Semeru
A Memory-Disaggregated Managed Runtime

Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim and Harry Xu

UCLA, MIT, Texas A&M University, Ohio State University, UCLA
Disaggregated Datacenter

CPU Servers
- small memory

Network (e.g., InfiniBand)
- Reliable network
- Micro-second latency
- Tens of GBs bandwidth

Memory Servers
- weak cores

Storage Servers
- weak cores
Process Execution Model

CPU Server

CPU

Local Memory(cache)

Swap

Memory Server

Memory Server

Memory Server

RDMA over InfiniBand

Process
Process Execution Model

CPU Server
- CPU
- Local Memory (cache)
- Swap

Memory Server

60 ns

RDMA over InfiniBand
Process Execution Model

Process

CPU Server
- CPU
- Local Memory (cache)
- Swap

Memory Server
- RDMA over InfiniBand

60 ns

~ 10 μs
150 Times Slower
Limitations of Previous Work

- Previous works focus on semantics-agnostic optimizations
  - Reduce or hide the remote access latency
  - Prefetch data to reduce the remote access frequency

- Cloud applications – written in managed languages
  - Heap space: Reserved virtual space from OS
  - Garbage Collection (GC): Automatic memory management
  - Object-oriented data structures

*Managed language applications often have poorer locality than native programs*
Poor Data Locality

Object-oriented data structures

- Random memory access – poor locality, hard to predict access pattern
- Pointer-chasing memory access – latency sensitive
Resources Racing

GC slows down the applications

- The concurrent GC threads race resources, e.g., local cache and InfiniBand bandwidth, with the application threads
# Slowdown of Spark Applications

- Both applications and GC slow down significantly on a disaggregated cluster.
- GC is on the critical path:
  - GC increases the pause time
  - GC slows down the application’s execution

<table>
<thead>
<tr>
<th>Cache Ratio</th>
<th>Apps</th>
<th>GC</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Swap</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>50%</td>
<td>2.0X</td>
<td>24.7X</td>
<td>8.4X</td>
</tr>
<tr>
<td>25%</td>
<td>5.3X</td>
<td>53.5X</td>
<td>18.9X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Ratio</th>
<th>Apps</th>
<th>GC</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Swap</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>50%</td>
<td>1.2X</td>
<td>2.0X</td>
<td>1.4X</td>
</tr>
<tr>
<td>25%</td>
<td>2.0X</td>
<td>3.3X</td>
<td>2.3X</td>
</tr>
</tbody>
</table>

Spark GraphX TriangleCounting

Spark MLlib KMeans
Major Insights

- Offload part of GC to memory servers where the data is located
  - Good fit for weak compute on memory servers
  - Near memory computing for high throughput
  - GC can run *concurrently* and *continuously*

- Utilize GC to adjust the data layout for applications

*Semeru – A Disaggregated Managed Runtime*
Challenges

- #1 What memory abstraction to provide?
  - Universal Java Heap (UJH)

- #2 What to offload?

- #3 How to efficiently swap data?
Universal Java Heap (UJH)

- A normal JVM runs on the CPU server, accessing the whole Java heap
Universal Java Heap (UJH)

- A normal JVM runs on the CPU server, accessing the whole Java heap.

- A Lightweight-JVM (LJVM) runs on each memory server, accessing its assigned Java heap range.
Universal Java Heap (UJH)

- A normal JVM runs on the CPU server, accessing the whole Java heap

- A Lightweight-JVM (LJVM) runs on each memory server, accessing its assigned Java heap range

- Each object has the **same virtual address** on both the CPU server and memory servers
CPU Server Cache Management

- Write-back policy
  - Objects are allocated in CPU server memory (local cache)
  - Only *dirty* pages are evicted to memory servers
  - When a page is freed by GC, it returns to the *Init* state

State Machine of Virtual Page

- *Init*
- *Allocate*
- *Cached-Clean*
- *Cached-Dirty*
- *Evicted*
- *Swap in*
- *Swap out*
- *Free (unmap)*
Challenges

- Universal Java Heap (UJH)
- #2 What to offload?
  - Memory Server Concurrent Tracing (MSCT)
- #3 How to efficiently swap data?
Disaggregated GC Overview

- Offload *tracing* to memory servers
  - Memory Server Concurrent Tracing (MSCT)
Disaggregated GC Overview

- Offload *tracing* to memory servers
  - Memory Server Concurrent Tracing (MSCT)

- Keep a GC phase on CPU server for *memory reclamation*
  - CPU Server Stop-the-world Collector (CSSC)
Disaggregated GC Overview

- Offload *tracing* to memory servers
  - Memory Server Concurrent Tracing (MSCT)

- Keep a GC phase on CPU server for *memory reclamation*
  - CPU Server Stop-the-world Collector (CSSC)

![Diagram showing coordination between CPU Server, Memory Servers, MSCT, and CSSC](image)
MSCT – Regions to be Traced

