Shenandoah GC
Part I: The Garbage Collector That Could

Aleksey Shipilëv
shade@redhat.com
@shipilev
Safe Harbor / Тихая Гавань

Anything on this or any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.
Basics
Basics: OpenJDK GCs Landscape

Serial, Parallel:

- Copy
- Mark
- Compact

Young GC  Old GC
Basics: OpenJDK GCs 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
Basics: OpenJDK GCs 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

- Notes:
  - Does not solve fragmentation :(  
  - Smaller, adjustable, but still a pause :(
Basics: OpenJDK GCs Landscape

Serial, Parallel:
- Copy
- Mark
- Compact

CMS:
- Copy
- Concurrent Mark
- Conc. Sweep
  - Init Mark
  - Finish Mark
- Does not solve fragmentation :(

G1:
- Copy
- Concurrent Mark
- Compact
  - Init Mark
  - Finish Mark
- Smaller, adjustable, but still a pause :(

Shenandoah, ZGC:
- Concurrent Mark
- Conc. Compact
  - Init Mark
  - Finish Mark
- Smaller, adjustable, but still a pause :(
Basics: Concurrent GC Only For Large Heaps?
Basics: Concurrent GC Only For Large Heaps?

\[ Latency_{stw} = \alpha \times Size_{heap} \times MemRefs_{stw} \times MemLatency_{avg} \]
Latency_{stw} = \alpha \cdot Size_{heap} \cdot Mem\ Refs_{stw} \cdot MemLatency_{avg}

- Heap size collected per GC cycle, MB
- Memory references during STW, accesses/MB
- End-to-end memory latency, ns/access
## Basics: Concurrent GC Only For Large Heaps?

<table>
<thead>
<tr>
<th>Observation</th>
<th>$\alpha \times Size_{heap}$</th>
<th>$MemRefs_{stw}$</th>
<th>$MemLatency_{avg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large heap</td>
<td>↑↑</td>
<td>↓↓</td>
<td>$\approx$</td>
</tr>
</tbody>
</table>

- Large heap: large live data sets $\Rightarrow$ need concurrent GC
Basics: Concurrent GC Only For Large Heaps?

<table>
<thead>
<tr>
<th>Observation</th>
<th>( \alpha \times \text{Size}_{heap} )</th>
<th>( \text{MemRefs}_{stw} )</th>
<th>( \text{MemLatency}_{avg} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large heap</td>
<td>↑↑</td>
<td>↓↓</td>
<td>≈</td>
</tr>
<tr>
<td>Slow hardware</td>
<td>≈</td>
<td>↓↓</td>
<td>↑↑</td>
</tr>
</tbody>
</table>

- Large heap: large live data sets ⇒ need concurrent GC
- Slow hardware: memory is slow ⇒ need concurrent GC
Basics: Slow Hardware

Raspberry Pi 3, running springboot-petclinic:

# -XX:+UseShenandoahGC
Pause Init Mark 8.991ms
Concurrent marking 409M->411M(512M) 246.580ms
Pause Final Mark 3.063ms
Concurrent cleanup 411M->89M(512M) 1.877ms

# -XX:+UseParallelGC
Pause Young (Allocation Failure) 323M->47M(464M) 220.702ms

# -XX:+UseG1GC
Pause Young (G1 Evacuation Pause) 410M->38M(512M) 164.573ms
Basics: Releases

Easy to access (development) releases: try it now!
https://wiki.openjdk.java.net/display/shenandoah/

- Dev follows latest JDK, backports to 11, 10, and 8
- JDK 8 backport ships in RHEL 7.4+, Fedora 24+
- JDK 11 backport ships in Fedora 27+
- Nightly development builds (tarballs, Docker images)

```
docker run -it --rm shipilev/openjdk-shenandoah \
   java -XX:+UseShenandoahGC -Xlog:gc -version
```
Basics: This Message Is Brought To You By

- IMHO, discussing gory GC details without «GC Handbook» is a waste of time
- Many GCs appear super-innovative, but in fact they reuse (or reinvent) ideas from the GC Handbook
- Combinations of those ideas give rise to many concrete GCs
Overview
Overview: Heap Structure

Shenandoah is a *regionalized* GC

- Heap division, humongous regions, etc are similar to G1
- Collects garbage regions first by default
- Not generational by default, no young/old separation, even temporally
- Tracking inter-region references is not needed by default
Overview: Usual Cycle

Three major phases:
Overview: Usual Cycle

Three major phases:
1. Concurrent marking
Overview: Usual Cycle

Three major phases:
1. Concurrent marking
2. Concurrent evacuation
Overview: Usual Cycle

Three major phases:
1. Concurrent marking
2. Concurrent evacuation
3. Concurrent update references (optional)
Overview: Usual Cycle

Three major phases:
1. Concurrent marking
2. Concurrent evacuation
3. Concurrent update references (optional)
Overview: Usual Log

LRUFragger, 100 GB heap, \( \approx 80 \) GB live data:

Pause Init Mark 0.227ms
Concurrent marking 84864M->85952M(102400M) 1386.157ms
Pause Final Mark 0.806ms
Concurrent cleanup 85952M->85985M(102400M) 0.176ms
Concurrent evacuation 85985M->98560M(102400M) 473.575ms
Pause Init Update Refs 0.046ms
Concurrent update references 98560M->98944M(102400M) 422.959ms
Pause Final Update Refs 0.088ms
Concurrent cleanup 98944M->84568M(102400M) 18.608ms
Overview: Usual Log

