Oracle Code One 2019

G1 and ZGC: A Look into the Progress of Garbage Collection in Java

[DEV4459]

Mikael Vidstedt, Director, Java Virtual Machine
Paul Su, Director, JVM Garbage Collection

Oracle
September 16, 2019
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, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle’s future plans, expectations, beliefs, intentions and prospects are “forward-looking statements” and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle’s Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading “Risk Factors.” These filings are available on the SEC’s website or on Oracle’s website at http://www.oracle.com/investor. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.
Agenda

• Introduction to garbage collection
• GC tradeoffs
• G1
• ZGC
Introduction to Garbage Collection
Introduction to Garbage Collection
Introduction to Garbage Collection
Introduction to Garbage Collection
Introduction to Garbage Collection
Introduction to Garbage Collection
Introduction to Garbage Collection
Collectors in the JVM

<table>
<thead>
<tr>
<th>GC</th>
<th>Optimized For</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Memory Footprint</td>
</tr>
<tr>
<td>Parallel</td>
<td>Throughput</td>
</tr>
<tr>
<td>G1</td>
<td>Throughput/Latency Balance</td>
</tr>
<tr>
<td>CMS</td>
<td>Latency</td>
</tr>
<tr>
<td>ZGC</td>
<td>Scalability/Low Latency</td>
</tr>
</tbody>
</table>
The G1 Garbage Collector

The default garbage collector since JDK 9
- First introduced in 6u14
- Supported since 7u4

The goal: throughput and low latency

The default pause target for G1 is 200 milliseconds
- Higher pause goal → more throughput, higher latency
- Lower pause goal → less throughput, lower latency
G1 - Generational region-based memory management

- The heap is split into multiple regions
- Region size depends on heap size, e.g. 2 MB for 4 GB heap
G1 - Generational region-based memory management

- New objects are allocated into eden (E) regions
G1 - Generational region-based memory management

• A young collection happens after a number of eden regions have been allocated
G1 - Generational region-based memory management

- Young collections compactly copy live objects in eden regions to survivor regions (S)
G1 - Generational region-based memory management

- Objects will then continue to be allocated in eden regions
G1 - Generational region-based memory management

- If objects survive multiple young collections, then they are compactly copied into an old region (O)
G1 - Generational region-based memory management

- After a while the heap fills up with eden, survivor and old regions
G1 - Generational region-based memory management

- All live objects in old regions are then marked *concurrently*
- The Java application is *not* stopped
G1 - Generational region-based memory management

- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.
G1 - Generational region-based memory management

- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.
G1 - Generational region-based memory management

- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | O |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ O |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   | O |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ O |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   | O |   | S |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ O | O |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   | O |   | O |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ O | O |   |   |   |
|   |   |   |   |   |   |   |   |   |   | O |   | O |   | O |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ O | O |   |   |   |
|   |   |   |   |   |   |   |   |   | O |   | O |   | O |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ O | O |   |   |   |

