Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments

Seyyed Ahmad Javadi, Amogha Suresh, Muhammad Wajahat, Anshul Gandhi

November 22, 2019
Cloud Computing

Cloud providers
Operate cloud infrastructures
Great budget expenditure for:
  • Data center equipment
  • Power provisioning

Tenants
Rent Virtual Machines (VMs)
Cloud Computing

Cloud providers
Operate cloud infrastructures
Great budget expenditure for:
• Data center equipment
• Power provisioning

Tenants
Rent Virtual Machines (VMs)

➢ Virtual resources might be provisioned (via tenants) for peak load
➢ Tenants’ VM placement (via providers) is challenging
Low Resource Utilization in Cloud Environments

CDF of average CPU and memory usage, Alibaba cluster trace (2018).

Cumulative probability, $F(x)$

$X = \text{Average usage}$

fg = foreground/online workload
Low Resource Utilization in Cloud Environments

CDF of average CPU and memory usage, Alibaba cluster trace (2018).

Time (days)

CPU utilization (%)

Cumulative probability, $F(x)$

X = Average usage

VM-level CPU usage for the Azure trace (2017).

fg = foreground/online workload

$F(x)$
Low Resource Utilization in Cloud Environments

Cumulative probability, \( F(x) \)

CPU utilization (%)

Great opportunity to use cloud idle resources

\( fg = \) foreground/online workload
CDF of average CPU and memory usage, Alibaba cluster trace (2018).

Cumulative probability, $F(x)$

$X = $ Average usage
Opportunity: Running Background Batch Workload

Key challenge: Resource contention

- May violate SLOs of *foreground*
  *dynamic workload*
- Foreground workload is a *black-box*, SLOs not known

CDF of average CPU and memory usage, Alibaba cluster trace (2018).

Cumulative probability, $F(x)$

$X = $ Average usage

$bg = $ background/batch workload
Problem statement: How to schedule background batch jobs to improve utilization without hurting black-box foreground performance?

Key challenge: Resource contention

- May violate SLOs of foreground dynamic workload
- Foreground workload is a black-box, SLOs not known
Outline

➢ Prior approaches

➢ Our approach: Scavenger
  • High-level idea
  • Resource regulation algorithm
  • Evaluation methodology
  • Evaluation results

➢ Conclusion
Prior approaches

- Treat foreground as white-box (assume SLO is known)
  - Bistro (ATC’15, Facebook)
  - Heracles (ISCA’15, Google)
  - History-based harvesting (OSDI’16, Microsoft)
  - PARTIES (ASPLOS ‘19, SAIL group-Cornell Uni.)
- Typically focus only on one resource (need some critical profiling)
  - dCat (EuroSys’18, IBM)
  - Perflso (ATC’18, Microsoft)
    - Reprofiles often if workload changes
Our approach: Scavenger

- Considers foreground workloads as a **black-box**
- Takes **multiple resources** (processor, memory, nw) into account
- Is a dynamic and tunable solution
- Uses container as the **agile execution environment** for batch jobs
Scavenger Daemon

- Background resource regulation is the main design decision
  - Dealing with resource contention is challenging

Using Linux’s cpuset cgroups

Ubuntu 16.04, KVM, Docker
Scavenger Daemon

- Background resource regulation is the main design decision
  - Dealing with resource contention is challenging

Using Linux’s cpuset cgroups

CPU Cores
VM: Web serving
Container: DCopy
Last Level Cache (LLC)

Ubuntu 16.04, KVM, Docker

Background CPU usage (%)

95th percentile RT degradation (%)
Scavenger Daemon

- Background resource regulation is the main design decision
  - Dealing with resource contention is challenging

Using Linux’s `cpuset` cgroups

<table>
<thead>
<tr>
<th>VM</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web serving</td>
<td>DCopy</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

CPU Cores

Last Level Cache (LLC)

Ubuntu 16.04, KVM, Docker

95%ile RT degradation (%)

