Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications

Rodrigo Bruno, Paulo Ferreira: INESC-ID / Instituto Superior Técnico, University of Lisbon
Ruslan Synytsky, Tetiana Fydorenchyk: Jelastic
Jia Rao: The University of Texas at Arlington
Hang Huang, Song Wu: Huazhong University of Science and Technology

ISMM 18 June @ Philadelphia, USA
Unused Resources in the Cloud

- Real data from Jelastic cloud provider between 2014 and 2017
- More than 25 TBs of unused RAM in 2017
- Most cloud providers charge for reserved resources
  - Users are paying for resources that are not used!
- Cloud users are forced to overprovision
  - Memory requirements not known
  - Dynamic workloads
Unused Resources in the Cloud

- Real data from Jelastic cloud provider between 2014 and 2017
- More than 25 TBs of unused RAM in 2017
- Most cloud providers charge for reserved resources
  - Users are paying for resources that are not used!
- Cloud users are forced to overprovision
  - Memory requirements not known
  - Dynamic workloads
“Pay-as-you-Go” vs “Pay-as-you-Use”

Pay-as-you-Go
Pay for statically-sized instances

Pay-as-you-use
Pay for used resources
“Pay-as-you-Use” for JVM Applications

- Proof-of-concept experiment, 1 instance, one task processed after startup and then idle.
“Pay-as-you-Use” for JVM Applications

- Proof-of-concept experiment, 1 instance, one task processed after startup and then idle

Problem 1: The JVM does not release RAM even if it is not being used (committed)!
Problem 2: Applications cannot scale beyond Max Heap limit!
“Pay-as-you-Use” for JVM Applications

- Proof-of-concept experiment, 1 instance, one task processed after startup and then idle.

Dynamic Vertical Scalability is a requirement for taking advantage of “Pay-as-you-Use”

Problem 1: The JVM does not release RAM even if it is not being used (committed)!
Problem 2: Applications cannot scale beyond Max Heap limit!
Goal: vertical memory scalability for JVM apps

Improve the way the JVM fits in the virtualization stack (system-VMs and containers).
Goal: vertical memory scalability for JVM apps

Improve the way the JVM fits in the virtualization stack (system-VMs and containers).

Goal 1: give memory back to the host engine when it is not being used
Goal: vertical memory scalability for JVM apps

Improve the way the JVM fits in the virtualization stack (system-VMs and containers).

**Goal 1:** give memory back to the host engine when it is not being used

**Goal 2:** allow the JVM to grow its memory beyond the limit defined at launch time
Goal: vertical memory scalability for JVM apps

Improve the way the JVM fits in the virtualization stack (system-VMs and containers).

Goal 1: give memory back to the host engine when it is not being used

Goal 2: allow the JVM to grow its memory beyond the limit defined at launch time

Goal 3: negligible negative throughput or memory footprint impact
Goal: vertical memory scalability for JVM apps

Improve the way the JVM fits in the virtualization stack (system-VMs and containers).

**Goal 1:** give memory back to the host engine when it is not being used

**Goal 2:** allow the JVM to grow its memory beyond the limit defined at launch time

**Goal 3:** negligible negative throughput or memory footprint impact

**Goal 4:** negligible pause-time for scaling memory
Goal: vertical memory scalability for JVM apps

Improve the way the JVM fits in the virtualization stack (system-VMs and containers).

Goal 1: give memory back to the host engine when it is not being used

Goal 2: allow the JVM to grow its memory beyond the limit defined at launch time

Goal 3: negligible negative throughput or memory footprint impact

Goal 4: negligible pause-time for scaling memory

Goal 5: no changes to the host engine/OS
Can’t we use JVM tuning and/or cloud management tools?

No...
Can’t we use JVM tuning and/or cloud management tools?

No...

**Reason 1**: it is not possible to force the JVM to release memory from the outside (even a Full GC won’t do it for some collectors such as PS).
Can’t we use JVM tuning and/or cloud management tools?

No...

**Reason 1:** it is not possible to force the JVM to release memory from the outside (even a Full GC won’t do it for some collectors such as PS).

**Reason 2:** Horizontal scaling does not work if suddenly you need more memory than what you have in a single instance. It also requires more infrastructure and sophisticated algorithms to manage multiple instances;
Can’t we use JVM tuning and/or cloud management tools?

No...

**Reason 1:** it is not possible to force the JVM to release memory from the outside (even a Full GC won’t do it for some collectors such as PS).

**Reason 2:** Horizontal scaling does not work if suddenly you need more memory than what you have in a single instance. It also requires more infrastructure and sophisticated algorithms to manage multiple instances;

**Reason 3:** Setting a very high memory limit for the JVM solves the lack of memory problem but worsens reason 1;
Can’t we use JVM tuning and/or cloud management tools?

No...

**Reason 1:** it is not possible to force the JVM to release memory from the outside (even a Full GC won’t do it for some collectors such as PS).