LRUFragger, 100 GB heap, $\approx$ 80 GB live data:

Pause Init Mark 0.227ms
Concurrent marking 84864M–>85952M(102400M) 1386.157ms
Pause Final Mark 0.806ms
Concurrent cleanup 85952M–>85985M(102400M) 0.176ms
Concurrent evacuation 85985M–>98560M(102400M) 473.575ms
Pause Init Update Refs 0.046ms
Concurrent update references 98560M–>98944M(102400M) 422.959ms
Pause Final Update Refs 0.088ms
Concurrent cleanup 98944M–>84568M(102400M) 18.608ms
Phases
Mark: Reachability

To catch a garbage, you have to think like a garbage, know if there are references to the object
To catch a garbage, you have to *think like a garbage*—know if there are references to the object.

Three basic approaches:

1. **No-op**: ignore the problem (*Epsilon GC*)
To catch a garbage, you have to *think like a garbage*—know if there are references to the object.

Three basic approaches:

1. **No-op**: ignore the problem (*Epsilon GC*)
2. **Reference counting**: track the number of references, and when refcount drops to 0, treat the object as garbage
Mark: Reachability

To catch a garbage, you have to *think like a garbage*—know if there are references to the object.

Three basic approaches:

1. **No-op**: ignore the problem (*Epsilon GC*)
2. **Reference counting**: track the number of references, and when refcount drops to 0, treat the object as garbage
3. **Tracing**: walk the object graph, find reachable objects, treat *everything else* as garbage
Mark: Three-Color Abstraction

Assign *colors* to the objects:

1. White: not yet visited
2. Gray: visited, but references are not scanned yet
3. Black: visited, and fully scanned
Mark: Three-Color Abstraction

Assign *colors* to the objects:

1. White: not yet visited
2. Gray: visited, but references are not scanned yet
3. Black: visited, and fully scanned

Daily Blues:
«All the marking algorithms do is coloring white gray, and then coloring gray black»
When application is stopped, everything is trivial!
Nothing messes up the scan...
Mark: Stop-The-World Mark

Found all roots, color them Black, because they are implicitly reachable
Mark: Stop-The-World Mark

References from Black are now Gray, scanning Gray references
Mark: Stop-The-World Mark

Finished scanning Gray, color them Black; new references are Gray
Mark: Stop-The-World Mark

Gray → Black;
reachable from Gray → Gray
Mark: Stop-The-World Mark

Gray → Black;
reachable from Gray → Gray
Mark: Stop-The-World Mark

Gray → Black;
reachable from Gray → Gray
Mark: Stop-The-World Mark

Gray → Black;
reachable from Gray → Gray
Mark: Stop-The-World Mark

Finished: everything reachable is Black; all garbage is White
Concurrent Mark: Mutator Problems

With *concurrent* mark everything gets complicated: the application runs and actively mutates the object graph during the mark.

We contemptuously call it *mutator* because of that.
Concurrent Mark: Mutator Problems

Wavefront is here, and starts scanning the references in Gray object...
Concurrent Mark: Mutator Problems

Mutator removes the reference from Gray... and inserts it to Black!
Concurrent Mark: Mutator Problems

...or mutator inserted the reference to *transitively reachable* White object into Black
Concurrent Mark: Mutator Problems

...or mutator inserted the reference to \textit{transitively reachable} White object into Black
Mark had finished, and boom: we have reachable White objects, which we will now reclaim, corrupting the heap.
Another quirk: created new **new object**, and inserted it into Black
There are at least three approaches to solve this problem. All of them require intercepting heap accesses. Short on time, we shall discuss what G1 and Shenandoah are doing.
Concurrent Mark: SATB

Color all *removed* referents Gray
Concurrent Mark: SATB

Color all new objects **Black**
Concurrent Mark: SATB

Finishing...
CONCURRENT MARK: SATB

Done!
Concurrent Mark: SATB

«Snapshot At The Beginning»:
marked all reachable at mark start
Concurrent Mark: SATB Barrier

```assembly
# check if we are marking
testb 0x2, 0x20(%r15)
jne OMG-MARKING
BACK:
   # ... actual store follows ...

# somewhere much later
OMG-MARKING:
   # tens of instructions that add old value
   # to thread-local buffer, check for overflow,
   # call into VM slowpath to process the buffer
   ...
   jmp BACK
```
Concurrent Mark: Two Pauses\(^1\)

**Init Mark:** stop the mutator to avoid races
   1. Walk and mark all roots
   2. Arm SATB barriers

**Final Mark:** stop the mutator to avoid races
   1. Drain the thread buffers
   2. Finish work from buffer updates

\(^1\)These can actually be concurrent, but that is not very practical
Concurrent Mark: Two Pauses

**Init Mark:** stop the mutator to avoid races
1. Walk and mark all roots ← **most heavy-weight**
2. Arm SATB barriers

**Final Mark:** stop the mutator to avoid races
1. Drain the thread buffers
2. Finish work from buffer updates ← **most heavy-weight**

---

¹These can actually be concurrent, but that is not very practical
## Concurrent Mark: Barriers Cost$^2$