Instruction Per Cycle (IPC) degradation(%)

Background CPU usage (%)

- Using Linux’s `cpuset` cgroups
Scavenger Daemon

- Background resource regulation is the main design decision
  - Dealing with resource contention is challenging

Ubuntu 16.04, KVM, Docker

Using Linux’s `cpuset cgroups`

- IPC is used as performance proxy
  - 95%ile RT degradation (%)
Resource Regulation Algorithm

- Scavenger determines availability of resources for bg jobs
  - Background CPU load (cgroups)
    - CPU quota (maximum CPU cycles given to a process under the CFS)
  - Memory capacity (libvit)
  - Network bandwidth (TC)
Resource Regulation Algorithm

Our generic online algorithm

- Monitor VMs’ perf metric (e.g., memory usage) for window-size
- Calculate mean, $\mu$, and standard deviation, $\sigma$
- React based on the VMs’ perf metric and $\mu \pm c \cdot \sigma$

Simplified illustration

Normalized metric value [memory usage, network usage]

$\text{Reactive Window: } bg = 1 - (\mu + c \cdot \sigma)$
Evaluation Methodology

- Scavenger prototype implementation
  - Largely written in C++ and shell script (~750 lines of code)
Evaluation Methodology

- Scavenger prototype implementation
  - Largely written in C++ and shell script (~750 lines of code)

<table>
<thead>
<tr>
<th>Foreground</th>
<th>Training</th>
<th>CloudSuite</th>
<th>Widely used benchmark suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>TailBench</td>
<td></td>
<td>Designed for latency-critical applications</td>
</tr>
<tr>
<td>Background (SparkBench)</td>
<td>KMeans</td>
<td></td>
<td>A popular clustering algorithm</td>
</tr>
<tr>
<td></td>
<td>SparkPi</td>
<td></td>
<td>Computes Pi with very high precision</td>
</tr>
</tbody>
</table>
Evaluation Methodology

- Scavenger prototype implementation
  - Largely written in C++ and shell script (~750 lines of code)

<table>
<thead>
<tr>
<th>Foreground</th>
<th>Training</th>
<th>CloudSuite</th>
<th>Widely used benchmark suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>TailBench</td>
<td></td>
<td>Designed for latency-critical applications</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Background (SparkBench)</th>
<th>KMeans</th>
<th>A popular clustering algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SparkPi</td>
<td>Computes Pi with very high precision</td>
</tr>
</tbody>
</table>

Sensitivity analysis  Experimental evaluation
The load generators employed in TailBench are open-loop.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Domain</th>
<th>Tail latency scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xapian</td>
<td>Online search</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>Moses</td>
<td>Real-time translation</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>Silo</td>
<td>In-memory database (OLTP)</td>
<td>Microseconds</td>
</tr>
<tr>
<td>Specjbb</td>
<td>Java middleware</td>
<td>Microseconds</td>
</tr>
<tr>
<td>Masstree</td>
<td>Key-value store</td>
<td>Microseconds</td>
</tr>
<tr>
<td>Shore</td>
<td>On-disk database (OLTP)</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>Sphinx</td>
<td>Speech recognition</td>
<td>Seconds</td>
</tr>
<tr>
<td>Img-dnn</td>
<td>Image recognition</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

Cloud Testbed

- 250GB DRAM
- 10 Gb/s network
- KVM, Docker
- Ubuntu 16.04

<table>
<thead>
<tr>
<th>Processor socket 0</th>
<th>Processor socket 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC of size 25MB</td>
<td>LLC of size 25MB</td>
</tr>
<tr>
<td>250GB DRAM</td>
<td>10 Gb/s network</td>
</tr>
</tbody>
</table>

