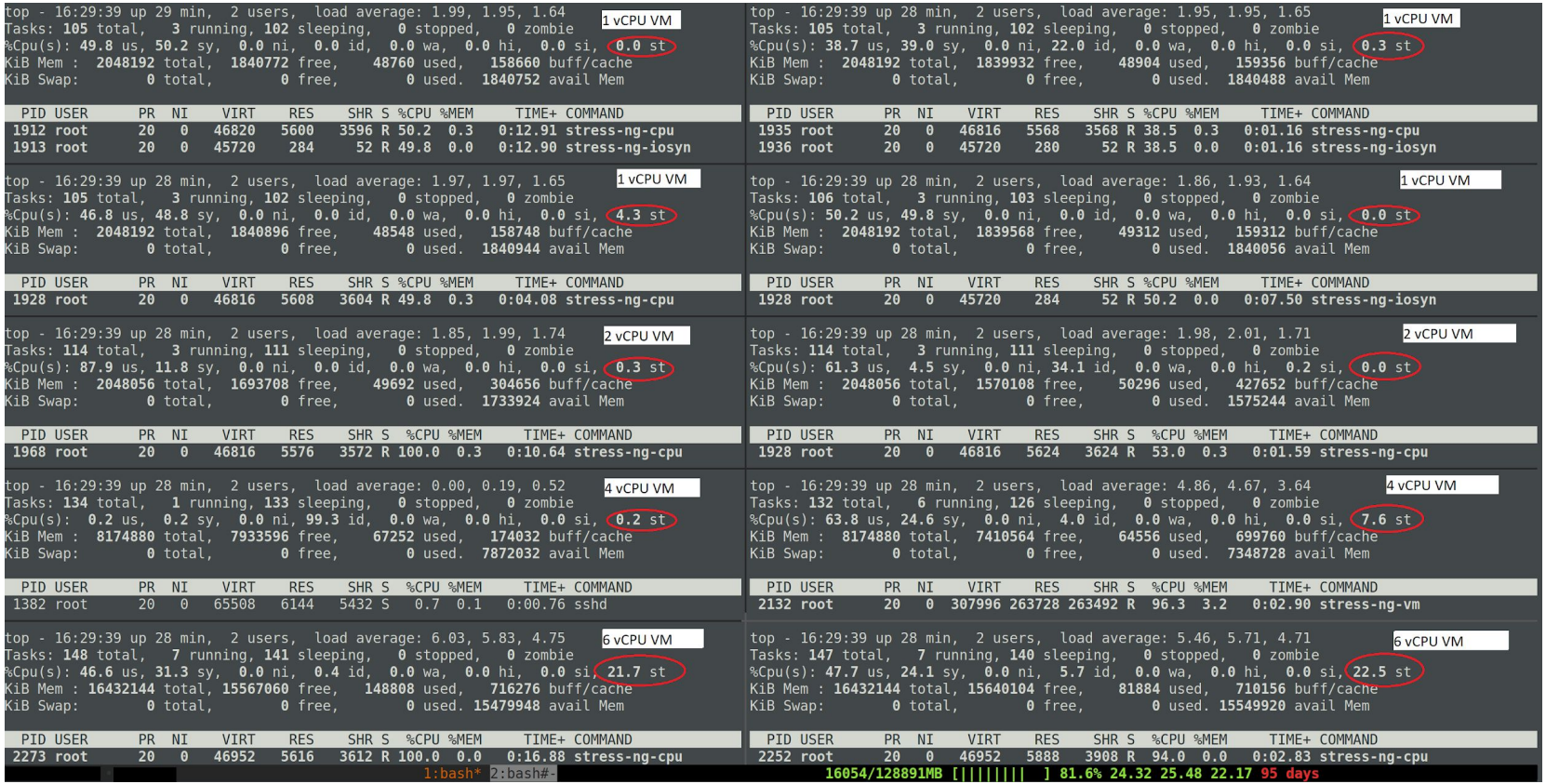
- Busy Steal
  - CPU utilization + steal is close to 100%
  - VM could have made use of the stolen time had it not been stolen
- Idle Steal
  - Idle VM experiencing steal: utilization + steal is significantly below 100%
  - VM could not have used the stolen time even if available.