<table>
<thead>
<tr>
<th></th>
<th>Throughput hit, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>SATB</td>
<td></td>
</tr>
<tr>
<td>Cmp</td>
<td>-1.6</td>
</tr>
<tr>
<td>Cps</td>
<td>-3.5</td>
</tr>
<tr>
<td>Cry</td>
<td></td>
</tr>
<tr>
<td>Der</td>
<td>-1.6</td>
</tr>
<tr>
<td>Mpg</td>
<td></td>
</tr>
<tr>
<td>Smk</td>
<td></td>
</tr>
<tr>
<td>Ser</td>
<td></td>
</tr>
<tr>
<td>Sfl</td>
<td></td>
</tr>
<tr>
<td>Xml</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

$^2$Performance compared to STW Shenandoah with all barriers disabled.
Concurrent Mark: Observations

1. Extended concurrency needs to pay with more barriers
   - Ideal STW GC beats ideal concurrent GC on pure throughput
   - If you do not care about GC pauses, just use good STW GC
   - Empty GC log does not mean no GC overhead
Concurrent Mark: Observations

1. Extended concurrency needs to pay with more barriers
   - Ideal STW GC beats ideal concurrent GC on pure throughput
   - If you do not care about GC pauses, just use good STW GC
   - Empty GC log does not mean no GC overhead

2. Hiding references from mark prolongs final mark pause
   - Weak references with unreachable referents, finalizers
   - «Old» objects hidden in SATB buffers
Problem: there is the object, the object is referenced from somewhere, need to move it to new location.
Step 1: Stop The World, evasive maneuver to distract mutator from looking into our mess.
Copy: Stop-The-World

Step 2: Copy the object with all its contents
Copy: Stop-The-World

Step 3.1:
Update all references: save the pointer that forwards to the copy.
Step 3.2: Update all references: walk the heap, replace all refs with fwdptr destination
Step 3.2: Update all references: walk the heap, replace all refs with fwdptr destination
Copy: Stop-The-World

Everything is fine in the world, set the mutators free! Done!
Concurrent Copy: Mutator Problems

With **concurrent** copying everything gets is significantly harder: the application writes into the objects while we are moving the same objects!

http://vernova-dasha.livejournal.com/77066.html
While object is being moved, there are two copies of the object, and both are reachable!
Concurrent Copy: Mutator Problems

Thread A writes $y = 4$ to one copy, and Thread B writes $x = 5$ to another. Which copy is correct now, huh?
Concurrent Copy: Brooks Pointers

Idea:
Brooks pointer: object version change with additional atomically changed indirection
**Concurrent Copy: Brooks Pointers**

**Step 1:**
Copy the object, initialize its forwarding pointer to self.
Concurrent Copy: Brooks Pointers

We now have the copy of the object, but no one knows about it.
Concurrent Copy: Brooks Pointers

Step 2: CAS! Atomically install forwarding pointer to point to new copy. If CAS had failed, discover the copy via forwarding pointer.
Concurrent Copy: Brooks Pointers

Step 3: Rewrite the references at our own pace in the rest of the heap
If somebody reaches the old copy via the old reference, it has to dereference via fwdptr and discover the actual object copy!
Concurrent Copy: Brooks Pointers

Step 4:
All references are updated, recycle the from-space copy
Concurrent Copy: Brooks Pointers

```
3LIDE29/80.j3HENANDOAHGCz,!LEKSEy3HIPIL
```

```
V,2018,D:20180914113310+02'00'
```

Done!

```
Fwd Ptr
Headers
  x = 5
  y = 4
  z = 3
```

"From" space

"To" space

Slide 29/80. «Shenandoah GC», Aleksey Shipilëv, 2018, D:20180914113310+02'00'
Write Barriers: Motivation

To-space invariant: Writes should happen in to-space only, otherwise they are lost when cycle is finished.
Write Barriers: Fastpath

testb 0x1, 0x20(%r15)  # Heap is stable?
jne  OMG-FORWARDED-OBJECTS
BACK:
    # ... actual store follows ...

# somewhere much later
OMG-FORWARDED-OBJECTS:
    mov  -0x8(%rbp),%r10  # Resolve via fwdptr
    testb 0x4, 0x20(%r15)  # Evacuation in progress?
jne  OMG-EVACUATION
jmp  BACK
Write Barriers: Slowpath

stub WriteBarrier(obj) {
    if (in-collection-set(obj) && // target is in from-space
data-ptrs-to-self(obj)) { // no copy yet
        val copy = copy(obj);
        if (CAS(fwd-ptr-addr(obj), obj, copy)) {
            return copy; // success!
        } else {
            return fwd-ptr(obj); // someone beat us to it
        }
    }
}
Write Barriers: GC Evacuation Code

```c
stub evacuate(obj) {
    if (in-collection-set(obj) && // target is in from-space
        fwd-ptrs-to-self(obj)) { // no copy yet
        copy = copy(obj);
        CAS(fwd-ptr-addr(obj), obj, copy);
    }
}
```

Termination guarantees:
Always copy **out of** collection set.
Double forwarding is the GC error.
### Write Barriers: Barriers Cost

<table>
<thead>
<tr>
<th></th>
<th>Throughput hit, %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SATB</td>
</tr>
<tr>
<td>Cmp</td>
<td>-1.6</td>
</tr>
<tr>
<td>Cps</td>
<td>-3.5</td>
</tr>
<tr>
<td>Cry</td>
<td>-1.1</td>
</tr>
<tr>
<td>Der</td>
<td>-1.6</td>
</tr>
<tr>
<td>Mpg</td>
<td>-2.1</td>
</tr>
<tr>
<td>Smk</td>
<td>-0.5</td>
</tr>
<tr>
<td>Ser</td>
<td>-4.0</td>
</tr>
<tr>
<td>Sfl</td>
<td>-2.7</td>
</tr>
<tr>
<td>Xml</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