PM₁
PM₂

Resource Manager, Name Node, Data Node
Cloud Testbed

<table>
<thead>
<tr>
<th>VM₁</th>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>250GB DRAM</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor socket 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC of size 25MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor socket 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC of size 25MB</td>
</tr>
</tbody>
</table>

| 10 Gb/s network |

KVM, Docker, Ubuntu 16.04

Resource Manager, Name Node, Data Node

PM₁

PM₂
Cloud Testbed

VM\textsubscript{1} | Background
---|---
0 | 1 | 2 | 3 | 4
5 | 6 | 7 | 8 | 9

LLC of size 25MB
Processor socket 0
250GB DRAM

VM\textsubscript{2} | Background
---|---
0 | 1 | 2 | 3 | 4
5 | 6 | 7 | 8 | 9

LLC of size 25MB
Processor socket 1
10 Gb/s network

KVM, Docker, Ubuntu 16.04

Resource Manager, Name Node, Data Node
Outline

➢ Prior approaches

➢ Our approach: Scavenger
  • High-level idea
  • Resource regulation algorithm
  • Evaluation methodology

➢ Evaluation results

➢ Conclusion
Evaluation with Spark jobs as background

95%ile latency degradation (%) Better bg: SparkPi
Evaluation with Spark jobs as background

<table>
<thead>
<tr>
<th>VM1 Workload II VM2 Workload</th>
<th>bg: SparkPi</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Memory</td>
</tr>
<tr>
<td>43%↑</td>
<td>201%↑</td>
</tr>
</tbody>
</table>

95%ile latency degradation (%)

Better

Better

Better

Better

Better
Evaluation with Spark jobs as background

VM₁ Workload vs VM₂ Workload

95%ile latency degradation (%)

Better

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>43%↑</td>
<td>201%↑</td>
</tr>
</tbody>
</table>

bg: SparkPi

bg: KMeans
Evaluation with Spark jobs as background

VM1 Workload vs VM2 Workload

<table>
<thead>
<tr>
<th>Workload</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>bg: SparkPi</td>
<td>43%↑</td>
<td>201%↑</td>
</tr>
<tr>
<td>bg: KMeans</td>
<td>34%↑</td>
<td>321%↑</td>
</tr>
</tbody>
</table>

95%ile latency degradation (%)
Better

Better
Media-streaming as foreground

Lab testbed: 2-vCPU foreground VM, 2-core background container.
Scavenger outperforms static approaches while affording higher background usage.

Lab testbed: 2-vCPU foreground VM, 2-core background container.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>37%↑</td>
<td>180Mbps ↑</td>
</tr>
</tbody>
</table>

Scavenger outperforms static approaches while affording higher background usage.
Limit Study With DCopy as the Background

Cloud testbed: 4-vCPU foreground VM, 6-core background DCopy container.

Normalized 95%ile latency

Better

The diagram shows the normalized 95%ile latency for various workloads: xapian, moses, silo, specjbb, masstree, shore, sphinx, and img-dnn. The X-axis represents the different workloads, and the Y-axis represents the latency values.

- **No background** is represented by blue bars.
- **Baseline** is represented by red bars.
- **Scavenger** is represented by yellow bars.

The results indicate that DCopy as a background task improves the normalized 95%ile latency for all workloads compared to the baseline and no background scenarios.
Limit Study With DCopy as the Background

Cloud testbed: 4-vCPU foreground VM, 6-core background DCopy container.

normalized 95%ile latency

Better

Scavenger can successfully and aggressively regulate bg workload to mitigate its impact on fg performance.

3-5% CPU ↑

Normalized 95%ile latency

No background
Baseline
Scavenger
Conclusion

- Significant opportunity to use cloud idle resource
- Important features of cloud tenant’s VM workloads
  - Black-box, SLOs not known
  - Dynamic behavior
- Scavenger: Dynamic, black-box multi-resource manager
  - Does not instrument or profile the tenant VMs offline.
  - Increases server utilization without compromising the resource demands of tenant VMs.
Thank You 😊

Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments

Q&A

Seyyed Ahmad Javadi
sjavadi@cs.stonybrook.edu
PACE Lab at Stony Brook University
ACM Symposium on Cloud Computing 2019