**Reason 2:** Horizontal scaling does not work if suddenly you need more memory than what you have in a single instance. It also requires more infrastructure and sophisticated algorithms to manage multiple instances;

**Reason 3:** Setting a very high memory limit for the JVM solves the lack of memory problem but worsens reason 1;

**Reason 4:** Rebooting the JVM to adjust the memory limit takes a long time leading to service unavailability, which is prohibitive for many applications.
Dynamic Vertical Memory Scaling

2-step solution:
Dynamic Vertical Memory Scaling

2-step solution:

Step 1:
1. dynamically increase or decrease the JVM memory limit (i.e. amount of memory available to the application)
2. allow the cloud user to change this limit (this can also be done programmatically)
Dynamic Vertical Memory Scaling

2-step solution:

**Step 1:**
1. dynamically increase or decrease the JVM memory limit (i.e. amount of memory available to the application)
2. allow the cloud user to change this limit (this can also be done programmatically)

**Step 2:**
1. JVM heap sizing strategy that sizes the heap according to the application’s used memory
2. Even if no GC is triggered, the heap size should be checked
Step 1: Current Max Heap Size

- We introduce a new JVM variable: CurrentMaxHeapSize
  - can be set at launch time or at runtime, no need to guess the heap size beforehand
  - once set, the heap cannot grow beyond its value
- Max heap size can be set to a conservatively high value (only affects reserved memory not committed memory)

**Algorithm 1 Set Current Maximum Heap Size**

1. procedure SET_CURRENT_MAX_MEMORY(new_max)
2. committed_mem ← CommittedMemory
3. reserved_mem ← MaxMemory
4. if new_max > reserved_mem then
   return failure
5. if new_max < committed_mem then
   trigger GC
   committed_mem ← CommittedMemory
   if new_max < committed_mem then
     return failure
7. CurrentMaxMemory ← new_max
8. return success
Step 2: Periodic Heap Resizing Checks

- **if...**
  - unused heap memory is large (line 6)
  - last GC was a long ago (line 8)
- **do...** heap resize

- MaxOverCommittedMem and MinTimeBetweenGCs can be set at launch time or at runtime

- We do not implement a new heap sizing algorithm, the JVM already has advanced ergonomic policies
  - we “just” determine when to run it

---

**Algorithm 2 Should Resize Heap Check**

```
1: procedure SHOULD_RESIZE_HEAP
2:   commit_mem ← CommittedMemory
3:   used_mem ← UsedMemory
4:   time_since_gc ← TimeSinceLastGC
5:   over_commit ← commit_mem - used_mem
6:   if over_commit < MaxOverCommittedMem then
7:     return false
8:   if time_since_gc < MinTimeBetweenGCs then
9:     return false
10: return true
```
Execution Memory Usage log

- **Xmx**: CurrentMaxHeapSize
- **MaxOverCommitted**
- **Committed Heap**
- **Used Heap**
- **MinTimeBetweenGCs**
- **MinCommitted**
Implementation

- Solution implemented in the OpenJDK 9 HotSpot JVM

- CurrentMaxHeapSize, MaxOverCommittedMem, and MinTimeBetweenGCs are runtime variables that can be set at JVM launch time or at runtime;

- Periodic heap sizing checks are integrated in the VM control thread loop (executed nearly every second);

- JVM allocation path and heap growing respects CurrentMaxHeapSize

- Two collectors supported:
  - Garbage First, most advanced GC, the new by-default
  - Parallel Scavenge, widely used parallel collector

- We reuse the ergonomics code already added into the GC to implement the heap sizing operation
Evaluation

- Compare: G1 vs VG1 (vertical G1); PS vs VPS (vertical PS)
- Benchmarks: DaCapo 9.12 and Tomcat web server (real workload)
- Host node: Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz, 32GBs DDR4 of RAM, Linux 4.9
- Host engine: Docker 17.12
- Each JVM runs in an isolated container
# DaCapo 9.12 Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Iterations</th>
<th>CMaxMem</th>
<th>MaxOCMem</th>
<th>MinTimeGCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>AVR microcontrollers</td>
<td>5</td>
<td>32 MB</td>
<td>16 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>fop</td>
<td>XSL-FO to PDF</td>
<td>200</td>
<td>512 MB</td>
<td>32 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>h2</td>
<td>JDBCbench-like in-memory</td>
<td>5</td>
<td>1024 MB</td>
<td>256 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>jython</td>
<td>interprets the pybench</td>
<td>5</td>
<td>128 MB</td>
<td>32 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>luindex</td>
<td>lucene indexing</td>
<td>100</td>
<td>256 MB</td>
<td>32 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>pmd</td>
<td>searches code problems</td>
<td>10</td>
<td>256 MB</td>
<td>32 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>sunflow</td>
<td>ray tracing</td>
<td>5</td>
<td>128 MB</td>
<td>16 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>tradebeans</td>
<td>daytrader benchmark</td>
<td>5</td>
<td>512 MB</td>
<td>128 MB</td>
<td>10 sec</td>
</tr>
<tr>
<td>xalan</td>
<td>XML to HTML</td>
<td>5</td>
<td>64 MB</td>
<td>16 MB</td>
<td>10 sec</td>
</tr>
</tbody>
</table>
Memory Scalability - JVM Heap Size (MB)
Memory Scalability - JVM Heap Size (MB)