2Performance compared to STW Shenandoah with all barriers disabled
Write Barriers: Observations

1. Shenandoah needs WB on **all** stores
   - Field stores – obviously
   - Locking the object – changes header ⇒ needs WB
   - Computing identity hash code – changes header ⇒ needs WB
Write Barriers: Observations

1. Shenandoah needs WB on all stores
   - Field stores – obviously
   - Locking the object – changes header $\Rightarrow$ needs WB
   - Computing identity hash code – changes header $\Rightarrow$ needs WB

2. Passive WB cost is low
   - Writes, even the primitive ones, are rare
   - The cost of L1-load-test-branch is low
Write Barriers: Observations

1. Shenandoah needs WB on all stores
   - Field stores – obviously
   - Locking the object – changes header ⇒ needs WB
   - Computing identity hash code – changes header ⇒ needs WB

2. Passive WB cost is low
   - Writes, even the primitive ones, are rare
   - The cost of L1-load-test-branch is low

3. Active WB cost is moderate
   - GC does the bulk of the work
   - In optimized barrier paths, fwdptr CAS is the major cost
Heap reads have to (?) dereference via the forwarding pointer, to discover the actual object copy.
Read Barriers: Implementation

```c
# read barrier: dereference via fwdptr
mov -0x8(%r10),%r10  # obj = *(obj - 8)

# ...actual read from %r10 follows...
```
Read Barriers: Implementation

```c
# read barrier: dereference via fwdptr
mov     -0x8(%r10),%r10  # obj = *(obj - 8)

# ...actual read from %r10 follows...
```

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Score</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>base</td>
<td>+3 RBs</td>
</tr>
<tr>
<td>time</td>
<td>4.6 ± 0.1</td>
<td>5.3 ± 0.1</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>12.3 ± 0.2</td>
<td>15.1 ± 0.3</td>
</tr>
<tr>
<td>cycles</td>
<td>18.7 ± 0.3</td>
<td>21.6 ± 0.3</td>
</tr>
<tr>
<td>instructions</td>
<td>26.6 ± 0.2</td>
<td>30.3 ± 0.3</td>
</tr>
</tbody>
</table>
## Read Barriers: Barriers Cost

<table>
<thead>
<tr>
<th></th>
<th>Throughput hit, %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SATB</td>
</tr>
<tr>
<td>Cmp</td>
<td>-1.6</td>
</tr>
<tr>
<td>Cps</td>
<td>-3.5</td>
</tr>
<tr>
<td>Cry</td>
<td></td>
</tr>
<tr>
<td>Der</td>
<td>-1.6</td>
</tr>
<tr>
<td>Mpg</td>
<td></td>
</tr>
<tr>
<td>Smk</td>
<td></td>
</tr>
<tr>
<td>Ser</td>
<td></td>
</tr>
<tr>
<td>Sfl</td>
<td></td>
</tr>
<tr>
<td>Xml</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

Performance compared to STW Shenandoah with all barriers disabled.
Read Barriers: Observations

1. Shenandoah needs RBs before most loads
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains
Read Barriers: Observations

1. Shenandoah needs RBs before **most** loads
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains

2. Passive RB cost is moderate
   - Dependent load that hits the same cache line as object
Read Barriers: Observations

1. Shenandoah needs RBs before most loads
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains

2. Passive RB cost is moderate
   - Dependent load that hits the same cache line as object

3. Active RB cost is moderate
   - Does not differ much from passive RB
What if we compare from-copy and to-copy *themselves*?

\[(a_1 \equiv a_2) \rightarrow ???\]
CMP: Trouble

What if we compare from-copy and to-copy *themselves*?

\[(a_1 == a_2) \rightarrow ???\]

But *machine ptrs* are not equal... Oops.
Having two \textit{physical} copies of the same \textit{logical} object, 
\texttt{«==»} has to compare \textit{logical} objects

\begin{verbatim}
# compare the ptrs; if equal, good!
cmp %rcx,%rdx    # if (a1 == a2) ...
je EQUALS

# false negative? have to compare to-copy:
mov -0x8(%rcx),%rcx # a1 = *(a1 - 8)
mov -0x8(%rdx),%rdx # a2 = *(a2 - 8)

# compare again:
cmp %rcx,%rdx    # if (a1 == a2) ...
\end{verbatim}
# CMP: Barriers Cost

<table>
<thead>
<tr>
<th></th>
<th>Throughput hit, %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SATB</td>
</tr>
<tr>
<td>Cmp</td>
<td>-1.6</td>
</tr>
<tr>
<td>Cps</td>
<td>-3.5</td>
</tr>
<tr>
<td>Cry</td>
<td></td>
</tr>
<tr>
<td>Der</td>
<td>-1.6</td>
</tr>
<tr>
<td>Mpg</td>
<td>-2.1</td>
</tr>
<tr>
<td>Smk</td>
<td>-0.5</td>
</tr>
<tr>
<td>Ser</td>
<td>-4.0</td>
</tr>
<tr>
<td>Sfl</td>
<td>-2.7</td>
</tr>
<tr>
<td>Xml</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

\(^2\text{Performance compared to STW Shenandoah with all barriers disabled}\)
1. Shenandoah needs to handle ref comparisons specially
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains
CMP: Observations