Steal as seen from Hypervisor

Total Steal experienced by all VMs in a Hypervisor

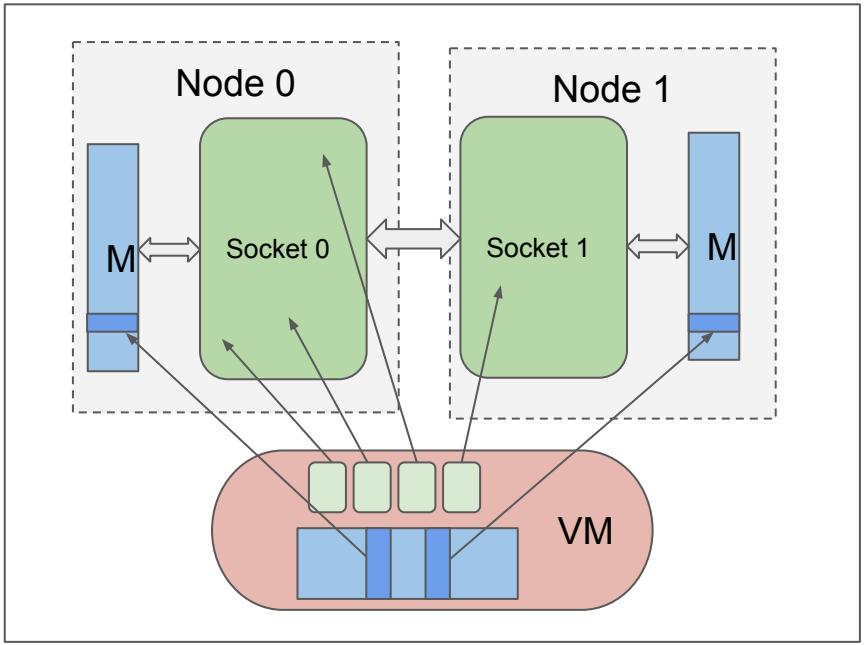
- Busy Steal
  - More runnable vCPUs than physical CPUs
  - Not mitigatable in software
  - Migrating VMs out of the busy HV is the probable solution
- Idle Steal
  - Caused due to scheduler limitations, config issues etc
  - Mitigatable in software

