Shenandoah GC
Part I: The Garbage Collector That Could

Aleksey Shipilëv
shade@redhat.com
@shipilev
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.
IMHO, discussing GC without having first read the «GC Handbook» is a waste of time, and regurgitating known stuff

It may appear that \$name GC is a super-duper-innovative, but in fact many GCs reuse (or reinvent) ideas from that textbook
Overview
Overview: Landscape

Serial, Parallel:

- Copy
- Mark
- Compact

Young GC → Old GC
Overview: Landscape

Serial, Parallel:
- Copy
- Mark
- Compact

CMS:
- Copy
- Concurrent Mark
- Conc. Sweep

Young GC  —  Old GC

Still a pause :(  —  Does not solve fragmentation :(  
Init Mark  —  Finish Mark
Overview: Landscape

Young GC  Old GC

Serial, Parallel:
- Copy
- Mark
- Compact

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

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

Does not solve fragmentation :(  Smaller, adjustable, but still a pause :(  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:
- Conc. Partial
- Concurrent Mark
- Conc. Compact
  - Init Mark
  - Finish Mark

Still a pause :

Does not solve fragmentation :

Smaller, adjustable, but still a pause :

Smaller, adjustable, but still a pause :(
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: Cycle

Three major phases:
Overview: Cycle

Three major phases:
1. Snapshot-at-the-beginning concurrent mark
Overview: Cycle

Three major phases:
1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
Overview: Cycle

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

Three major phases:
1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
3. Concurrent update references (optional)
Overview: Usual Log

LRUFragger, 100 GB heap, ≈ 80 GB LDS:

Pause Init Mark 0.437ms
Concurrent marking 76780M→77260M(102400M) 700.185ms
Pause Final Mark 0.698ms
Concurrent cleanup 77288M→77296M(102400M) 3.176ms
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 0.351ms
Concurrent cleanup 85928M→56620M(102400M) 14.316ms
Overview: Usual Log

LRUFragger, 100 GB heap, ≈ 80 GB LDS:

Pause Init Mark 0.437ms
Concurrent marking 76780M→77260M(102400M) 700.185ms
Pause Final Mark 0.698ms
Concurrent cleanup 77288M→77296M(102400M) 3.176ms
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 0.351ms
Concurrent cleanup 85928M→56620M(102400M) 14.316ms
Basics
Concurrent Mark: Reachability

To catch a garbage, you have to think like a garbage -
know if there are references to the object
Concurrent 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, and treat everything as reachable *(see Epsilon GC)*
Concurrent 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, and treat everything as reachable *(see Epsilon GC)*

2. **Mark-***: walk the object graph, find reachable objects, treat everything else as garbage
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, and treat everything as reachable *(see Epsilon GC)*

2. **Mark-***: walk the object graph, find reachable objects, treat everything else as garbage

3. **Reference counting**: count the number of references, and when refcount drops to 0, treat the object as garbage
Concurrent 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
Concurrent 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»
Concurrent Mark: Stop-The-World Mark

When application is stopped, everything is trivial!
Nothing messes up the scan...
Concurrent Mark: Stop-The-World Mark

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

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

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

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

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

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

Gray $\rightarrow$ Black;
reachable from Gray $\rightarrow$ Gray
Concurrent 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 *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.
Concurrent Mark: SATB

Color all \textbf{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, Fastpath

# read TLS flag
movsbl 0x378(%r15),%r10  # flag = *(TLS + 0x378)

# if that flag is up...
test %r10,%r10  # if (flag) ...
jne OMG-SATB-ENABLED

# perform the actual store to %r12 and offset 0x42
mov %r11,0x42(%r12)  # *(obj + 0x42) = r11
OMG-SATB-ENABLED:

# read the old value from the field
mov 0x2c(%rbp),%r10d  # oldval = *(obj + 0x2c)

# take the the head of thread-local buffer
mov 0x388(%r15),%r11  # qhead = *(TLS + 0x388)

# then tens of instructions that add old value
# to local buffer, check for overflow, call into
# VM slowpath to process the thread-local buffer, etc.
Concurrent Mark: Two Pauses

Init Mark:
1. Stop the mutator to avoid races
2. Color the rootset Black
3. Arm SATB barriers