1. Shenandoah needs to handle ref comparisons specially
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains

2. Passive CMP cost is low
   - Barely detectable in most cases
   - Comparisons with `null` are frequent and optimized
CMP: Observations

1. Shenandoah needs to handle ref comparisons specially
- Cannot make RBs much heavier
- Optimizing compilers move and coalesce RB – massive gains

2. Passive CMP cost is low
- Barely detectable in most cases
- Comparisons with `null` are frequent and optimized

3. Active CMP cost is low
- Does not differ much from passive RB
## Overall: Barriers Cost

<table>
<thead>
<tr>
<th></th>
<th>SATB</th>
<th>WB</th>
<th>RB</th>
<th>CMP*</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmp</td>
<td>-1.6</td>
<td>-3.5</td>
<td>-7.7</td>
<td>-14.3</td>
<td></td>
</tr>
<tr>
<td>Cps</td>
<td>-3.5</td>
<td></td>
<td>-11.4</td>
<td>-13.7</td>
<td></td>
</tr>
<tr>
<td>Cry</td>
<td></td>
<td>-1.1</td>
<td></td>
<td>-4.3</td>
<td></td>
</tr>
<tr>
<td>Der</td>
<td>-1.6</td>
<td></td>
<td>-7.4</td>
<td>-9.3</td>
<td></td>
</tr>
<tr>
<td>Mpg</td>
<td></td>
<td>-2.1</td>
<td>-12.4</td>
<td>-14.8</td>
<td></td>
</tr>
<tr>
<td>Smk</td>
<td>-0.5</td>
<td></td>
<td>-4.9</td>
<td>-2.6</td>
<td></td>
</tr>
<tr>
<td>Ser</td>
<td>-4.0</td>
<td></td>
<td>-7.1</td>
<td>-11.1</td>
<td></td>
</tr>
<tr>
<td>Sfl</td>
<td>-2.7</td>
<td></td>
<td>-6.7</td>
<td>-11.3</td>
<td></td>
</tr>
<tr>
<td>Xml</td>
<td>-3.1</td>
<td>-3.5</td>
<td>-9.5</td>
<td>-15.6</td>
<td></td>
</tr>
</tbody>
</table>

\(^2\) Performance compared to STW Shenandoah with all barriers disabled
Overall: Observations

1. Easily portable across HW architectures
   - Special needs: CAS (performance is important, but not critical)
   - x86_64 and AArch64 are major implemented targets
   - Theoretically works with 32-bit arches (but not ported yet)
Overall: Observations

1. Easily portable across HW architectures
   - Special needs: CAS (performance is important, but not critical)
   - x86_64 and AArch64 are major implemented targets
   - Theoretically works with 32-bit arches (but not ported yet)

2. Trivially portable across OSes
   - Special needs: none
   - Linux is a major target, Windows is minor target
   - Adopters build on Mac OS without problems
Overall: Observations

1. Easily portable across HW architectures
   - Special needs: CAS (performance is important, but not critical)
   - x86_64 and AArch64 are major implemented targets
   - Theoretically works with 32-bit arches (but not ported yet)

2. Trivially portable across OSes
   - Special needs: none
   - Linux is a major target, Windows is minor target
   - Adopters build on Mac OS without problems

3. VM interactions are simple enough
   - Play well with compressed oops: separate fwdptr
   - OS/CPU-specific things only for barriers codegen
Intermezzo
Intermezzo: Generational Hypotheses

Weak hypothesis: most objects die young
Intermezzo: Generational Hypotheses

**Strong hypothesis:**
the older the object,
the less chance it has to die
Intermezzo: Generational Hypotheses

**Strong hypothesis:**
the older the object, the less chance it has to die

In-memory LRU-like caches are the prime counterexamples
Intermezzo: LRU, Pesky Workload

Very inconvenient workload for *simple* generational GCs

- Early on, many young objects die, and oldies survive: *weak GH is valid, strong GH is valid*
- Suddenly, old objects start to die: *weak GH is valid, strong GH is not valid anymore!*
- Naive GCs trip over and burn
Intermezzo: The Simplest LRU

The simplest LRU implementation in Java?
Intermezzo: The Simplest LRU

The simplest LRU implementation in Java?

cache = new LinkedHashMap<>(size*4/3, 0.75f, true) {
    @Override
    protected boolean removeEldestEntry(Map.Entry<> eldest) {
        return size() > size;
    }
};
Intermezzo: Testing

Boring config:
1. Latest improvements in all GCs: shenandoah/jdk forest
2. Decent multithreading: 8 threads on 16-thread i7-7820X
3. Larger heap: -Xmx100g -Xms100g
4. 90% hit rate, 90% reads, 10% writes
5. Size (LDS) = 0..100% of -Xmx

Varying cache size ⇒ varying LDS ⇒ make GC uncomfortable
Intermezzo: Pauses vs. LDS

The graph compares pause time (in seconds) against live data size (% of heap) for three different garbage collection algorithms: Parallel, CMS, and Shenandoah. The x-axis represents the live data size as a percentage of the heap, while the y-axis shows the pause time in seconds. The data is represented on a log-log scale, indicating a power-law relationship between live data size and pause time.
Intermezzo: Pauses vs. LDS
Intermezzo: Pauses vs. LDS

<table>
<thead>
<tr>
<th>Parallel</th>
<th>CMS</th>
<th>Shenandoah</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
</tbody>
</table>

