ZGC Concurrent Class Unloading

Another safepoint operation bites the dust

Erik Österlund
Garbage Collection Engineer
Java Platform Group, Oracle
February 04, 2019
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Agenda

1. What is ZGC? What is class unloading?
2. Overview of phases
3. Concurrent code and metadata unloading
4. Evaluation
5. Future plans
What is ZGC?
New Concurrent GC in JDK 11

(Experimental feature, Linux/x86_64 only)
## ZGC Goals

<table>
<thead>
<tr>
<th>TB</th>
<th>Multi-terabyte heaps</th>
<th>10\text{ms}</th>
<th>Max GC pause time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Easy to tune</td>
<td>15%</td>
<td>Max application throughput reduction</td>
</tr>
</tbody>
</table>
# What is a concurrent GC

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>Parallel</th>
<th>CMS</th>
<th>G1</th>
<th>ZGC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Marking</strong></td>
<td>-</td>
<td>-</td>
<td>✔*</td>
<td>✔*</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Relocation/Compaction</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Reference Processing</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Relocation Set Selection</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>StringTable Cleaning</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>JNI WeakRef Cleaning</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>JNI GlobalRefs Scanning</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Class Unloading</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Thread Stack Scanning</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*) Old Gen strong references Only
Concurrent Class Unloading
Released in **JDK 12** for **ZGC**
Traditional Class Unloading

• Step 1: Marking (**concurrent**)
  – Mark metadata (classes, CLDs) when marking objects

• Step 2: Reference processing (**STW**)
  – Need to know what is reachable from finalizers before class unloading

• Step 3: Unloading (**STW**)
  – Unload code cache
  – Unload metadata
ZGC Concurrent Class Unloading

• Step 1: Marking (concurrent)
  – Mark metadata (classes, CLDs) when marking objects
  – Mark both strong and final reachable graphs

• Step 2: Reference processing (concurrent)
  – Already know what is reachable from finalizers before class unloading

• Step 3: Unloading (concurrent)
  – Unload code cache
  – Unload metadata
ZGC Phases

Pause Mark Start → Concurrent Mark/Remap → Concurrent Reference Processing → Concurrent Class Unloading → Pause Relocate Start → Concurrent Relocate → GC Cycle

Pause Mark End → Concurrent Prepare for Reloc.
Step 1: Marking
Marking overview

• Color heap object pointers with appropriate marked color
  – Special bit pattern for edges reachable from finalizers only
  – Mutator load barriers upgrade them to strongly reachable when loaded

• Mark metadata objects similarly
  – Mark metadata reachable from objects, with strong/final reachability
Step 2: Reference Processing
Reference processing overview

- WeakReferences cleared if referent not strongly reachable
- PhantomReferences cleared if referent not reachable
- "Weak" VM datastructures have "phantom" strength
  - Classes die if not reachable (including from finalizers)
- Each access on weak/phantom is annotated using my Access API
- A class is dead if a phantom load of its holder returns NULL
Step 3: Unloading
# Stale Datastructures after Reference Processing

<table>
<thead>
<tr>
<th>Subklass/sibling/implementor lists</th>
<th>Package table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method data objects</td>
<td>Symbol table</td>
</tr>
<tr>
<td>Instance class dependency context</td>
<td>Resolved method table</td>
</tr>
<tr>
<td>jli.CallSite dependency context</td>
<td>Loader constraint table</td>
</tr>
<tr>
<td>Class loader data graph</td>
<td>Resolution error table</td>
</tr>
<tr>
<td>Protection domain cache table</td>
<td>Metaspace</td>
</tr>
<tr>
<td>Module table</td>
<td>Code cache (JIT compiled code)</td>
</tr>
<tr>
<td>String table</td>
<td>Inline caches</td>
</tr>
</tbody>
</table>
Basically everything is a huge mess...
...and we just continue running anyway
ZGC Unloading Overview

Pause Mark End

Reference Processing

Class Unloading

Unlink

Purge

Handshake
ZGC Unloading Overview

Pause Mark End

Reference Processing

Class Unloading

Unlink

Metadata

Handshake

Purge

Metadata
ZGC Unloading Overview

Pause Mark End

Reference Processing

Class Unloading

Unlink

Metadata

Code

Purge

Metadata

Code
Concurrent Code Unloading
Code cache

• Colored pointers into Java heap
  – Misaligned immediate values
  – Which color should native compiled object references have?
  – Need a way to paint native compiled object references

• Inline caches (CompiledIC) pointing at now dead native methods (nmethods), because of dead object references
  – Running any such code yields crashes
  – Need a way of preventing calls to dead native methods
NMethod Entry Barriers