Hypervisor “idle” steal



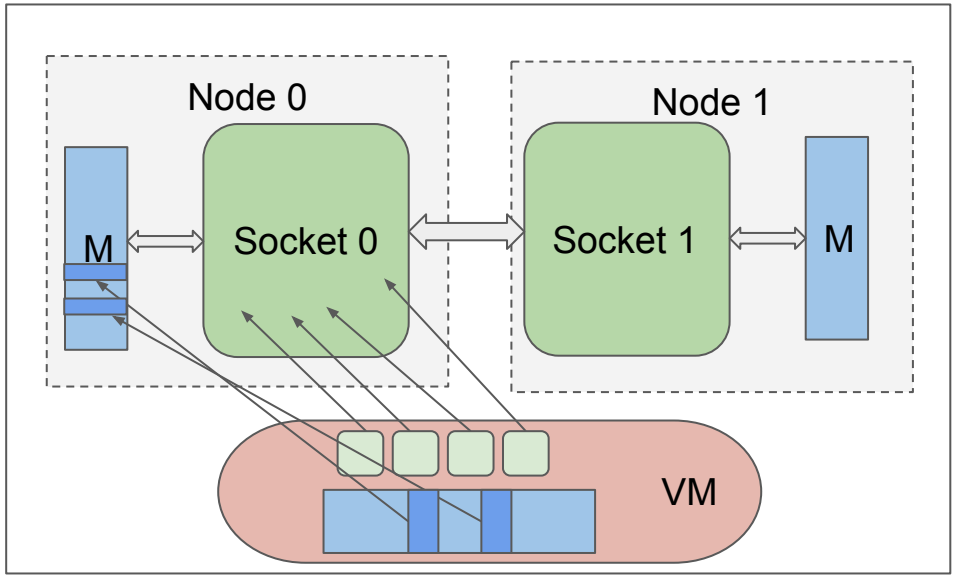
Hypervisor “idle” steal: NUMA balancing

- VMs span all the NUMA nodes by default.
- Linux automatic NUMA balancing: keep tasks closer to their memory
  - Memory follow CPU model
  - CPU follow memory model
- Migration threads takes up cpu, resulting in steal