- **Parallel**
  - No STW
  - Young GC

- **CMS**
  - Live Data Size, % of heap

- **Shenandoah**
  - No STW
  - Young GC

*Slide 52/80. «Shenandoah GC», Aleksey Shipilëv, 2018, D:20180914113310+02'00'
Intermezzo: Pauses vs. LDS

- **Parallel CMS**
- **CMS**
- **Shenandoah**

![Graph showing pause time vs. live data size for different garbage collectors.](image-url)

**Heap Overload**

**Pause time, sec (all safepoints)**

**Live Data Size, % of heap**

**Heap Overload**

**Parallel CMS Shenandoah**

**Graphs**

- **Parallel CMS**
- **CMS**
- **Shenandoah**

**X-axis**: Live Data Size, % of heap

**Y-axis**: Pause time, sec (all safepoints)

**Legend**

- **Parallel CMS**
- **CMS**
- **Shenandoah**
Intermezzo: Perf vs. LDS

Slide 53/80. "Shenandoah GC", Aleksey Shipilëv, 2018, D:20180914113310+02'00"
Intermezzo: Perf vs. LDS

Operation Time, sec

Live Data Size, % of heap

GC Pause Time, %

GC work happens in background

Live Data Size, % of heap

gc  Parallel  CMS  Shenandoah

Parallel CMS Shenandoah
Intermezzo: Perf vs. LDS

...and application appears faster!

GC work happens in background
Command and Control
Concurrent GCs are in-background heavy-lifters

- Rely on collecting **faster** than applications allocate
- *Frequently* works by itself: threads do useful work, GC threads are high-priority, there is enough heap to absorb allocations
- *Practical* concurrent GCs have to care about unfortunate cases as well
Command and Control: Off To The Races

[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is above the time for allocation rate (3254.90 MB/s) to deplete free headroom (13071M)

Want better conc GC performance, less frequent GC cycles?

- **GC Time.** Get more GC threads, have coarser objects, etc
- **Allocation Rate.** Get easy on excessive allocations
- **Heap Size.** Give concurrent GC more heap to play with
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is above the time for allocation rate (3254.90 MB/s) to deplete free headroom (13071M)

Want better conc GC performance, less frequent GC cycles?

- **GC Time.** Get more GC threads, have coarser objects, etc
- **Allocation Rate.** Get easy on excessive allocations
- **Heap Size.** Give concurrent GC more heap to play with
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is above the time for allocation rate (3254.90 MB/s) to deplete free headroom (13071M)

Want better conc GC performance, less frequent GC cycles?

- **GC Time.** Get more GC threads, have coarser objects, etc
- **Allocation Rate.** Get easy on excessive allocations
- **Heap Size.** Give concurrent GC more heap to play with
Command and Control: Off To The Races

[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is above the time for allocation rate (3254.90 MB/s) to deplete free headroom (13071M)

Want better conc GC performance, less frequent GC cycles?

- **GC Time.** Get more GC threads, have coarser objects, etc
- **Allocation Rate.** Get easy on excessive allocations
- **Heap Size.** Give concurrent GC more heap to play with
Command and Control: Living Space

Problem:
Concurrent GC needs breathing room to succeed, while applications allocate like madmen.

Things that help:
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade
Immediates: Living Space

**Problem:**
Concurrent GC needs breathing room to succeed, while applications allocate like madmen

Things that help:
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade
Immediates: Obvious Shortcut

GC(7) Pause Init Mark 0.614ms
GC(7) Concurrent marking 76812M–>76864M(102400M) 1.650ms
GC(7) Total Garbage: 76798M
GC(7) Immediate Garbage: 75072M, 2346 regions (97% of total)
GC(7) Pause Final Mark 0.758ms
GC(7) Concurrent cleanup 76864M–>1844M(102400M) 3.346ms
Immediates: Obvious Shortcut

GC(7) Pause Init Mark 0.614ms
GC(7) Concurrent marking 76812M→76864M(102400M) 1.650ms
GC(7) Total Garbage: 76798M
GC(7) Immediate Garbage: 75072M, 2346 regions (97% of total)
GC(7) Pause Final Mark 0.758ms
GC(7) Concurrent cleanup 76864M→1844M(102400M) 3.346ms

1. Mark is fast, because most things are dead
Immediates: Obvious Shortcut

GC(7) Pause Init Mark 0.614ms
GC(7) Concurrent marking 76812M->76864M(102400M) 1.650ms
GC(7) Total Garbage: 76798M
GC(7) Immediate Garbage: 75072M, 2346 regions (97% of total)
GC(7) Pause Final Mark 0.758ms
GC(7) Concurrent cleanup 76864M->1844M(102400M) 3.346ms

1. Mark is fast, because most things are dead
2. Lots of fully dead regions, because most objects are dead
Immediates: Obvious Shortcut

GC(7) Pause Init Mark 0.614ms
GC(7) Concurrent marking 76812M→76864M(102400M) 1.650ms
GC(7) Total Garbage: 76798M
GC(7) Immediate Garbage: 75072M, 2346 regions (97% of total)
GC(7) Pause Final Mark 0.758ms
GC(7) Concurrent cleanup 76864M→1844M(102400M) 3.346ms

1. Mark is fast, because most things are dead
2. Lots of fully dead regions, because most objects are dead
3. Cycle shortcuts, because why bother...
Footprint: Living Space