Memory Server, LJVM#1

Heap Slice for LJVM#1

Evicted Region#1  Evicted Region#2  Evicted Region#3

Page *cached* in CPU server

Page *evicted* to memory server
MSCT – Regions to be Traced

Memory Server, LJVM#1

Heap Slice for LJVM#1

Evicted Region#1

Evicted Region#2

Evicted Region#3

Page \textit{cached} in CPU server

Page \textit{evicted} to memory server
MSCT – Regions to be Traced

Heap Slice for LJVM#1

Evicted Region#1
Evicted Region#2
Evicted Region#3

Memory Server, LJVM#1

Page \textit{cached} in CPU server
Page \textit{evicted} to memory server

Tracing Order: Region#2 \rightarrow Region#3

\textbf{Generation Hypothesis:}
Newly allocated objects are more likely to die.
MSCT – Tracing Roots

Tracing roots for each region

- References from stack variables
- References from other regions

**Diagram:**

- CPU Server, JVM
  - Stack variables
  - Other regions
- Memory Server, LJVM#1
  - Region#2
  - Object #N
  - Object #M
Tracing roots for each region

- References from stack variables
- References from other regions
CPU Server Stop-The-World Collection (CSSC)

- CPU server GC is the main collection phase
  - Trace the cached regions on the CPU server
  - Coordinate CPU server and memory servers for space compaction
  - Adjust the data layout for applications
Semeru Design Outline

- Universal Java Heap (UJH)
- Disaggregated GC
  - Memory Server Concurrent Tracing (MSCT)
  - CPU Server Stop-The-World Collection (CSSC)
- #3 How to design the swap system?
Swap System Overview

CPU Server

Runtime

Paging

RDMA

Data Path

Control Path

Provide kernel information to runtime

Memory Server

Memory Server

Memory Server

Memory Server
Swap System Overview

- **Runtime**
- **Paging**
- **RDMA**
- **Data Path**
- **Control Path**
- **Scatter/Gather**

CPU Server

- Provide kernel information to runtime

Memory Servers

- Memory Server
- Memory Server
Experiment Setup

- 2 CPUs per server
  Intel Xeon E5-2640 v3 @2.60GHz, 8 cores
- InfiniBand
  ConnectX®-3, MT4099, 40Gb/s
- CPU Local Memory
  DDR4-1866, Limit capacity by CGroup

- 3 memory servers per application
- 2 cores per server
  Intel Xeon E5-2640 v3
  Limit number of cores
  Fix CPU freq to 1.2GHz / 2.6GHz
Overall Performance

- **Workloads**
  - 5 Spark applications
  - 3 Flink applications

- **Datasets**
  - Wikipedia
  - KDD

- **Configurations**
  - Baseline: No swap
  - NVMe-oF
  - RAMDisk

### 50% Cache

<table>
<thead>
<tr>
<th></th>
<th>Apps</th>
<th>GC</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1-NVMe-oF</td>
<td>2.00X</td>
<td>4.44X</td>
<td>2.24X</td>
</tr>
<tr>
<td>G1-RAMDisk</td>
<td>1.82X</td>
<td>2.79X</td>
<td>1.87X</td>
</tr>
<tr>
<td>Semeru</td>
<td>1.06X</td>
<td>1.42X</td>
<td><strong>1.08X</strong></td>
</tr>
</tbody>
</table>

### 25% Cache

<table>
<thead>
<tr>
<th></th>
<th>Apps</th>
<th>GC</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1-NVMe-oF</td>
<td>3.85X</td>
<td>14.13X</td>
<td>4.58X</td>
</tr>
<tr>
<td>G1-RAMDisk</td>
<td>3.16X</td>
<td>4.59X</td>
<td>3.23X</td>
</tr>
<tr>
<td>Semeru</td>
<td>1.22X</td>
<td>2.67X</td>
<td><strong>1.32X</strong></td>
</tr>
</tbody>
</table>
Memory-Server Tracing Performance

- **GC Improvement**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Tracing Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput (MB/s)</td>
</tr>
<tr>
<td></td>
<td>Core Utilization</td>
</tr>
<tr>
<td><strong>(Memory Server)</strong></td>
<td></td>
</tr>
<tr>
<td>Single core, 1.2 GHz</td>
<td>418.3</td>
</tr>
<tr>
<td></td>
<td>29.0%</td>
</tr>
<tr>
<td>Single core, 2.6 GHz</td>
<td>922.2</td>
</tr>
<tr>
<td></td>
<td>12.4%</td>
</tr>
<tr>
<td><strong>(CPU Server)</strong></td>
<td></td>
</tr>
<tr>
<td>Single core, 2.6 GHz</td>
<td>93.9</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

- **Offload tracing to memory servers increases throughput** 8.8X

- **Weak core is powerful enough to do continuous tracing on memory servers**
Conclusions

- Semeru achieves superior efficiency on the disaggregated cluster via
  - A co-design of the runtime and swap system
  - Careful coordination of different GC tasks

- Disaggregation performance could benefit much more from a redesigned runtime than semantics-agnostic optimizations
Q&A

Thanks

wangchenxi@cs.ucla.edu