Heap
```
G1 - Generational region-based memory management

- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.

Heap
G1 - Generational region-based memory management

• When no more old regions are suitable for collection, then G1 will resume doing young collections.
G1 enhancements since JDK 8

13 GC-related JEPs since JDK 8
• 5 related to G1
G1 enhancements since JDK 8

13 GC-related JEPs since JDK 8
  • 5 related to G1
~1450 GC enhancements
  • ~699 related to G1

The JEPs represent only a small portion of the work going into the GC area.
G1 enhancements between JDK 8 and JDK 9

- Latency
- Memory
- Throughput
G1 enhancements between JDK 8 and JDK 9

**Latency**
- Fast Evacuation
- Failure Handling

**Ergonomic**
- Thread Tuning

**Memory**
- Parallelize GC Phases
- Lazy RemSet Initialization
- Heap Resizing Ergonomics
- RemSet Space Reductions

**Throughput**
- Parallelize GC Phases
- Lazy Thread Initialization
- Lazy RemSet Initialization
- Archive regions
- CDS support
- Container Awareness
- Lazy Thread Initialization

**Superscripts indicate JDK versions containing significant work in that area**
Adaptive Mark Start/Initiating Heap Occupancy Percentage (IHOP)
G1 enhancements between JDK 9 and JDK 10

**Latency**
- Fast Evacuation Failure Handling
- Optimize Evacuation
- Ergonomic Thread Tuning
- Parallel Pretouch
- Elastic TLAB

**Memory**
- Parallelize GC Phases
- Lazy Thread Initialization
- Lazy RemSet Initialization
- Heap Resizing Ergonomics
- RemSet Space Reductions
- Container Awareness

**Throughput**
- Parallel Full GC
- Improved Refinement
- Adaptive Mark Start

Superscripts indicate JDK versions containing significant work in that area.
Parallel Full GC

- Now uses the same number of parallel threads as young and mixed GCs
G1 enhancements between JDK 10 and JDK 11

**Latency**
- Reference Precleaning\(^{11}\)
- Fast Evacuation Failure Handling\(^{9}\)
- Parallel Reference Processing\(^{11}\)
- Rebuild RemSets On the Fly\(^{11}\)
- Ergonomic Thread Tuning\(^{9,11}\)
- Elastic TLAB\(^{9}\)
- Parallel Full GC\(^{10}\)

**Throughput**
- Parallelize GC Phases\(^{9+}\)
- Lazy RemSet Initialization\(^{9}\)
- Lazy Thread Initialization\(^{9}\)
- Memory
  - RemSet Space Reductions\(^{9,11}\)
  - Marking Space Reductions\(^{11}\)
  - Container Awareness\(^{9,11}\)
- Marking Scalability\(^{9}\)
- Improved Refinement\(^{9}\)
- Adaptive Mark Start\(^{9}\)

**Memory**
- Archive regions CDS support\(^{9}\)
- Rebuild
- RemSets
- On the Fly\(^{11}\)
- Parallel Pretouch\(^{9}\)
- Heap Resizing Ergonomics\(^{9}\)

Superscripts indicate JDK versions containing significant work in that area.
Rebuild Remembered Sets on the Fly

• Remembered sets are data structures that track references into a region
• Can occupy a significant amount of memory i.e. 20% of the total heap
  • Remembered sets for old regions are particularly large
• G1 maintains remembered sets for all regions
  • But only need remembered sets for old regions during Mixed GCs
Rebuild Remembered Sets on the Fly

• Dynamically rebuild remembered sets after the concurrent mark phase
• Only build remembered sets for regions in the collection set
• Improves both throughput and latency
G1 enhancements between JDK 11 and JDK 12

Latency
- Reference Precleaning\textsuperscript{11}
- Fast Evacuation Failure Handling\textsuperscript{9}
- Parallel Reference Processing\textsuperscript{11}
- Parallelize GC Phases\textsuperscript{9+}
- Lazy Thread Initialization\textsuperscript{9}
- Lazy RemSet Initialization\textsuperscript{9}
- Uncommit At Remark\textsuperscript{12}
- Memory
- Old Gen On NVDIMM\textsuperscript{12}
- Archive regions CDS support\textsuperscript{9}
- Container Awareness\textsuperscript{9,11}
- Throughput
- Adaptive Mark Start\textsuperscript{9}
- Improved Refinement\textsuperscript{9}
- Elongated TLAB\textsuperscript{9}
- Flexible Full GC\textsuperscript{10}
- Rebuild RemSets On the Fly\textsuperscript{11}
- Ergonomic Thread Tuning\textsuperscript{9,11}
- Parallel Pretouch\textsuperscript{9}
- Heap Resizing Ergonomics\textsuperscript{9}
- RemSet Space Reductions\textsuperscript{9,11}
- Marking Space Reductions\textsuperscript{11}

Superscripts indicate JDK versions containing significant work in that area.
