Shenandoah: An ultra-low pause time garbage collector for OpenJDK

Christine Flood
Principal Software Engineers
Red Hat

Roman Kennke
Shenandoah

• Why do we need it?
• What does it do?
• How does it work?
• What's the current state?
• What's left to do?
Isn't GC A Solved Problem?

- Not when folks are bending over backwards to avoid GC pauses.
- [https://github.com/OpenHFT/HugeCollections](https://github.com/OpenHFT/HugeCollections)
  - Huge Collections for Java using efficient off heap storage
Isn't GC A Solved Problem?

- Not when Java applications aren't being used for Quality of Service (QOS) applications because of GC pauses keep them from meeting their response time goals.
Isn't GC A Solved Problem?

- Would you buy a car that handled perfectly most of the time but once a day crashed out of control?
- Then you don't want an incremental GC that falls back on stop the world full GCs.
Incremental GC without concurrent evacuation
The problem

- As long as pause time is a function of heap size, GC cannot scale to large heaps
- Shenandoah aims to make pause time heap-size independent
Why do we need another Garbage Collector?

- OpenJDK currently has:
  - SerialGC
  - ParallelGC
  - ParNew/Concurrent Mark Sweep (CMS)
  - Garbage First (G1)
- All programmers care about is end to end throughput and program pause times.
- Shenandoah is all about shorter pause times.
What do you mean by ultra-low pause time?

- Our short term goal is to have < 10ms GC pause times for 100gb+ heaps.
Why not pause-less?

- Our long term goal is to have an entirely pause-less collector.
- Shenandoah is a good first step.
What does the GC do?

- Tracing live objects in the heap
- Copying objects
- Updating references
- Walking the stack frames and other roots
Current Concurrent GCs

- Tracing live objects
- Copying objects
- Updating references
- Walking the stack frames and other roots
Shenandoah

- Tracing live objects
- Copying objects
- Updating references
- Walking the stack frames and other roots

-> pause time is now function of roots-size, e.g. number and size of stacks
Shenandoah 2.0

- Tracing live objects
- Copying objects
- Updating references
- Walking the stack frames and other roots
What exactly are we organizing?
When we reorganize objects we need to copy the objects and update the stack locations to point to their new addresses.
Why concurrent compaction is hard.

Stack Frame Method Foo

Reference

Value 42

Reference

Reference

Stack Frame Method Bar

Value 6.847

Reference

Reference

Reference

Heap

B

A

C
What does Shenandoah do to optimize GC time?

- Only collect part of the heap at one time.
  - No reason to touch long lived data that isn't changing.
- Use multiple processors to GC faster (parallel).
- Do GC work while the program is running (concurrent).
Shenandoah divides the heap into regions.

Region 1

Region 2

Region 3

Region 4

Region 5
We use concurrent marking to keep track of the live data in each region.

<table>
<thead>
<tr>
<th>Regions</th>
<th>Live Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region 1</td>
<td>20k</td>
</tr>
<tr>
<td>Region 2</td>
<td>100k</td>
</tr>
<tr>
<td>Region 3</td>
<td>500k</td>
</tr>
<tr>
<td>Region 4</td>
<td>10k</td>
</tr>
<tr>
<td>Region 5</td>
<td>70k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regions</th>
<th>Live Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region 6</td>
<td>200k</td>
</tr>
<tr>
<td>Region 7</td>
<td>100k</td>
</tr>
<tr>
<td>Region 8</td>
<td>empty</td>
</tr>
<tr>
<td>Region 9</td>
<td>empty</td>
</tr>
<tr>
<td>Region 10</td>
<td>empty</td>
</tr>
</tbody>
</table>
We pick the most garbage-y regions for evacuation.

<table>
<thead>
<tr>
<th>Regions</th>
<th>Live Data</th>
<th>Regions</th>
<th>Live Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region 1</td>
<td>20k From-region</td>
<td>Region 6</td>
<td>200k</td>
</tr>
<tr>
<td>Region 2</td>
<td>100k</td>
<td>Region 7</td>
<td>100k</td>
</tr>
<tr>
<td>Region 3</td>
<td>500k</td>
<td>Region 8</td>
<td>to-region</td>
</tr>
<tr>
<td>Region 4</td>
<td>10k From-region</td>
<td>Region 9</td>
<td>empty</td>
</tr>
<tr>
<td>Region 5</td>
<td>70k</td>
<td>Region 10</td>
<td>empty</td>
</tr>
</tbody>
</table>
Shenandoah GC cycle

- 1\textsuperscript{st} pause: Initialize marking
- Concurrent marking
- 2\textsuperscript{nd} pause: Finish marking and prepare evacuation
- Concurrent evacuation
- 3\textsuperscript{rd} pause: Finish evacuation (and prepare reference updating)
- (Concurrent reference updating)
- (4\textsuperscript{rd} pause: Finish reference updating)
Shenandoah: Current implementation

