Deterministic Container Resource Management in Derivative Clouds

Chandra Prakash, Prashanth, Umesh Bellur, Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

19th April, 2018

International Conference on Cloud Engineering
Derivative cloud

Customers

Request Cloud service

Provide Cloud service

Rent VMs

Native IaaS Cloud
(Amazon EC2, Google, etc.)

VM

Memory: 1 GB
CPU: 1
Cost: $1.2/hour

Memory: 10 GB
CPU: 10
Cost: $10/hour
Dual control over resources

- Hypervisor and guest OS both control the same resources
- Hypervisor not aware of containers requirements
Hypervisor mechanism under consideration

- **Ballooning** is used to achieve memory overcommitment\(^1\)
- **vCPU scaling** is used to reduce scheduling overheads in over committed situation \(^2\)

---


Undesirable effects due to ballooning

Happens

Desired
Existing memory reclamation in containers

- Memory provisioning knobs: *Hard-Limit* and *Soft-Limit*
- *exceed*: difference between memory usage and Soft-Limit
- **SMR** (Soft Memory Reclaimed): memory reclaimed from local LRU
- **GLR** (Global LRU Reclaimed): memory reclaimed from global LRU
Existing memory reclamation in containers

- **Memory Reclamation**
  - Triggered on containers reaching their hard limits
  - **Container specific Reclamation**
    - Reclaims memory using container specific LRU
  - **System Wide Reclamation**
    - Triggered on system reaching its memory limits
    - Reclaims memory from container that is exceeding the most (SMR)
    - Reclaims from global LRU (GLR)

Repeat the two until request is satisfied.
Impact of ballooning

Memory reclamation rate: 2 GB every 30 seconds (generated from host after 100 seconds)
Impact of ballooning

Default configuration of four containers

<table>
<thead>
<tr>
<th>Container</th>
<th>Hard-limit(GB)</th>
<th>Soft-limit(GB)</th>
<th>Key size (# records)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redis-Low</td>
<td>2</td>
<td>0.5</td>
<td>500K</td>
</tr>
<tr>
<td>Redis-High</td>
<td>4</td>
<td>1</td>
<td>1000K</td>
</tr>
<tr>
<td>Mongo-Low</td>
<td>2</td>
<td>0.5</td>
<td>500K</td>
</tr>
<tr>
<td>Mongo-High</td>
<td>4</td>
<td>1</td>
<td>1000K</td>
</tr>
</tbody>
</table>

Memory usage ratio

Time (sec)
Impact of ballooning

- Existing knobs (limits) do not guarantee proportionate memory allocation during memory pressure situations
CPU provisioning Issues

Container configuration
Pin container to CPU --> cpuset.cpus
Maintain container cpu share --> cpu.share

Hypervisor optimization
Reduce scheduling overhead --> Dynamic vCPU scaling

Hypervisor unaware of cpuset.cpus & cpu.share
Container requirement disturbed
Impact of vCPU scaling

Experimental setup

<table>
<thead>
<tr>
<th>VM configuration</th>
<th>7 vCPUs and 8GB Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of containers inside VM</td>
<td>3</td>
</tr>
<tr>
<td>CPU allocation ratio</td>
<td>1:1:4</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Sysbench</td>
</tr>
<tr>
<td>vCPU scaling down frequency</td>
<td>1 vCPU every 120s (vCPU1,2,3,&amp;4)</td>
</tr>
<tr>
<td>vCPU mapping using cpuset.cpus</td>
<td>C1: vCPU1, C2: vCPU2, C3: vCPU3,4,5,&amp;6</td>
</tr>
</tbody>
</table>

### CPU utilization without pinning

![Graph showing CPU utilization without pinning](image)

### CPU utilization with pinning

![Graph showing CPU utilization with pinning](image)
Impact of vCPU scaling

- Pinning and scaling $\implies$ non-deterministic CPU utilization
- **Desired goal:** achieve pinning benefits + maintain CPU share

![CPU utilization without pinning](image1)

![CPU utilization with pinning](image2)
Summary of issues in nesting setup

- Ballooning may fail to satisfy container requirements
- vCPU scaling may not respect cpu share with cpu pinning
Our approach

