Shenandoah GC

...and how it looks like in September 2017

Aleksey Shipilëv

shade@redhat.com
@shipilev
Disclaimers

This talk:

1. ...assumes some knowledge of GC internals: this is implementors-to-implementors talk, not implementors-to-users – we are here to troll for ideas

2. ...briefly covers successes, and thoroughly covers challenges: mind the *availability heuristics* that can confuse you into thinking challenges outweigh the successes

3. ...covers many topics, so if you have blinked and lost the thread of thought, wait a little up until the next (ahem) safepoint
Overview
Overview: Landscape

Serial, Parallel:

Young GC — Old GC

Copy — Mark — Compact
Overview: Landscape

Serial, Parallel:
- Copy
- Mark
- Compact

CMS:
- Copy
- Concurrent Mark
- Conc. Sweep

Still a pause :(  
Init Mark  
Finish Mark 

Does not solve fragmentation :(  

Young GC  Old GC
Overview: Landscape

Young GC

Serial, Parallel:
- Copy
- Mark
- Compact

CMS:
- Copy
- Concurrent Mark
- Conc. Sweep

G1:
- Copy
- Concurrent Mark
- Compact

Still a pause :(

Does not solve fragmentation :(

Smaller, adjustable, but still a pause :(

Smaller, adjustable, but still a pause :(
Overview: Landscape

Serial, Parallel:
- Copy
- Mark
- Compact

CMS:
- Copy
- Concurrent Mark
- Conc. Sweep
  - Init Mark
  - Finish Mark

G1:
- Copy
- Concurrent Mark
- Compact
  - Init Mark
  - Finish Mark

Shenandoah (in development):
- Emit (╯°□°）╯︵ ┻━┻
- Concurrent Mark
- Conc. Compact
  - Init Mark
  - Finish Mark
Overview: Key Idea

Brooks forwarding pointer to help concurrent copying:

fwdptr is attached to every object, at all times
Overview: Key Idea

Brooks forwarding pointer to help concurrent copying:

fwdptr always points to most actual (to-space) copy, and gets atomically updated during evacuation
Overview: Key Idea

Brooks forwarding pointer to help concurrent copying:

Barriers maintain the to-space invariant:
«All writes happen into to-space copy»
Overview: Key Idea

Brooks forwarding pointer to help concurrent copying:

Barriers also help to select the to-space copy for reading
Overview: Key Idea

Brooks forwarding pointer to help concurrent copying:

Allows to update the heap references concurrently too
Overview: GC Cycle

Regular cycle:

1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
3. Concurrent update references (optional)
Overview: GC Cycle

Regular cycle:

1. Snapshot-at-the-beginning concurrent mark
Overview: GC Cycle

Regular cycle:
1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
Overview: GC Cycle

Regular cycle:
1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
3. Concurrent update references (optional)
Overview: GC Cycle

Regular cycle:
1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
3. Concurrent update references (optional)
Successes
Successes: Almost Concurrent Works!

LRUFragger, 100 GB heap, ≈ 80 GB LDS:

Pause Init Mark 0.437ms
Concurrent marking 76780M–>77260M(102400M) 700.185ms
Pause Final Mark 77260M–>77288M(102400M) 0.698ms
Concurrent cleanup 77288M–>77296M(102400M) 0.176ms
Concurrent evacuation 77296M–>85696M(102400M) 405.312ms
Pause Init Update Refs 0.038ms
Concurrent update references 85700M–>85928M(102400M) 319.116ms
Pause Final Update Refs 85928M–>85928M(102400M) 0.351ms
Concurrent cleanup 85928M–>56620M(102400M) 14.316ms
Successes: Almost Concurrent Works!