We use as many threads as are available to do concurrent phases.
Initialize marking pause

- We scan thread stacks and other GC roots
- Mark all objects referenced by roots
- Place all their children on work queues
Initialize marking pause

Stack Frame
Method Foo

Reference
Value
42
Reference
Reference

Stack Frame
Method Bar

Value
6.847
Reference
Reference
Reference
Reference

Object
Array
Object
Object
Object
Object
Concurrent marking

- Mark all live objects by tracing references
- Uses snapshot-at-the-beginning (later...)
- Uses workstealing queues for parallel processing
- Counts live data per region
- Optionally update references from previous evacuation
Concurrent marking

Stack Frame Method Foo

Reference
Value
42
Reference
Reference

Stack Frame Method Bar
Value
6.847
Reference
Reference
Reference
Reference
Snapshot at the beginning

- Anything live at Initial Marking is considered live.
- Anything allocated since Initial Marking is considered live.
What's tricky about SATB?

Start of Concurrent Marking

- Before Y gets marked, it's reassigned to another already-marked object.

Sometime during marking

- We'd miss to mark it because Z won't be processed again.

- Requires write barrier to log previous values (e.g. Y) into a queue.
- SATB queues get processed concurrently.
Finish marking pause

- Rescan all roots to mark any newly referenced objects
- Process weak refs
- Drain remaining SATB buffers
- Drain mark stacks
- Reclaim humonguous regions
- Prepare evacuation:
  - Select from- and to-regions
  - Scan roots to evacuate any root-referenced-objects
Concurrent evacuation

- Scan all from-regions
- Copy all marked objects to to-regions
Update references

• Prepare update references pause:
  • Scan GC roots and update all references to to-regions

• Concurrent reference updating:
  • Scan all regions (except from-regions) and update all refs to point to to-regions
  • Need to ensure only to-space-refs are written
  • Need to ensure that we don't override updates by Java threads
Update references race #1

- GC thread updates reference x.foo to point to Foo'
- Java thread assigns some value Bar to x.foo
  - Bar can point to from-space and we don't update it again
- Solution: resolve forwarding pointers of values before writing them:
- Java thread assigns some value Bar' to x.foo
Update references race #2

- GC thread reads \( x.foo \) being Foo
- Java thread assigns Bar' to \( x.foo \)
- GC thread writes resolved Foo' to \( x.foo \)
  - We miss the Java thread's update!
- Solution: CAS
- GC thread CASes Foo' to \( x.foo \) only when \( x.foo \) is still Foo
Update references

- Optionally piggy-back reference updating to marking
- Advantage: fewer pauses, no extra heap scanning
- Disadvantage: requires to keep around from-regions
How do we do this while the Java threads are running?

- Forwarding Pointers

Accesses go through a forwarding pointer to find the real location of the Object.
Forwarding pointers

```
Foo

From-Region

A
B

To-Region

A'
```
Added benefit to forwarding pointers

We can be lazy about updating references to objects in from regions.
How to move an object while the program is running.

- Read the forwarding pointer to the from-region
- Allocate a temporary copy of the object in a to-region.
- Copy the data.
- CAS the forwarding pointer to point to the new copy.
  - If you succeed carry on.
  - If you fail, another thread already copied the object and you can roll back your allocation and carry on.
Forwarding Pointers - reads

Reading an object in a From-region doesn't trigger an evacuation.

Note: If reads were to cause copying we might have a “read storm” where every operation required copying an object. Our intention is that since we are only copying on writes we will have less bursty behavior.
Forwarding Pointers - writes

Writing an object in a From-Region will trigger an evacuation of that object to a To-Region and the write will occur in there.

Invariant: Writes never occur in from-regions.
Forwarding Pointers – writes of references

We resolve all references before we write them.

Invariant: Never write a reference to a from-region object into a to-region object.
Forwarding Pointers – comparisons

- A == A' => false?
- We resolve references before we compare them
Forwarding Pointers – comparisons

What if another thread evacuates A after we resolved the first operand? $A == A' \Rightarrow \text{false}$?

-> we need evacuating write barriers!
Forwarding Pointers - CAS

- CompareAndSwap(A.X, Foo, Bar)
- We need to copy A to a to-region (write invariant).
- We need to copy Foo to a to-region.
  - (Don't want to get the CAS wrong because the object moved).
- We need to copy Bar to a to-region
  - (Can't write pointers to from-regions into to-region objects).
- We need to retry on CAS failure when Foo' could be == *(A.X)
Forwarding Pointers - CAS

- CompareAndSwap(A.X, Foo, Bar)
Does that mean a write could keep a Java thread from making progress for an unbounded amount of time?