Final Mark:
1. Stop the mutator to avoid races
2. Drain the SATB buffers
3. Finish work from SATB updates
Concurrent Mark: Two Pauses

Init Mark:
1. Stop the mutator to avoid races
2. Color the rootset Black ← most heavy-weight
3. Arm SATB barriers

Final Mark:
1. Stop the mutator to avoid races
2. Drain the SATB buffers
3. Finish work from SATB updates ← most heavy-weight
## Concurrent Mark: Barriers Cost

<table>
<thead>
<tr>
<th>Throughput hit, %</th>
<th>SATB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmp</td>
<td>-2.8</td>
</tr>
<tr>
<td>Cps</td>
<td></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>-2.6</td>
</tr>
</tbody>
</table>

1 Performance compared to STW Shenandoah with all barriers disabled
Concurrent Mark: Observations

1. Throughput-wise, well engineered STW GC would beat well engineered concurrent GC

**Translation:** If you don’t care about GC pauses, just use good STW GC
Concurrent Mark: Observations

1. Throughput-wise, well engineered STW GC would beat well engineered concurrent GC
   
   **Translation:** If you don’t care about GC pauses, just use good STW GC

2. Barrier costs are there even without GC cycles happening
   
   **Translation:** Running the application that causes no GC cycles? Less sophisticated GC gives less overheads
Concurrent Copy: Stop-The-World

Problem:
there is the object, the object is referenced from somewhere, need to move it to new location
Concurrent Copy: Stop-The-World

**Step 1:** Stop The World, evasive maneuver to distract mutator from looking into our mess

"From" space

<table>
<thead>
<tr>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
</tr>
<tr>
<td>y = 2</td>
</tr>
<tr>
<td>z = 3</td>
</tr>
</tbody>
</table>

"To" space
Concurrent Copy: Stop-The-World

Step 2:
Copy the object with all its contents

Slide 21/75. «Shenandoah GC», Aleksey Shipilëv, 2017, D:20171121102551+01'00'
Concurrent Copy: Stop-The-World

Step 3.1: Update all references: save the pointer that forwards to the copy
Concurrent Copy: Stop-The-World

**Step 3.2:**
Update all references: walk the heap, replace all refs with fwdptr destination

Slide 21/75. «Shenandoah GC», Aleksey Shipilëv, 2017, D:20171121102551+01'00'
Concurrent Copy: Stop-The-World

**Step 3.2:**
Update all references: walk the heap, replace all refs with fwdptr destination
Concurrent Copy: Stop-The-World

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

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: Java Analogy

class VersionUpdater<T, V> {
    final AtomicReference<T> ref = ...;

    void writeValue(V value) {
        do {
            T oldObj = ref.get();
            T newObj = copy(oldObj);
            newObj.set(value);
        } while (!ref.compareAndSet(oldObj, newObj));
    }
}

Everyone wrote this thing about a hundred times...
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

"From" space

"To" space

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

Done!
Write Barriers: Motivation

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

# read the thread-local flag
movzbl 0x3d8(%r15),%r11d  # flag = *(TLS + 0x3d8)

# if that flag is set, then...
test %r11d,%r11d  # if (flag) ...
jne OMG-EVAC-ENABLED

# make sure we have the to-copy
mov -0x8(%rbp),%r10  # obj = *(obj - 8)

# store into to-copy r10 at offset 0x30
mov %r10,0x30(%r10)  # *(obj + 0x30) = r10
stub Write(val, obj, offset) {
    if (evac-in-progress && in-collection-set(obj) && fwd-ptrs-to-self(obj)) { // no copy yet
        val copy = copy(obj);
        *(copy + offset) = val; // actual write
        if (CAS(fwd-ptr-addr(obj), obj, copy)) {
            return; // success!
        }
    }

    obj = fwd-ptr(obj); // write to actual copy
    *(obj + offset) = val; // actual write
}
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>-2.8</td>
</tr>
<tr>
<td>Cps</td>
<td>-1.5</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>-2.6</td>
</tr>
</tbody>
</table>

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

1. Shenandoah needs WB on all stores

   **Translation:** Field stores, locking the object, computing the identity hash code the first time, etc – all require write barriers
Write Barriers: Observations