- Native cloud provider can be public or private
- We can’t control or change hypervisor in case of public cloud
- We provide solution at guest OS level
Proposed memory policies

Proportionate memory allocation
- Allocate memory according to credit share of containers

Application-specific differentiated memory reclamation
- Protect memory sensitive container(s) from memory reclamation
Proposed CPU policies

Maximize dedicated vCPU while maintaining allocation ratio
▶ To get maximum benefits of pinning

Provide pinned vCPU(s) to a subset of containers
▶ Based on application nature or user requirement
Contribution

- Modified the memory reclamation logic in *memory cgroup* subsystem

- Provided an additional definition of *exceed*

- Performed several modifications in Linux kernel
  - Added extra parameters in *memory* and *cpu cgroups*
  - Added control to maximize SMR
  - Provided knob to control reclamation chunk size

- Created a *cpuset* calculator in user space
Modified memory reclamation

- **Triggers**:
  - Containers reaching their hard limits
  - System reaching its memory limits

- **Reclamation**:
  - Container specific
    - Reclaims memory using container specific LRU
  - System wide
    - If exceed of all containers is less than or equal to 0:
      - Recalculate exceed according to new definition
      - Reclaim from global LRU
    - Otherwise:
      - Reclaim memory from the container that is exceeding the most
      - Repeat until request is satisfied

- **Equation**:
  - \( \text{exceed} = \text{memory}_\text{usage} - \text{proportionate}_\text{share} \)
Effectiveness of memory policies

- Ratio of memory weights: 1:2

Able to maintain memory usage ratio when *exceed of all containers* become less than or equal zero (after 300 second)
Memory is not reclaimed from YCSB application container (memory sensitive) and it’s throughput remains intact
vCPU reallocation design

- Change Notifier
  - If no change in vCPUs (Periodic Loop)
  - If change noticed in vCPUs
- Available vCPUs
- vCPU allotment to each container
- cpuset calculator
- Update cpuset
- Find vCPU share of each container
- Allocate dedicated vCPUs to each container equal to integer part of share.
- If vCPU share of a container ≠ integer
- Allocate remaining vCPUs to container.

### Container 1
- cpu.share: 2
- vCPU share: 5/3 (1.66)
- #Dedicated vCPUs: 1
- cpuset.cpus: [1, 5]

### Container 2
- cpu.share: 4
- vCPU share: 10/3 (3.33)
- #Dedicated vCPUs: 3
- cpuset.cpus: [2, 3, 4, 5]

#vCPUs: 5 [1, 2, 3, 4, 5]
Effectiveness of vCPU reallocation

Experimental setup

<table>
<thead>
<tr>
<th>VM configuration</th>
<th>7 vCPUs and 8GB Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of containers inside VM</td>
<td>3</td>
</tr>
<tr>
<td>CPU allocation ratio</td>
<td>1:4:5</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Sysbench &amp; Twitter</td>
</tr>
<tr>
<td>VCPU scaling down frequency</td>
<td>1 vCPU every 120s (vCPU1,2,3,4)</td>
</tr>
</tbody>
</table>

CPU utilization by each container

Twitter throughput with scaling down vCPUs

- Able to maintain CPU share along with pinning
Conclusion:

- Quantified the impact of hypervisor actions on containers running inside VM
- Proposed user-defined policies to mitigate the impact of hypervisor actions
- Demonstrated the effectiveness of memory and CPU policies empirically

Future work:

- Design an efficient algorithm for container placement in derivative (nested) setup
Thank you
Questions???

Email id: chandrap@cse.iitb.ac.in
Modifications in Linux kernel

- Added a *weight* parameter in *memory* cgroup and a *pin* parameter in *cpu* cgroup.

- Modified the `balance_pgdat()` routine (Linux kernel version 4.7).

```
Listing 1: Original reclamation logic

For every reclamation request:
  SMR();
  GLR();
```

```
Listing 2: Modified reclamation logic

For every reclamation request:
  NoOfReclaimedPages = SMR();
  if (NoOfReclaimedPages == 0):
    GLR();
```

- Created a kernel module to control the reclamation chunk size.