- No
- Copies are bounded by region size.
- Objects that are larger than a region are treated specially.
Costs and Benefits

- Costs
- Space
  - An extra word / object can be expensive if you have a lot of small objects.
- Time
  - Reads and Writes require barriers

- Benefits
- Ultra-low pause times which can be important to interactive and QOS applications.
Read Barriers?

- Unconditional Read Barrier
- Read of Foo.X requires us to first read the forwarding pointer to figure out where Foo is and then read the offset for X.
Write Barriers?

- We need the SATB write barrier which adds previous reference values onto a queue to be scanned.
- We also need write barriers on all writes (even base types) to ensure we copy objects in targeted regions before we write to them.
Current status

- We have something working.
- We can pass small tests specjvm, specjbb
- We have passed smoke test for
  - Eclipse, Thermostat
- We are working on performance tuning
  - Radargun, Elastic Search/Lucene
Current status

- Barriers in interpreter, C1, C2
- Humonguous objects
- Reference support (weak, soft, phantom, JNI)
- JNI critical regions
- System.gc() / Full-GC on out-of-memory
Barrier implementations

- Read barriers:
  - Interpreter: assembly
  - C1: LIR
  - C2: IR

- Write barriers:
  - Interpreter: assembly
  - C1: LIR & assembly
  - C2: IR
  - Runtime calls for actual work
Humonguous objects

- Objects larger than one region
- Span several regions
- Are never copied
- Are reclaimed right after marking
Reference support

- Regular marking stops at weakrefs
- After marking, weakrefs get processed at pause
- Shenandoah processes weakrefs in parallel
- Large datastructures behind weakrefs impede pause times!!
Reference support

- Regular marking stops at weakrefs
- After marking, weakrefs get processed at pause
- Shenandoah processes weakrefs in parallel
- Large datastructures behind weakrefs impede pause times!!
JNI critical regions

- void * GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean *isCopy);
- void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void *carray, jint mode);
- JVM has two options:
  - Return direct pointer to array -> array must not move
  - Return copy of array -> copy overhead
JNI critical regions

- Shenandoah delays evacuation until all critical regions have been left
- Native code may enter critical regions during concurrent evacuation!
  - Shorter JNI stalls than other Gcs
  - Possible through write barrier on array before entering critical region
System.gc()

- Used for System.gc() calls and emergency collections on out of memory
- Implemented by STW mark-compact
- All garbage gets squeezed out
- Maximum compaction to start-of-heap
- Care must be taken to align on region boundaries
- Humonguous objects need special treatment
- Requires ability to abort ongoing GC
Abort ongoing GC

- While marking
  - Scratch mark bitmaps
  - Cleanup
  - Skip evacuation
- While evacuating
  - Stop copying immediately
  - Still need to update references (even from-space)
  - Otherwise risk object clones
- While updating refs: sorry, can't stop
Why aren't we done yet?

- GC pauses are encouraging
- Throughput is not yet where we want it to be
  - Cost of barriers?
  - Barriers inhibiting C2 optimizations?
  - Inefficiencies in the GC?
  - Less than optimum heuristics?
- Integrate into upstream OpenJDK
Performance Tuning

• One very important area for performance tuning will be heuristics.
  • When to start concurrent marking?
  • How large should the collection set be?
  • How many concurrent threads to use?
  • How to deal with sudden allocation bursts?
  • How to ensure not running out of memory during GC?
  • Locality considerations?
Performance tuning

- Insert barriers after C2 optimizations
- Implement barrier-specific optimizations
  - Exploit Java Memory Model
- Improve GC algorithm
  - E.g. use prefetching, buffering, etc

•
Shenandoah in OpenJDK

- JEP 189: http://openjdk.java.net/jeps/189
- Will propose project @ OpenJDK soon
Encouraging preliminary results
SpecJVM2008 compiler

- Not our target application.
- We are just starting performance tuning.
- Shenandoah
  - Initial Mark (avg=5.65ms, max=9.53ms, total=1.40s)
  - Final Mark (avg=8.74ms, max=15.43ms, total=2.17s)
- As compared to G1
  - (avg=31.38ms, max=75.48ms, total=6.84s)
Future Work

- Finish big application testing.
- Move the barriers to right before code generation.
- Barrier-specific C2 opts?
- Exploit Java Memory Model?
- Heuristics tuning!
- Generational Shenandoah?
- Remembered Sets for updating roots and freeing memory sooner?
- Round Robin Thread Stopping?
- NUMA Aware?
References

- “Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware” - Brooks
- “Garbage-first garbage collection” - Detlefs, Flood, Heller, Printezis.
More information

- Download the code and try it
  - http://icedtea.classpath.org/wiki/Shenandoah
- Blogs
  - http://christineflood.wordpress.com/
  - http://rkeninke.wordpress.com/
- Email
  - chf@redhat.com
  - rkeninke@redhat.com