1. Shenandoah needs WB on all stores
   **Translation**: Field stores, locking the object, computing the identity hash code the first time, etc – all require write barriers

2. Application steps on WB slowpath very rarely: only during evacuation phase, on a few evacuated objects, on those objects that were not yet visited by GC
   **Translation**: In practice, WBs have low overhead
Read Barriers: Motivation

Heap reads have to (?) dereference via the forwarding pointer, to discover the actual object copy.
Read Barriers: Implementation

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

# heap read!
mov   0x30(%r10),%r10d    # val = *(obj + 0x30)
```
# Read Barriers: Implementation

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

# heap read!
mov    0x30(%r10),%r10d          # val = *(obj + 0x30)
```

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

<table>
<thead>
<tr>
<th>Throughput hit, %</th>
<th>SATB</th>
<th>WB</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmp</td>
<td>-2.8</td>
<td>-2.9</td>
<td>-9.8</td>
</tr>
<tr>
<td>Cps</td>
<td></td>
<td>-1.5</td>
<td>-11.6</td>
</tr>
<tr>
<td>Cry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Der</td>
<td>-1.6</td>
<td>-2.5</td>
<td>-8.9</td>
</tr>
<tr>
<td>Mpg</td>
<td></td>
<td>-9.9</td>
<td>-10.9</td>
</tr>
<tr>
<td>Smk</td>
<td>-1.7</td>
<td>-0.7</td>
<td></td>
</tr>
<tr>
<td>Ser</td>
<td>-2.6</td>
<td>-9.4</td>
<td></td>
</tr>
<tr>
<td>Sfl</td>
<td>-12.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xml</td>
<td>-2.6</td>
<td>-2.8</td>
<td>-13.7</td>
</tr>
</tbody>
</table>

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

1. RBs are cheap, but there are lots of them

Translation: cannot make RBs much heavier

---

2Use tagged/colored pointers seems odd because of this
Read Barriers: Observations

1. RBs are cheap, but there are lots of them
   **Translation:** cannot make RBs much heavier

2. The observed overhead depends heavily on optimizers ability to eliminate, hoist and coalesce barriers
   **Translation:** high-performance GC development assumes optimizing compiler work

---

2Use tagged/colored pointers seems odd because of this
CMP: Trouble

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

\[(a_1 \equiv a_2) \rightarrow ???\]
What if we compare from-copy and to-copy *themselves*?

\[(a1 == a2) \rightarrow ???\]

But *machine ptrs* are not equal... Oops.
CMP: Exotic Barriers

Having two *physical* copies of the same *logical* object, «==» has to compare *logical* objects

```c
# 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) ...
```
## 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>-2.8</td>
</tr>
<tr>
<td>Cps</td>
<td></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>-1.7</td>
</tr>
<tr>
<td>Ser</td>
<td>-2.6</td>
</tr>
<tr>
<td>Sfl</td>
<td></td>
</tr>
<tr>
<td>Xml</td>
<td>-2.6</td>
</tr>
</tbody>
</table>

1 Performance compared to STW Shenandoah with all barriers disabled
CMP: Observations

1. Full-fledged «==» reference comparisons are rare, and special kinds of comparisons are well-optimized

**Translation:** cmp barriers are not affecting much, a == null does not require barriers, etc.
**CMP: Observations**

1. Full-fledged «==» reference comparisons are rare, and special kinds of comparisons are well-optimized

   **Translation:** cmp barriers are not affecting much, `a == null` does not require barriers, etc.

2. There is also the problem with reference CASes, but the failure there is also rare

   **Translation:** if CAS had failed, you have much larger performance problems...
# Overall: 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>-2.8</td>
</tr>
<tr>
<td>Cps</td>
<td></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>-1.7</td>
</tr>
<tr>
<td>Ser</td>
<td>-2.6</td>
</tr>
<tr>
<td>Sfl</td>
<td></td>
</tr>
<tr>
<td>Xml</td>
<td>-2.6</td>
</tr>
</tbody>
</table>

↑ Performance compared to STW Shenandoah with all barriers disabled
Overall: Observations

1. Shenandoah barriers do not require special hardware or special OS support!

**Translation:** No need for kernel patches, pricey hardware, vendor lock-in distros, etc.
1. Shenandoah barriers **do not** require special hardware or special OS support!