Abortable Mixed GCs

• G1 attempts to avoid exceeding the pause time target by using heuristics to select a collection of regions that can be collected within the given time.
• Once started, all the selected regions must be collected.
• Can exceed the pause target if the collection set is too large.
Abortable Mixed GCs

- Split the collection set into mandatory and optional regions
- Mandatory regions are always collected
- Optional regions can be processed incrementally until there is no time left
G1 enhancements between JDK 12 and JDK 13

- Reference PreCleaning\(^9\)
- Fast Evacuation Failure Handling\(^9\)
- Parallelize GC Phases\(^9\+)
- Lazy Thread Initialization\(^9\)
- Lazy RemSet Initialization\(^9\)
- Old Gen On NVDIMM\(^12\)
- Archive regions CDS support\(^9\)
- Rebuild RemSets On the Fly\(^11\)
- Ergonomic Thread Tuning\(^9,11\)
- Parallel Pretouch\(^9\)
- Heap Resizing Ergonomics\(^9\)
- RemSet Space Reductions\(^9,11,13\)
- Superscripts indicate JDK versions containing significant work in that area

**Latency**
- Eliminate Locks\(^{13+}\)
- Elastic TLAB\(^9\)
- Improved Refinement\(^9\)

**Memory**
- Container Awareness\(^9,11,13\)

**Throughput**
- Old Gen On NVDIMM\(^12\)
- Archive regions CDS support\(^9\)
- Rebuild RemSets On the Fly\(^11\)
- Ergonomic Thread Tuning\(^9,11\)
- Parallel Pretouch\(^9\)
- Heap Resizing Ergonomics\(^9\)
- RemSet Space Reductions\(^9,11,13\)
- Superscripts indicate JDK versions containing significant work in that area
G1 performance improvements between JDK 8 and JDK 14

Heap Size: 30GB
OS: Oracle Linux 7.4
HW: Intel Xeon E5-2690 2.9GHz
2 sockets, 8 cores

- 12% improvement in maximum throughput between JDK 8 and JDK 14
- 65% improvement in responsiveness
G1 enhancements JDK 14 and beyond

Latency
- Improved RemSet Scan
- Predictions
- Reduce Barrier overhead

Throughput
- Improved NUMA support

Memory
- RemSet Storage
ZGC - A Scalable Low-Latency Garbage Collector

TB

Multi-terabyte heaps

10ms

Max GC pause time

Easy to tune

15%

Max application throughput reduction
ZGC at a Glance

Concurrent Tracing Compacting Single generation
Region-based NUMA-aware Load barriers Colored pointers
ZGC pause times **do not** increase with the heap or live-set size
ZGC pause times do increase with the root-set size

(Number of Java Threads)
Auto-tuning

Design: No knobs until proven differently!
Application Threads
(aka. "Mutator Threads")
Allocate memory (\texttt{new})
Generate garbage

GC Threads
Collect garbage
Free up memory for allocation

Allocation Rate > Collection Rate → Allocation Stall
Avoiding Allocation Stalls

Generate less garbage  Collect garbage faster
Avoiding Allocation Stalls

Generate less garbage
Avoid allocating objects
Run application more slowly
JIT compiler optimizations

Collect garbage faster
Avoiding Allocation Stalls

Generate less garbage
Collect garbage faster

Avoid allocating objects
Run application more slowly
JIT compiler optimizations
Avoiding Allocation Stalls

Generate less garbage
Avoid allocating objects
Run application more slowly
JIT compiler optimizations

Collect garbage faster
Avoiding Allocation Stalls

Generate less garbage
Avoid allocating objects
Run application more slowly
JIT compiler optimizations

Collect garbage faster
Speed up GC implementation
Use more GC threads
Have application threads help out
Use more memory
“Barriers” – GC Callbacks

Barriers

Small piece of code injected by the JVM
Executed when accessing/updating an object

Store / Load Barriers (aka. Write / Read)

Store: Executed when storing a reference to the Java heap
Load: Executed when reading a reference from the Java heap
Use of Barriers in GCs

Existing GCs make use of store (write) barriers
  True for G1, Parallel, Serial, CMS
  Helps speed up generational support

ZGC uses load barriers
  Mutator threads take on some additional work
ZGC Load barrier

String name = person.name;

<load barrier>

String copy = name; // No barrier
name.isEmpty(); // No barrier
int age = person.age; // No barrier
ZGC Load barrier

String name = person.name;
<load barrier>
ZGC Load barrier

String name = person.name;
<load barrier>
ZGC Load barrier

String name = person.name;
if (!good(name)) {
    name = slow_case(name);
}
ZGC Load barrier