• Arm all nmethods not on stack in GC pause
  – Change global epoch value, caught with cmp; je; at verified entry

• Trap calls to armed nmethods
  – NMethods are "good" or "bad" based on object pointer liveness

• When entering good nmethods
  – Fix up object pointers (oops)
  – Disarm barrier by patching cmp immediate value

• When entering bad nmethods
  – Re-resolve the call
Example: Calling inline cache to dead nmethod
Java Heap

Java Object

Code Heap

nmethod A

oop

oop

CompiledIC

C2I adapter

Metaspace

Method
Example: Calling inline cache to stale but live nmethod
Concurrent Code Unloading

• Unlink
  – Clean stale inline caches (patch machine code that Java threads run)
  – Fixup object references (patching more machine code)
  – Disarm entry barriers (yet some more machine code patching)
  – Unlink nmethods from dependency contexts (lock-free unlinking)
  – Unlink exception caches (more lock-free unlinking)

• Global rendezvous handshake

• Purge
  – Purge unloading nmethods with make_unloaded()
  – Sweeper subsequently frees up memory in code cache
Concurrent Metadata Unloading
Structure

• Unlink
  – Expose logically already unlinked view of data to mutators
  – Subclass/sibling/implementor lists (lock-free)
  – Method data object (lock-free and per-MDO lock)
  – Protection domain cache (lock)
  – Class loader data graph (lock)
  – StringTable and SymbolTable (crazy concurrent)

• Rendezvous handshake

• Purge
  – Delete Klass, Method, CLD, handles, table entries, etc.
Evaluation
SPECjbb®2015 – Score

Mode: Composite
Heap Size: 128G
OS: Oracle Linux 7.4
HW: Intel Xeon E5-2690 2.9GHz
2 sockets, 16 cores (32 hw-threads)

SPECjbb®2015 is a registered trademark of the Standard Performance Evaluation Corporation (spec.org). The actual results are not represented as compliant because the SUT may not meet SPEC's requirements for general availability.
SPECjbb® 2015 – Score

(Higher is better)

Mode: Composite
Heap Size: 128G
OS: Oracle Linux 7.4
HW: Intel Xeon E5-2690 2.9GHz
2 sockets, 16 cores (32 hw-threads)

SPECjbb® 2015 is a registered trademark of the Standard Performance Evaluation Corporation (spec.org). The actual results are not represented as compliant because the SUT may not meet SPEC's requirements for general availability.
SPECjbb®2015 – Pause Times
SPECjbb®2015 – Pause Times

![Bar chart showing GC Pause Times for different Java environments. The chart includes bars for ZGC JDK11, ZGC JDK12, Parallel, and G1, with a log scale on the y-axis. The chart compares average and maximum pause times.]
SPECjbb®2015 – Pause Times
Future Plans
Future Plans

• Short-term
  – Turn ZGC into a product feature
Future Plans

• Short-term
  – Turn ZGC into a product feature

Remove experimental status
Future Plans

• Short-term
  – Turn ZGC into a product feature
Future Plans

• **Short-term**
  – Turn ZGC into a product feature

• **Long-term**
  – Generational
  – Sub-millisecond max pause times
  – Additional platform support
  – Graal JIT support
Future Plans

• **Short-term**
  – Turn ZGC into a product feature

• **Long-term**
  – Generational
  – Sub-millisecond max pause times
  – Additional platform support
  – Graal JIT support

**Generational**
• Withstand higher allocation rates
• Lower heap overhead
• Lower CPU usage
Future Plans

• **Short-term**
  – Turn ZGC into a product feature

• **Long-term**
  – Generational
  – Sub-millisecond max pause times
  – Additional platform support
  – Graal JIT support

**Sub-millisecond max pause times**

• Within reach
• Reduce root set size
• Time-to-Safepoint, etc
Future Plans

• **Short-term**
  – Turn ZGC into a product feature

• **Long-term**
  – Generational
  – Sub-millisecond max pause times
  – Additional platform support
  – Graal JIT support

Additional platform support
• macOS?
• Windows?
• Sparc?
• Aarch64?
Future Plans

• **Short-term**
  – Turn ZGC into a product feature

• **Long-term**
  – Generational
  – Sub-millisecond max pause times
  – Additional platform support
  – Graal JIT support
Get Involved!
ZGC Project
Follow, Participate, Give Feedback

zgc-dev@openjdk.java.net

http://wiki.openjdk.java.net/display/zgc/Main
ZGC Project
Source Code

OpenJDK

http://hg.openjdk.java.net/jdk/jdk

http://hg.openjdk.java.net/zgc/zgc

Latest Stable

Bleeding Edge
Thanks!
Questions?