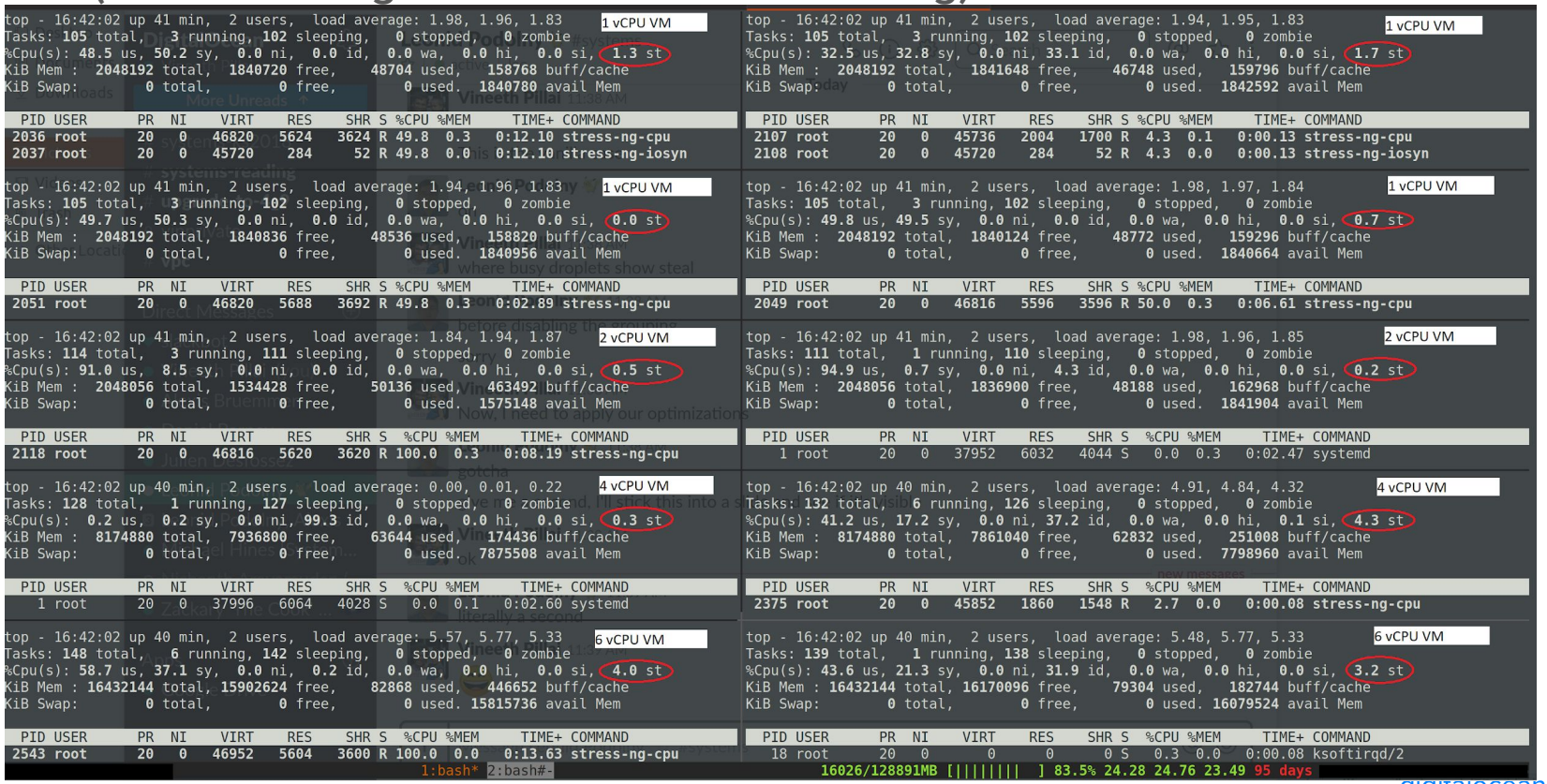
Hypervisor “idle” steal: NUMA balancing

- Mitigation
  - Pin VMs to NUMA nodes and
  - Disable NUMA balancing



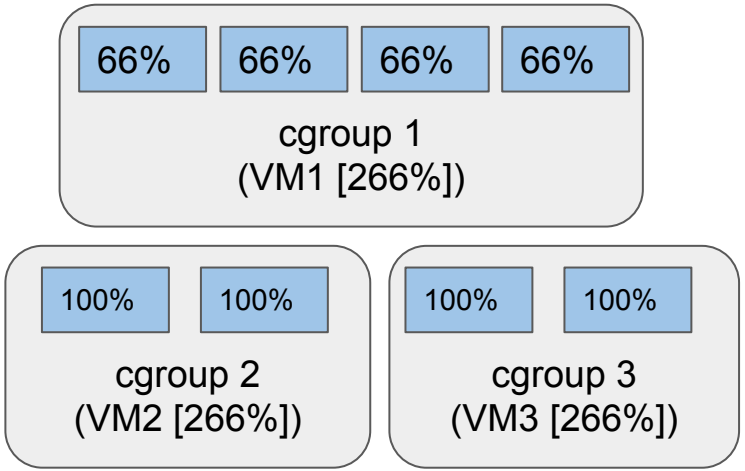
Hypervisor “idle” steal

(After disabling automatic NUMA balancing)



Hypervisor “idle” steal: process grouping

- Cgroups
  - Default cgroups created by libvirt is per-VM
  - Bigger VMs are considered equal in weight to smaller VMs due to per-VM cgroups
- Example: 8 cpu Hypervisor
  - 8 cpu (800%)
  - 3 VMs
    - 1x 4-vCPU
    - 2x 2-vCPU



Hypervisor “idle” steal: process grouping *(cntd.)*

- Mitigation
  - Disable CPU cgroups for VMs
- Side Effects
  - Loses the capability to control VM cpu utilization
  - Autogroup feature kicks in
  - Disabling autogroup feature works fine for newly launched VMs, but running VMs are still managed by the autogroup

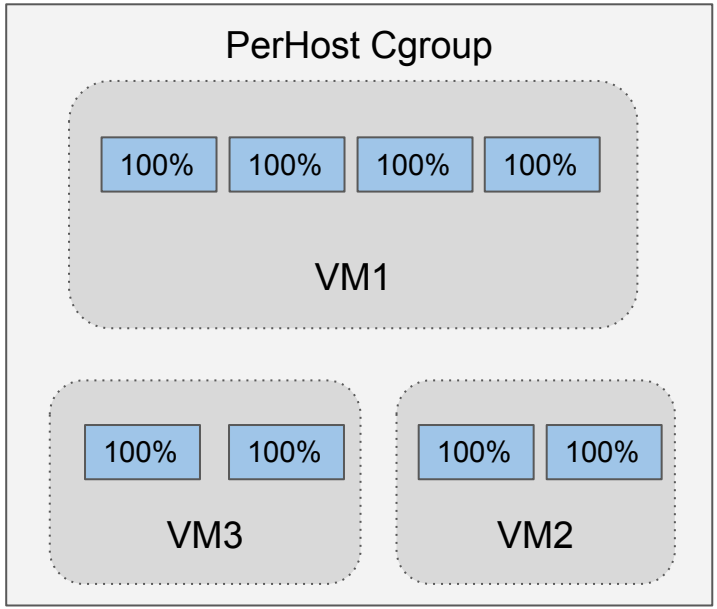
Hypervisor “idle” steal: process grouping (Contd...)