LRUFragger, 100 GB heap, ≈ 80 GB LDS:

Pause Init Mark 0.437ms
Concurrent marking 76780M→77260M(102400M) 700.185ms
Pause Final Mark 77260M→77288M(102400M) 0.698ms
Concurrent cleanup 77288M→77296M(102400M) 0.176ms
Concurrent evacuation 77296M→85696M(102400M) 405.312ms
Pause Init Update Refs 0.038ms
Concurrent update references 85700M→85928M(102400M) 319.116ms
Pause Final Update Refs 85928M→85928M(102400M) 0.351ms
Concurrent cleanup 85928M→56620M(102400M) 14.316ms
Successes: Concurrent Means Freedom

Mostly concurrent GC is very liberating!

- No rush doing concurrent phases: slow concurrent phase means more frequent cycles ⇒ steal more cycles from application, not pause it extensively

- Heuristics mistakes are (usually) much less painful: diminished throughput, but not increased pauses

- Control the GC cycle time budget: -XX:ConcGCThreads=...
Successes: Progress

Concurrent collector runs GC cycles without blocking mutator progress (translation: BMU/MMU is really good)

That means, we can do:

- ...thousands of GC cycles per minute ⇒ Very efficient testing that surface the rarest bugs
- ...**continuous** GC cycles when capacity is overwhelming ⇒ Ultimate sacrifice of throughput for latency
- ...**periodic** GCs without significant penalty ⇒ Idle applications get their floating garbage purged
Successes: Non-Generational Workloads

Shenandoah does not **need** Generational Hypothesis to hold true in order to operate efficiently

- Prime example: LRU/ARC-like in-memory caches
- It would **like** GH to be true: immediate garbage regions can be immediately reclaimed after mark, and cycle shortcuts
- **Partial** collections may use region age to focus on «younger» regions
Successes: Barriers Injection

Educated Bystander concern:
Where to inject the barriers?

Reality:

- Most heap accesses from VM are done via native accessors
- Most VM parts (e.g. compilers) hold on to JNI handles
- A very few naked reads and stores are done
- Story gets much better with JEP 304 «GC interface»
- Internal heap verification helps to catch missing barriers
Successes: Heap Management

Regionalized heap allows (un)committing individual regions

Some are willing to trade increased peak footprint for better idle footprint: per-region heap uncommit + periodic GCs
Successes: Releases

Easy to access (development) releases: try it now!

- Development in separate JDK 10 forest, regular backports to separate JDK 9 and 8u forests
- JDK 8u backports ship in RHEL 7.4+, Fedora 24+
- Nightly development builds (tarballs, Docker images)

```
docker run -it --rm shipilev/openjdk:10-shenandoah \
  java -XX:+UseShenandoahGC -Xlog:gc -version
```
Challenges
Challenges: Footprint Overhead

Shenandoah requires additional word per object for forwarding pointer at all times

- 1.5x worst case and 1.05-1.10x average overhead – but, counted in Java heap, not native structures – easier capacity planning
- Current pointer is uncompressed, no gain to compress due to object alignment constraints
- Moving fwdptr into synthetic object field promises substantial improvements – but, read barriers are already very overheady
Challenges: Barriers Overhead

Shenandoah requires much more barriers

1. SATB barriers for regular cycles
2. Write barriers on all stores, not only reference stores
3. Read barriers on almost all heap reads
4. Other exotic flavors of barriers: acmp, CAS, clone, ...
Challenges: Read Barriers

```assembly
# Read Barrier: dereference via fwdptr
mov -0x8(%r10),%r10  # obj = *(obj - 8)

# read the field at offset 0x30
mov 0x30(%r10),%r10d  # val = *(obj + 0x30)
```

- Very simple: single instruction
- Very frequent: before almost every heap read
- Optimizeable: move heap accesses ⇒ move the barriers
- Accounts for 0..15% throughput hit, depending on the workload
Challenges: Write Barriers

# Read TLS flag and see if evac is enabled
movzbl 0x3d8(%r15),%r11d  # flag = *(TLS + 0x3d8)
test %r11d,%r11d          # if (flag) ...
jne OMG-EVAC-ENABLED     # No, no, no!