**Problem:**
Concurrent GC needs breathing room to succeed, while applications allocate like madmen

**Things that help:**
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade
Footprint: Shenandoah Overheads

Shenandoah requires additional word per object for forwarding pointer at all times, plus some native structs

- **Java heap**: 1.5x worst and 1.05-1.10x avg overhead
  - «—»: the overhead is non-static
  - «+»: counted in Java heap – no surprise RSS inflation

- **Native structures**: 2x marking bitmaps, each 1/64 of heap
  - «—»: -Xmx is still not close to RSS
  - «+»: overhead is static: -Xmx100g means 103 GB RSS
Footprint: Shenandoah Overheads

Shenandoah requires additional word per object for forwarding pointer at all times, plus some native structs

- **Java heap**: 1.5x worst and 1.05-1.10x avg overhead
  
  «−»: the overhead is non-static
  «+»: counted in Java heap – no surprise RSS inflation

- **Native structures**: 2x marking bitmaps, each 1/64 of heap
  
  «−»: -Xmx is still not close to RSS
  «+»: overhead is static: -Xmx100g means 103 GB RSS

- **Surprise**: a significant part of footprint story is heap sizing, not per-object or per-heap overheads
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

G1
Sh
Sh (compact)
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

- G1
- Sh
- Sh (compact)
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

- G1
- Sh
- Sh (compact)
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

G1
Sh
Sh (compact)
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

G1
Sh
Sh (compact)
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

Aggressive expansion
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

Periodic GC

RSS, MB

Start Idle Load Idle Full GC Idle

time, sec

G1
Sh
Sh (compact)
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

RSS, MB
time, sec

G1
Sh
Sh (compact)

Second uncommit
Footprint: Heap Sizing

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

G1
Sh
Sh (compact)

Very frequent GCs
Footprint: CPU Time Tradeoffs

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

Java user CPU, %
time, sec
G1
Sh
Sh (compact)
Footprint: CPU Time Tradeoffs

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

Java user CPU, %
time, sec
G1
Sh
Sh (compact)
Footprint: CPU Time Tradeoffs

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

High footprint, low CPU

Java user CPU, %
time, sec

G1
Sh
Sh (compact)
Footprint: CPU Time Tradeoffs

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

Low footprint, high CPU
Footprint: CPU Time Tradeoffs

wildfly-swarm-rest-http, 30K rps, JDK head x86-64, -Xmx512m

Low footprint, low CPU

- G1
- Sh
- Sh (compact)
Footprint: Observations

1. Footprint story is nuanced
   - Blindly counting bytes taken by Java heap and GC does not cut it
   - First-order effect: heap sizing policies
   - Second-order effects: per-object and per-reference overheads

2. Forwarding pointer overhead is substantial, but manageable, especially when the alternative is giving up compressed opcodes.

3. Deleting footprints seems to be of most interest. Few adopters (none?) care about peak footprint, but we still do.
Footprint: Observations

1. Footprint story is nuanced
   - Blindly counting bytes taken by Java heap and GC does not cut it
   - First-order effect: heap sizing policies
   - Second-order effects: per-object and per-reference overheads

2. Forwarding ptr overhead is substantial, but manageable
   - ...especially when the alternative is giving up compressed oops
   - In-object fwdptr injection cuts the overhead down (see backup)
Footprint: Observations

1. Footprint story is nuanced
   - Blindly counting bytes taken by Java heap and GC does not cut it
   - First-order effect: heap sizing policies
   - Second-order effects: per-object and per-reference overheads

2. Forwarding ptr overhead is substantial, but manageable
   - ...especially when the alternative is giving up compressed oops
   - In-object fwdptr injection cuts the overhead down (see backup)

3. Idle footprint seems to be of most interest
   - Few adopters (none?) care about peak footprint, but we still do
   - Anecdote: I am running Shenandoah with my IDEA and CLion, because memory is scarce on my puny ultrabook
Pacing: Living Space

Problem:
Concurrent GC needs breathing room to succeed, while applications allocate like madmen

Things that help:
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade
Once memory is exhausted, perform GC
- Natural feedback loop: STW is the nominal mode
- Not really accessible for concurrent GC?
Pacing: Naive Conc GC Control Loop

- Memory is exhausted ⇒ stall allocation and wait for GC
- Technically not a GC pause, but still *local latency*
- AFs usually happen in all threads at once: *global latency*
Incremental pacing stalls allocations a bit at a time.

If AF happens, “degenerates”: completes under STW.

Pacing introduces latency, but the capped one.
Pacing: Max Pacing, Pauses

Shenandoah

Shenandoah (max pacing)

Live Data Size, % of heap

Pause time, sec (all safepoints)

10^{-4}

10^{-3}

10^{-2}

10^{-1}

10^{0}

10^{1}

0 20 40 60 80 100

0 20 40 60 80 100

0 20 40 60 80 100

0 20 40 60 80 100

Slide 69/80. «Shenandoah GC», Aleksey Shipilëv, 2018, D:20180914113310+02'00'
Pacing: Max Pacing, Pauses

Nuclear option: max pacing
Pacing: Max Pacing, Times

- Operation Time, sec
- GC Pause Time, %

Live Data Size, % of heap

gc Shenandoah Shenandoah (max pacing)
Pacing: Max Pacing, Times

Operation Time, sec

GC Pause Time, %

Pauses are invisible

Live Data Size, % of heap

Shenandoah

Shenandoah (max pacing)

gc

Shenandoah

Shenandoah (max pacing)
Pacing: Max Pacing, Times

Yet the progress is wrecked anyway

Pauses are invisible

Live Data Size, % of heap

Operation Time, sec

Live Data Size, % of heap

GC Pause Time, %

Shenandoah Shenandoah (max pacing)
Pacing: Observations

1. Pacing provides essential negative feedback loop
   - Thread allocates? Thread pays for it!
   - Thread does not allocate as much? It can run freely!
Pacing: Observations

1. Pacing provides essential negative feedback loop
   - Thread allocates? Thread pays for it!
   - Thread does not allocate as much? It can run freely!