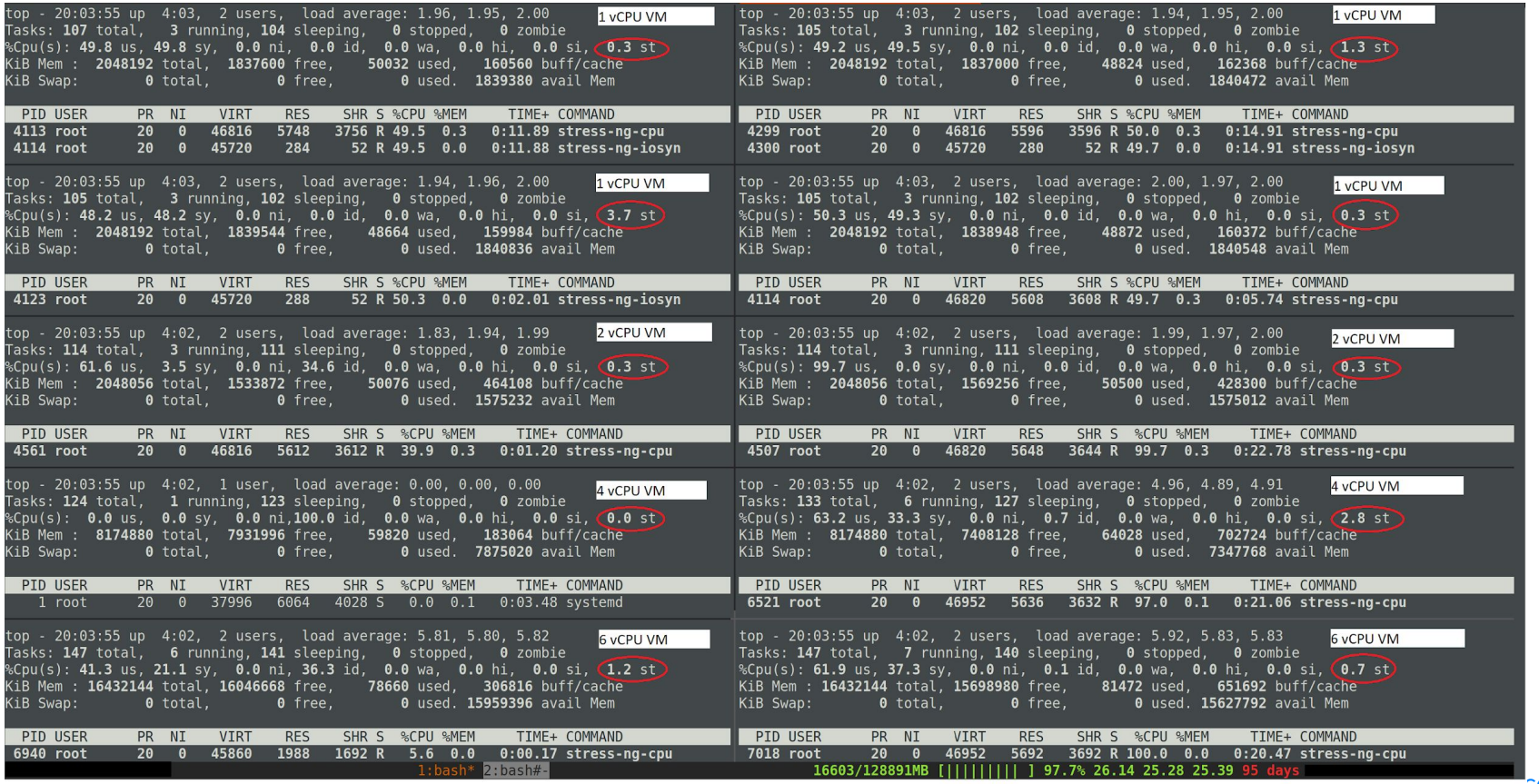
- Working solution: per-host VM Cgroup
  - Consolidate all vCPUs in one cgroup.
  - CFS allocates CPU proportionally to the number of vCPUs.
- Modified libvirt
  - A tunable to consolidate all vCPUS to one cpu cgroup and all cgroups parameters are same(default) for all VMs
    - `cfs.cfs_period_us, cfs.cfs_quota_us`
  - Moves a VM to its own cgroup if any cgroup parameters modified for a VM
  - Moves the VM back to perhost cgroup if the cgroup parameters are reverted to default