# Not enabled: read barrier
mov -0x8(%rbp),%r10      # obj = *(obj - 8)

# Store into the field!
mov %r10,0x30(%r10)      # *(obj + 0x30) = r10
Challenges: Write Barriers

```assembly
# Read TLS flag and see if evac is enabled
movzbl 0x3d8(%r15),%r11d        # flag = *(TLS + 0x3d8)
test %r11d,%r11d                 # if (flag) ...
jne OMG-EVAC-ENABLED            # No, no, no!

# Not enabled: read barrier
mov  -0x8(%rbp),%r10             # obj = *(obj - 8)

# Store into the field!
mov  %r10,0x30(%r10)             # *(obj + 0x30) = r10
```

Writing to field? Locking on object? Computing identity hash code? Writing down new klass? All those are object stores, all require WB
Challenges: Write Barriers

# Read TLS flag and see if evac is enabled
movzbl 0x3d8(%r15),%r11d  # flag = *(TLS + 0x3d8)
test %r11d,%r11d  # if (flag) ... 
jne OMG-EVAC-ENABLED  # No, no, no!

# Not enabled: read barrier
mov 0x3d8(%r15),%r11d  # flag = *(TLS + 0x3d8)
test %r11d,%r11d  # if (flag) ... 
jne OMG-EVAC-ENABLED  # No, no, no!

mov -0x8(%rbp),%r10    # obj = *(obj - 8)

# Store into the field!
mov %r10,0x30(%r10)    # *(obj + 0x30) = r10

Writes are rare, fast-path is fast, throughput overhead is 0..5%
Challenges: Exotic Barriers

**Shenandoah-specific** barriers: making sure comparisons work when both copies of the object are reachable.

Unequal machine ptrs $\neq$ unequal Java references now!

- **acmp barrier**: on comparison failure, do RBs, compare again
- Java ref comparisons in native VM code have to do it too
- **CAS barrier**: on CAS failure, do magic to dodge false positives
- Normally cost $< 1\%$ throughput
Challenges: Compiler Support

The key thing to cope with barriers overhead is Shenandoah-specific compiler optimizations (this is also the major source of interesting bugs)

- Hoisting read and write barriers out of the loops
- Eliminating barriers on known new objects, known constants
- Bypassing read barriers on unordered reads, e.g. final-s
- Optimizeable barriers straight in IR
- Coalescing barriers: SATB+WB, back-to-back barriers, etc
### Challenges: Compiler Support

<table>
<thead>
<tr>
<th>Test</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G1</td>
<td>Shen</td>
</tr>
<tr>
<td>Cmp</td>
<td>79</td>
<td>74</td>
</tr>
<tr>
<td>Cpr</td>
<td>125</td>
<td>86</td>
</tr>
<tr>
<td>Cry</td>
<td>79</td>
<td>63</td>
</tr>
<tr>
<td>Drb</td>
<td>78</td>
<td>70</td>
</tr>
<tr>
<td>Mpa</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>Sci</td>
<td>42</td>
<td>31</td>
</tr>
<tr>
<td>Ser</td>
<td>1639</td>
<td>1279</td>
</tr>
<tr>
<td>Sun</td>
<td>99</td>
<td>75</td>
</tr>
<tr>
<td>Xml</td>
<td>89</td>
<td>70</td>
</tr>
</tbody>
</table>

C1 codegens good barriers, but C2 also does high-level optimizations

---

1Caveat: Author made this experiment while inebriated after conference dinner, so...
Challenges: STW Woes

Pauses $\approx 1 \text{ ms}$ leave little time budget to deal with, but need to scan roots, cleanup runtime stuff, walk over regions...