**Translation:** No need for kernel patches, pricey hardware, vendor lock-in distros, etc

2. The throughput hit is mostly acceptable, taking note the latency improvements achieved

**Translation:** Latency-throughput tradeoff is here. Do not need low latency? Use STW GC.
Intermezzo
Intermezzo: Generational Hypotheses, Weak

Weak hypothesis: most objects die young
Intermezzo: Generational Hypothesis, Strong

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

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 (those that follow weak GH, and trust in strong GH)

1. Appears to be weak GH workload in the beginning
2. As cache population grows, Live Data Set (LDS) grows too. LDS is measured in gigabytes – it is a cache, after all
3. As cache gets full, old objects start to die, violating strong GH, much to naive GC surprise
4. GC heuristics trips over and burns
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/jdk10 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
Intermezzo: Pauses vs. LDS

The diagram shows the relationship between the pause time (in seconds) and the live data size (% of heap) for different garbage collection (GC) algorithms. The algorithms compared are G1, Parallel, CMS, and Shenandoah.

- **G1**: Shows a gradual increase in pause time with an increase in live data size.
- **Parallel**: Demonstrates a sudden increase in pause time at a certain live data size.
- **CMS**: Exhibits a fluctuation in pause time with a peak at a specific live data size.
- **Shenandoah**: Displays a consistent pattern with no significant change in pause time.

A notable feature is the absence of Stop-The-World (STW) pauses in the Shenandoah algorithm, indicated by the label "No STW Old GC."
Intermezzo: Pauses vs. LDS

- **G1**
- **Parallel**
- **CMS**
- **Shenandoah**

---

- **Pause time, sec (all safepoints):**
  - **G1:**
    - Live Data Size, % of heap:
      - **Pause time, sec (all safepoints):**
        - **No STW**
        - **Young GC**

---

Slide 48/75. «Shenandoah GC», Aleksey Shipilëv, 2017, D:20171121102551+01’00’
Intermezzo: Pauses vs. LDS

Too small Java heap
Intermezzo: Perf vs. LDS

**GC Pause Time, %**

- gc
- G1
- Parallel
- CMS
- Shenandoah

**Operation Time, sec**

- gc
- G1
- Parallel
- CMS
- Shenandoah
Advanced
Advanced: Major Assumption

Concurrent 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...

- But you have to also take care about unhappy paths!
Advanced: Living Space

**Problem:**
Concurrent GC needs breathing room to succeed

Things that help:
- Aggressive heap expansion: prefer taking more memory
- Immediate garbage shortcuts: free memory early
- Partial collections: collect easy parts of heap first
- Mutator pacing: stall allocators before they hit the wall
Footprint: Living Space

Problem:
Concurrent GC needs breathing room to succeed

Things that help:
- Aggressive heap expansion: prefer taking more memory
- Immediate garbage shortcuts: free memory early
- Partial collections: collect easy parts of heap first
- Mutator pacing: stall allocators before they hit the wall
Footprint: Internals

Usual **active** footprint overhead: 3..15% of heap size

1. Java heap: forwarding pointer (8 bytes/object)
2. Native: 2 marking bitmaps (1/64 bits per heap bit)
3. Native: $N_{CPU}$ workers ($\approx$ 2 MB / GC thread)
4. Native: region data ($\approx$ 1 KB per region)
Footprint: Internals

Usual **active** footprint overhead: 3..15% of heap size

1. Java heap: forwarding pointer (8 bytes/object)
2. Native: 2 marking bitmaps (1/64 bits per heap bit)
3. Native: $N_{CPU}$ workers ($\approx$ 2 MB / GC thread)
4. Native: region data ($\approx$ 1 KB per region)

Example: `-XX:+UseShenandoahGC -Xmx100G` means:

$\approx$ 90..95 GB accessible for Java objects,
$\approx$ 103 GB RSS for GC parts
Footprint: Internals

Usual active footprint overhead: 3..15% of heap size

But all of that is totally dwarfed by GC heap sizing policies

Example: -XX:+UseShenandoahGC -Xmx100G means:
≈ 90..95 GB accessible for Java objects,
≈ 103 GB RSS for GC parts
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m

Serial
Parallel
G1
Shenandoah

Start Idle Load Idle Full GC Idle
RSS, MB
time, sec
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m