Hypervisor “idle” steal: process grouping (Contd...)

per-host Cgroup Example

- 8 cpu Hypervisor
  - 8 cpu (800%)
  - 3 VMs
    - 1x 4-vCPU
    - 2x 2-vCPU





Hypervisor “idle” steal: per-host cgroup

1 vCPU VM

```
top - 20:03:55 up  4:03,  2 users,  load average: 1.96, 1.95, 2.00
Tasks: 107 total,   3 running, 104 sleeping,   0 stopped,   0 zombie
%Cpu(s): 49.8 us, 49.8 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
KiB Mem :  2048192 total,  1837600 free,    50032 used,   160560 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1839380 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 4113 root      20   0   46816   5748   3756 R 49.5  0.3   0:11.89 stress-ng-cpu
 4114 root      20   0   45720    284     52 R 49.5  0.0   0:11.88 stress-ng-iosyn
```

1 vCPU VM

```
top - 20:03:55 up  4:03,  2 users,  load average: 1.94, 1.95, 2.00
Tasks: 105 total,   3 running, 102 sleeping,   0 stopped,   0 zombie
%Cpu(s): 49.2 us, 49.5 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  1.3 st
KiB Mem :  2048192 total,  1837000 free,    48824 used,   162368 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1840472 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 4299 root      20   0   46816   5596   3596 R 50.0  0.3   0:14.91 stress-ng-cpu
 4300 root      20   0   45720    280     52 R 49.7  0.0   0:14.91 stress-ng-iosyn
```

1 vCPU VM

```
top - 20:03:55 up  4:03,  2 users,  load average: 1.94, 1.96, 2.00
Tasks: 105 total,   3 running, 102 sleeping,   0 stopped,   0 zombie
%Cpu(s): 48.2 us, 48.2 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  3.7 st
KiB Mem :  2048192 total,  1839544 free,    48664 used,   159984 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1840836 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 4123 root      20   0   45720    288     52 R 50.3  0.0   0:02.01 stress-ng-iosyn
```

1 vCPU VM

```
top - 20:03:55 up  4:03,  2 users,  load average: 2.00, 1.97, 2.00
Tasks: 105 total,   3 running, 102 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.3 us, 49.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
KiB Mem :  2048192 total,  1838948 free,    48872 used,   160372 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1840548 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 4114 root      20   0   46820   5608   3608 R 49.7  0.3   0:05.74 stress-ng-cpu
```

2 vCPU VM

```
top - 20:03:55 up  4:02,  2 users,  load average: 1.83, 1.94, 1.99
Tasks: 114 total,   3 running, 111 sleeping,   0 stopped,   0 zombie
%Cpu(s): 61.6 us,  3.5 sy,  0.0 ni, 34.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
KiB Mem :  2048056 total,  1533872 free,    50076 used,   464108 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1575232 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4561 root      20   0   46816   5612   3612 R  39.9  0.3   0:01.20 stress-ng-cpu
```

2 vCPU VM

```
top - 20:03:55 up  4:02,  2 users,  load average: 1.99, 1.97, 2.00
Tasks: 114 total,   3 running, 111 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.7 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
KiB Mem :  2048056 total,  1569256 free,    50500 used,   428300 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1575012 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4507 root      20   0   46820   5648   3644 R  99.7  0.3   0:22.78 stress-ng-cpu
```