Consider:

- Thread wakeup latency is easily more than 200 $\mu$s: parallelism does not give you all the bang – some parallelism is still efficient.
- Processing 10K regions means taking 100 $\text{ns}$ per region. Example: you can afford marking regions as «dirty», but cannot afford actually recycling them during the pause.
In Progress
In Progress: VM Support

Pauses $\leq 1 \text{ms}$ require more runtime support

Some examples:

- Time-To-SafePoint takes about that even without loopy code
- Safepoint auxiliaries: stack scans for method aging takes $> 1 \text{ms}$, cleanup can easily take $\gg 1 \text{ms}$
- Lots of roots, many are hard/messy to scan concurrently or in parallel: StringTable, synchronizer roots, etc.
In Progress: Partials

Full heap concurrent cycle takes the *throughput* toll on application.

Idea: partial collections!

- Requires knowing what parts of heap to scan for incoming refs
  - Card Table for Serial, Parallel, CMS
  - Card Table + Remembered Sets for G1
- Differs from regular cycle: selects the collection set without prior marking, thus more conservative
- *Generational* is the special case of partial
Concurrent collector allows for very coarse «connection matrix»:
Concurrent collector allows for very coarse «connection matrix»:
In Progress: Partials, Connection Matrix

Concurrent collector allows for very coarse «connection matrix»:
In Progress: Partial, Status

Current partial machinery does work!

- Implemented GC infra, and matrix barriers in all compilers
- Can use timestamps to bias towards younger and older regions
- Caveat, they are STW in current experiment:
  
  \[
  \begin{align*}
  \text{GC(7) Pause Partial} & \quad 2103M \rightarrow 2106M(10240M) \quad 4.209\text{ms} \\
  \text{GC(7) Concurrent cleanup} & \quad 2106M \rightarrow 59M(10240M) \quad 5.288\text{ms}
  \end{align*}
  \]

- Anecdote: sometimes, partial STW is shorter than regular STWs
In Progress: Concurrent Partial

Next step:
making partial collections *concurrent*
(work in progress)

Q: Matrix consistency during concurrent partial?
Q: New barriers required?
Q: Regular concurrent cycle is the special case of partial?
In Progress: Traversal Order

Spot the trouble:

Application active → Init Mark → Concurrent mark → Application active

Concurrent evacuation → Application active → Concurrent update refs → Application Active

Application Active → Final-UR → Final Mark

Application active → Init-UR → Concurrent mark → Application active
Separate marking and evacuation phases mean collector maintains the *allocation* order, not the *traversal* order.
In Progress: Traversal Order

Spot the trouble:

Q: Can coalesce evacuation and update-refs?
Q: Concurrent Partial can coalesce the phases?

Slide 30/35. «Shenandoah GC», Aleksey Shipilëv, 2017, D:20171002122223+02'00'
In Progress: Humongous and $2^K$ allocs

new byte [1024*1024] is the best fit for regionalized GC?

- Actually, in G1-style humongous allocs, the worst fit: objects have headers, and $2^K$-sized alloc would barely not fit, wasting one of the regions

Q: Can be redone with segregated-fits freelist maintained separately?
In Progress: Application Pacing

Concurrent collector GC relies on collecting faster than applications allocate: applications **always** see there is available memory

- In practice, this is frequently true: applications rarely do allocations only, GC threads are high-priority, there enough space to absorb allocations while GC is running...
- In some cases of *overloaded* heap, application outpaces GC, yielding Allocation Failure, and prompting STW

Q: Pace the application when heap is close to exhaustion?
In Progress: SATB or IU

SATB overestimates liveness:
all new allocations during mark are implicitly live

- Keeps lots of floating garbage that would need to wait for another cycle to be collected
- Has interesting implications on weak references processing: e.g. deadly embracing `Reference.get()`...

Q: Is incremental update more suitable here?
Conclusion
Conclusion: Ready for Experimental Use

Try it.
Break it.
Report the successes and failures.

https://wiki.openjdk.java.net/display/shenandoah/Main