2. Pacing introduces local latency
   - Hidden from the tools, hidden from usual GC log
   - Latency is not global, making perf analysis harder
Pacing: Observations

1. Pacing provides essential negative feedback loop
   - Thread allocates? Thread pays for it!
   - Thread does not allocate as much? It can run freely!

2. Pacing introduces local latency
   - Hidden from the tools, hidden from usual GC log
   - Latency is not global, making perf analysis harder

3. Nuclear option: max pacing delay = $+\infty$
   - Resolves the need for handling allocation failures: thread always stalls when memory is not available
   - Shenandoah caps delay at 10 ms to avoid cheating
Handling Failures: Living Space

**Problem:**
Concurrent GC needs breathing room to succeed, while applications allocate like madmen

**Things that help:**
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade
Handling Failures: Shenandoah Control Loop

If AF happens, «degenerates»: completes under STW
Handling Failures: Degenerated GC

Pause Init Update Refs 0.034ms
Cancelling GC: Allocation Failure
Concurrent update references 7265M→8126M(8192M) 248.467ms
Pause Degenerated GC (Update Refs) 8126M→2716M(8192M) 29.787ms

- First allocation failure dives into stop-the-world mode
- Degenerated GC *continues* the cycle
- Second allocation failure may upgrade to Full GC
Handling Failures: Degenerated GC

Pause Init Update Refs 0.034ms
Cancelling GC: Allocation Failure
Concurrent update references 7265M→8126M(8192M) 248.467ms
Pause Degenerated GC (Update Refs) 8126M→2716M(8192M) 29.787ms

- First allocation failure dives into stop-the-world mode
- Degenerated GC continues the cycle
- Second allocation failure may upgrade to Full GC
Handling Failures: Full GC

Full GC is the Maximum Credible Accident: Parallel, STW, Sliding «Lisp 2»-style GC.

- Designed to recover from anything: 99% full regions, heavy (humongous) fragmentation, abort from any point in concurrent GC, etc.
- Parallel: Multi-threaded, runs on-par with Parallel GC
- Sliding: No additional memory needed + reuses fwdptr slots to store forwarding data
Handling Failures: Observations

1. Being fully concurrent is nice, but own the failures
   - The failures will happen, accept it
   - «Our perfect GC melted down, because you forgot this magic VM option(, stupid)» flies only that far

2. Graceful and observable degradation is key
   - Getting worse incrementally is better than falling off the cliff

3. Failure paths and performance is important
   - Degraded GC is not throwing away progress
   - Full GC is optimized to do so
Handling Failures: Observations

1. Being fully concurrent is nice, but own the failures
   - The failures will happen, accept it
   - «Our perfect GC melted down, because you forgot this magic VM option, stupid» flies only that far

2. Graceful and observable degradation is key
   - Getting worse incrementally is better than falling off the cliff
   - Have enough logging to diagnose the degradations
Handling Failures: Observations

1. Being fully concurrent is nice, but own the failures
   - The failures will happen, accept it
   - «Our perfect GC melted down, because you forgot this magic VM option, stupid» flies only that far

2. Graceful and observable degradation is key
   - Getting worse incrementally is better than falling off the cliff
   - Have enough logging to diagnose the degradations

3. Failure paths performance is important
   - Degenerated GC is not throwing away progress
   - Full GC is optimized too
Conclusion
Conclusion: In Single Picture

Universal GC does not exist:
either low latency, or high throughput
(, or low memory footprint)

Choose this for your workload!
Conclusion: In Single Paragraph

1. No GC could detect what tradeoffs you are after: you have to tell it yourself
Conclusion: In Single Paragraph

1. No GC could detect what tradeoffs you are after: you have to tell it yourself

2. Stop-the-world GCs beat concurrent GCs in throughput and efficiency. **Parallel GC** is your choice!
Conclusion: In Single Paragraph

1. No GC could detect what tradeoffs you are after: you have to tell it yourself

2. Stop-the-world GCs beat concurrent GCs in throughput and efficiency. **Parallel GC** is your choice!

3. Concurrent Mark trims down the pauses significantly. **G1** is ready for this, use it!
Conclusion: In Single Paragraph

1. No GC could detect what tradeoffs you are after: you have to tell it yourself

2. Stop-the-world GCs beat concurrent GCs in throughput and efficiency. **Parallel GC** is your choice!

3. Concurrent Mark trims down the pauses significantly. **G1** is ready for this, use it!

4. Concurrent Copy/Compact needs to be addressed for even shallower pauses. This is where **Shenandoah** and **ZGC** come in!
Conclusion: Releases

Easy to access (development) releases: try it now!
https://wiki.openjdk.java.net/display/shenandoah/

- Dev follows latest JDK, backports to 11, 10, and 8
- JDK 8 backport ships in RHEL 7.4+, Fedora 24+
- JDK 11 backport ships in Fedora 27+
- Nightly development builds (tarballs, Docker images)

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