4 vCPU VM

```
top - 20:03:55 up  4:02,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 124 total,   1 running, 123 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  8174880 total,  7931996 free,    59820 used,   183064 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  7875020 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0   37996   6064   4028 S   0.0  0.1   0:03.48 systemd
```

4 vCPU VM

```
top - 20:03:55 up  4:02,  2 users,  load average: 4.96, 4.89, 4.91
Tasks: 133 total,   6 running, 127 sleeping,   0 stopped,   0 zombie
%Cpu(s): 63.2 us, 33.3 sy,  0.0 ni,  0.7 id,  0.0 wa,  0.0 hi,  0.0 si,  2.8 st
KiB Mem :  8174880 total,  7408128 free,    64028 used,   702724 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  7347768 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6521 root      20   0   46952   5636   3632 R  97.0  0.1   0:21.06 stress-ng-cpu
```

6 vCPU VM

```
top - 20:03:55 up  4:02,  2 users,  load average: 5.81, 5.80, 5.82
Tasks: 147 total,   6 running, 141 sleeping,   0 stopped,   0 zombie
%Cpu(s): 41.3 us, 21.1 sy,  0.0 ni, 36.3 id,  0.0 wa,  0.0 hi,  0.0 si,  1.2 st
KiB Mem : 16432144 total, 16046668 free,    78660 used,   306816 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 15959396 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6940 root      20   0   45860   1988   1692 R   5.6  0.0   0:00.17 stress-ng-cpu
```

6 vCPU VM

```
top - 20:03:55 up  4:02,  2 users,  load average: 5.92, 5.83, 5.83
Tasks: 147 total,   7 running, 140 sleeping,   0 stopped,   0 zombie
%Cpu(s): 61.9 us, 37.3 sy,  0.0 ni,  0.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
KiB Mem : 16432144 total, 15698980 free,    81472 used,   651692 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 15627792 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 7018 root      20   0   46952   5692   3692 R 100.0  0.0   0:20.47 stress-ng-cpu
```

```
1:bash* 2:bash#-                                  16603/128891MB [||||||||| ] 97.7% 26.14 25.28 25.39 95 days
```

Sidenote: CFS & Idle VMs

- On a fully utilized hypervisor, idle VMs might experience steal because they are penalized by CFS.
- When idle VMs share a hypervisor with busy VMs, they will be assigned a lower weight as their utilization is low.
    - Results in scheduler latency, when competing with busy VM vCPUs.

Octopus

- Userspace VM-placement daemon
  - Pins VMs to resources (CPUs, NUMA nodes)
  - As a result, occasionally migrates VMs across sockets
- VM awareness in vCPUs placement
- jiffy-level resolution not needed
- Functionality:
  - NUMA-partitioning
  - utilization tracking on overcommitted fleet
  - CPU-to-vCPU mapping on optimized HVs

Octopus

- Issues
  - swap usage during NUMA migration
  - OOMs when VM aggressively allocates RAM:
    - when VM allocates RAM faster than RAM is swapped to disk
    - at swapoff phase, when swap is disabled

Swapoff optimizations

- Swapoff implementation in kernel is not efficient
  - Goes through all process address space for each swap entry
  - For considerably large swap and a heavily loaded hypervisor with thousands of processes, might take hours or days.
- Usually swapoff happens during system shutdown
- We use it to migrate VMs between NUMA nodes and need it to be quick.
- Revived a dormant patch and initiated discussions upstream.
  - https://lkml.org/lkml/2018/10/3/638
  - Basic approach is to give a single pass on all process address space and page in the swapped out pages.

Failing NUMA migrations

- The process of migrating the RAM of VM between NUMA nodes might fail, even with swap to back it.
  - A dangerous side effect is OOM kill of VMs
- The placement service makes its best effort to avoid OOM by tuning the swappiness and pre-calculating available memory and swap before the migration.
  - Dynamic nature of the hypervisor breaks this approach
  - VM launches and destroys causes all calculations to go wrong
- Efforts started inhouse to have a kernel level mechanism to do a safe memory migration without OOM and fail gracefully.
  - migrate_pages(2)

Questions?

Thank you!