![Graph showing memory scalability and JVM heap size](image)

- Avg Heap Size (MB)
- Lower is Better

High allocation rate lead to higher improvements
Memory Scalability - Container Mem Usage (MB)
Memory Scalability - Container Mem Usage (MB)

High allocation rate lead to higher improvements

Lower is Better
Execution Time (ms)

![Execution Time Graph](image)

- **G1**
- **VG1**
- **PS**
- **VPS**

**Lower is Better**
Execution Time (ms)

Minor Throughput Overheads for most benchmarks

Lower is Better
Throughput vs Memory Tradeoff

![Graph showing throughput vs memory tradeoff with different benchmarks like avrora, jython, sunflow, tradebeans, xalan, luindex, fop, h2, and pmd.]
Different benchmarks have different memory throughput tradeoffs!
High Max Heap Limit Memory Overhead (h2 benchmark)

![Graph showing container memory usage (MB) vs max heap limit multiplier (1x = 1GB). VG1 and VPS lines are compared. Lower is better.]
High Max Heap Limit Memory Overhead (h2 benchmark)

Container Mem Usage (MB)

![Graph showing memory usage across different heap limit multipliers]

Lower is Better
High Max Heap Limit Memory Overhead (h2 benchmark)

High Xmx is compensated by periodic heap resizing!

Lower is Better
Real-world Scenario Experiment

- Tomcat web server with 4-16GBs (based on real Jelastic clients’ workloads)
  - utilized mostly during the day; at night (8 hours) the server is mostly idle
  - user sessions (which occupy most of the memory) timeout after 10 min
  - monthly cost estimation using Amazon EC2 (Ohio datacenter)
    - assuming one could change the instance resources on the fly
Real-world Scenario Experiment (mem utilization)

![Graph showing Container Mem Usage (MB) vs Time (hours)](image)

- **Lower is Better**
Real-world Scenario Experiment (mem utilization)

Container Mem Usage (MB)

Resources can be saved during idle time!

Lower is Better
Real-world Scenario Experiment (cost)

<table>
<thead>
<tr>
<th>Approach</th>
<th>During Day</th>
<th>During Night</th>
<th>Total</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>4GB-JVM</td>
<td>23.01$</td>
<td>11.53$</td>
<td>34.00$</td>
<td></td>
</tr>
<tr>
<td>4GB-VJVM</td>
<td>1.44$</td>
<td>24.44$</td>
<td>29.40%</td>
<td></td>
</tr>
<tr>
<td>8GB-JVM</td>
<td>46.03$</td>
<td>23.01$</td>
<td>69.04$</td>
<td>31.00%</td>
</tr>
<tr>
<td>8GB-VJVM</td>
<td>1.44$</td>
<td>47.47$</td>
<td>48.91%</td>
<td></td>
</tr>
<tr>
<td>16GB-JVM</td>
<td>92.06$</td>
<td>46.03$</td>
<td>138.00$</td>
<td>32.60%</td>
</tr>
<tr>
<td>16GB-VJVM</td>
<td>1.44$</td>
<td>93.50$</td>
<td>94.94%</td>
<td></td>
</tr>
<tr>
<td>32GB-JVM</td>
<td>184.12$</td>
<td>92.06$</td>
<td>276.00$</td>
<td>33.00%</td>
</tr>
<tr>
<td>32GB-VJVM</td>
<td>1.44$</td>
<td>185.00$</td>
<td>186.44$</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- Vertical Memory Scalability is an enabler for the “Pay-as-you-Use” model
- It can be implemented in the JVM with
  - negligible throughput cost
  - very promising footprint reductions
- Implementation can be easily ported to other GCs
- JEPs:
  - [http://openjdk.java.net/jeps/8204089](http://openjdk.java.net/jeps/8204089)
  - [http://openjdk.java.net/jeps/8204088](http://openjdk.java.net/jeps/8204088)
Conclusion

- Vertical Memory Scalability is an enabler for the “Pay-as-you-Use” model
- It can be implemented in the JVM with
  - negligible throughput cost
  - very promising footprint reductions
- Implementation can be easily ported to other GCs
- Code is working in production at Jelastic
- JEPs:
  - [http://openjdk.java.net/jeps/8204089](http://openjdk.java.net/jeps/8204089)
  - [http://openjdk.java.net/jeps/8204088](http://openjdk.java.net/jeps/8204088)

Thank you for your time!

Questions?

Rodrigo Bruno

email: rodrigo.bruno@tecnico.ulisboa.pt
webpage: [www.gsd.inesc-id.pt/~rbruno](http://www.gsd.inesc-id.pt/~rbruno)
github: [github.com/rodrigo-bruno](https://github.com/rodrigo-bruno)