Serial
Parallel
G1
Shenandoah
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m

RSS, MB

<table>
<thead>
<tr>
<th>time, sec</th>
<th>Start</th>
<th>Idle</th>
<th>Load</th>
<th>Idle</th>
<th>Full GC</th>
<th>Idle</th>
</tr>
</thead>
</table>

Serial
Parallel
G1
Shenandoah
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m

RSS, MB
time, sec
Serial
Parallel
G1
Shenandoah

First uncommit
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m

RSS, MB

time, sec

Periodic GC

Start Idle Load Idle Full GC Idle

Serial
Parallel
G1
Shenandoah
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m

Serial
Parallel
G1
Shenandoah

Second uncommit

Slide 55/75. «Shenandoah GC», Aleksey Shipilëv, 2017, D:20171121102551+01'00'
Footprint: Microservice Example

wildfly-swarm-rest-http, wrk http test, JDK 10 x86-64, -Xmx512m
Footprint: Shenandoah’s M.O.

«We shall take all the memory when we need it, but we shall also give it back when we don’t»

1. Start with \(-X_{ms}\) committed memory
2. Expand aggressively under load up to \(-X_{mx}\)
3. Stay close to \(-X_{mx}\) under load
4. Uncommit the heap and bitmaps down to zero when idle
5. Do periodic GCs to knock out floating garbage when idle

Tunables: \(-X_{ms}, -X_{mx}, \text{periodic GC interval, uncommit delay}\)
Immediates: Living Space

Problem:
Concurrent GC needs breathing room to succeed

Things that help:
- Aggressive heap expansion: prefer taking more memory
- Immediate garbage shortcuts: free memory early
- Partial collections: collect easy parts of heap first
- Mutator pacing: stall allocators before they hit the wall
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

Exploiting weak gen hypothesis:
**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

**Exploiting weak gen hypothesis:**

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

Exploiting weak gen hypothesis:
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

Exploiting weak gen hypothesis:
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...
Partials: Living Space

**Problem:**
Concurrent GC needs breathing room to succeed

Things that help:
- Aggressive heap expansion: prefer taking more memory
- Immediate garbage shortcuts: free memory early
- Partial collections: collect easy parts of heap first
- Mutator pacing: stall allocators before they hit the wall
Partials: Heap Segregation

Central Dogma:
Segregate parts of the heap by some property (age, size, class, context, thread), and collect the subheaps separately
Partials: Heap Segregation

Central Dogma:
Segregate parts of the heap by some property (age, size, class, context, thread), and collect the subheaps separately

Pesky detail:
requires knowing the incoming references to the collected sub-heap
Partials: Serial/Parallel/CMS

Most GCs exploit this by dividing the heap into *generations*
Young gen can be collected separately, if we know the incoming references from Old gen. Card Table records this for us with the write barriers.
Young collection processes Young gen, and dirty parts of Old gen, thus maintaining heap integrity.
Partials: G1

Card Table

RSets

Regions

G1 is more advanced: it has Remembered Sets
Write barrier marks the Card Table. But it is not enough to quickly collect a single region: we would need to scan all dirty cards.
Using Card Table, G1 asynchronously builds Remembered Sets: the list of blocks that contain references to each region.
Now we can quickly collect a single region: RSet tells us what dirty parts related to the concrete region.
In practice, naive RSets are uber-large. G1 becomes generational: some regions are young, and no need to record references between them.
Interesting trade-off: cannot collect a single young region now!

Requires a careful balancing act to make sure pause times are good, and RSet footprint is small!
Idea: why not to have *much coarser* card table, but for each region?
Then we can support the connection matrix, and know things about heap connectivity.
Partials: Shenandoah

Example: collect first region, and matrix tells us we also need to scan the fourth.
Example: collect first region, and matrix tells us we also need to scan the fourth.