```java
String name = person.name;
if (!good(name)) {
    name = slow_case(name);
}
```

Where to place the good/bad information?
In object? GC side structure?
Colored Pointers

Modern machines use 64-bit addresses/pointers

Exception: Compressed oops

Object pointers stored as 32-bit “indexes”
Colored Pointers

Modern machines use 64-bit addresses/pointers
Exception: CompressedOops
Object pointers stored as 32-bit “indexes”

For ZGC
References (object pointers) are always 64 bit
Colored Pointers

Modern machines use 64-bit addresses/pointers
   Exception: Compressed oops
   Object pointers stored as 32-bit “indexes”

For ZGC
   References (object pointers) are always 64 bit
   Only 64-bit platforms
   No compressed oops

Copyright © 2019 Oracle and/or its affiliates.
Load Barrier Using Colored Pointers

```assembly
mov 0x10(%rax), %rbx       // String n = person.name;
test %rbx, 0x20(%r15)      // Bad color?
jnz slow_path              // Yes -> Enter slow path and
                           // mark/relocate/remap, adjust
                           // 0x10(%rax) and %rbx
```
Load Barrier Using Colored Pointers

```assembly
mov 0x10(%rax), %rbx  // String n = person.name;
test %rbx, 0x20(%r15)  // Bad color?
jnz slow_path         // Yes -> Enter slow path and
                        // mark/relocate/remap, adjust
                        // 0x10(%rax) and %rbx
```

~4% execution overhead
**GC Performance**

**Heap Size:** 128GB  
**OS:** Oracle Linux 7.5  
**HW:** Intel Xeon E5-2690 2.9GHz  
2 sockets, 16 cores (32 hw-threads)
GC Pause Times

- **Average**
- **95%**
- **99%**
- **99.90%**
- **Max**

**GC Pause Times (ms)**

- **Parallel**
- **G1**
- **ZGC**
GC Pause Times (Logarithmic Scale)

![Bar chart showing GC pause times for Parallel, G1, and ZGC with logarithmic scale. The chart includes average, 95%, 99%, and 99.9% values.]
Using ZGC (JDK 11+)

-XX:+UnlockExperimentalVMOptions
-XX:+UseZGC
Tuning options

Look out for Allocation Stalls

Option #1: Increase the max heap size

-`Xmx<size>`
Trade memory for better latency

Option #2: Increase number of GC threads

-`XX:ConcGCThreads=<number>`
Trade CPU-time for better latency
Status/Recent Improvements

JDK 11

First JDK to include open sourced ZGC (Experimental)
Status/Recent Improvements

JDK 11
First JDK to include open sourced ZGC (Experimental)

JDK 12
Concurrent class unloading
Thread-local handshakes
Status/Recent Improvements

JDK 11
First JDK to include open sourced ZGC (Experimental)

JDK 12
Concurrent class unloading
Thread-local handshakes

JDK 13 – Released hours ago!
Max heap size increased to 16 TB (was: 4 TB)
Uncommit unused memory
Linux/aarch64 port
Next up: Productization

Stability

Super late barrier expansion

Support additional platforms
macOS, Windows, …
Potential Future Work

Generational support
  Leverage “Weak generational hypothesis”
  Manage higher allocation rates
  Reduce CPU utilization
Segmented Array Clearing
Chasing Sub-millisecond max pause times
  Concurrent thread stack scanning
Additional latency improvements
  Low latency VM
Further Reading

G1 links
• GC Tuning Guide
  • https://docs.oracle.com/en/java/javase/12/gctuning/garbage-first-garbage-collector.html

ZGC links
• ZGC wiki
  • https://wiki.openjdk.java.net/display/zgc/Main
• GC Tuning Guide
  • https://docs.oracle.com/en/java/javase/12/gctuning/z-garbage-collector1.html
Thank You