This works because the GC is concurrent, and we can spend time scanning the entire region!
Partials: Example

GC(75) Pause Init Mark 0.483ms
GC(75) Concurrent marking 33318M->45596M(51200M) 508.658ms
GC(75) Pause Final Mark 0.245ms
GC(75) Concurrent cleanup 45612M->16196M(51200M) 3.499ms

VS

GC(193) Pause Init Partial 1.913ms
GC(193) Concurrent partial 27062M->27082M(51200M) 0.108ms
GC(193) Pause Final Partial 0.570ms
GC(193) Concurrent cleanup 27086M->17092M(51200M) 15.241ms
1. Maintaining the connectivity data means more barriers!
   **Translation:** The increased GC efficiency need to offset more throughput overhead

2. *Optionality* helps where barriers overhead is too much
   **Translation:** No need to pay when partial doesn’t help

3. Advanced policies are possible, beyond generational
   **Example:** Take out lonely old regions
Mutator Pacing: Living Space

**Problem:**
Concurrent GC needs breathing room to succeed

Things that help:
- Aggressive heap expansion: prefer taking more memory
- Immediate garbage shortcuts: free memory early
- Partial collections: collect easy parts of heap first
- Mutator pacing: stall allocators before they hit the wall
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 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 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 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 solved for even shallower pauses. This is where Shenandoah comes in!
Conclusion: Releases

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

- Development in separate JDK 10 forest, regular backports to separate JDK 9 and 8u forests
- JDK 8u backport ships in RHEL 7.4+, Fedora 24+, and derivatives (CentOS, Oracle Linux, Amazon Linux, etc)
- Nightly development builds (tarballs, Docker images)

```
docker run -it --rm shipilev/openjdk:10-shenandoah \
  java -XX:+UseShenandoahGC -Xlog:gc -version
```
Trivia
### Trivia: Compiler Support

<table>
<thead>
<tr>
<th>Test</th>
<th>G1</th>
<th>Shen</th>
<th>%diff</th>
<th>G1</th>
<th>Shen</th>
<th>%diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmp</td>
<td>78</td>
<td>72</td>
<td>-7%</td>
<td>127</td>
<td>116</td>
<td>-8%</td>
</tr>
<tr>
<td>Cpr</td>
<td>125</td>
<td>86</td>
<td>-31%</td>
<td>146</td>
<td>125</td>
<td>-15%</td>
</tr>
<tr>
<td>Cry</td>
<td>79</td>
<td>62</td>
<td>-21%</td>
<td>238</td>
<td>240</td>
<td>+1%</td>
</tr>
<tr>
<td>Drb</td>
<td>75</td>
<td>69</td>
<td>-7%</td>
<td>165</td>
<td>150</td>
<td>-9%</td>
</tr>
<tr>
<td>Mpa</td>
<td>31</td>
<td>21</td>
<td>-33%</td>
<td>50</td>
<td>40</td>
<td>-20%</td>
</tr>
<tr>
<td>Sci</td>
<td>42</td>
<td>32</td>
<td>-23%</td>
<td>74</td>
<td>70</td>
<td>-5%</td>
</tr>
<tr>
<td>Ser</td>
<td>1626</td>
<td>1293</td>
<td>-20%</td>
<td>2450</td>
<td>2172</td>
<td>-11%</td>
</tr>
<tr>
<td>Sun</td>
<td>93</td>
<td>74</td>
<td>-20%</td>
<td>111</td>
<td>97</td>
<td>-13%</td>
</tr>
<tr>
<td>Xml</td>
<td>88</td>
<td>72</td>
<td>-19%</td>
<td>190</td>
<td>168</td>
<td>-12%</td>
</tr>
</tbody>
</table>

C1 codegens good barriers, but C2 **also** does high-level optimizations.
Trivia: JMM Tricks

We can read from-copy (i.e. skip RBs), as long as:

1. No locks, `volatile` reads/writes, memory barriers
2. No calls into the opaque methods
Trivia: JMM Tricks

We can read from-copy (i.e. skip RBs), as long as:
  1. No locks, volatile reads/writes, memory barriers
  2. No calls into the opaque methods

As the rule, we can:
  1. Avoid re-doing RBs after safepoints
  2. Erase RBs when reading final-s
Trivia: JMM Tricks

**final on fields finally improves performance!**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Score</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>plain</td>
<td>final</td>
</tr>
<tr>
<td>time</td>
<td>2.7 ± 0.1</td>
<td>2.6 ± 0.1</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>13.2 ± 0.1</td>
<td>11.2 ± 0.1</td>
</tr>
<tr>
<td>instructions</td>
<td>29.6 ± 0.6</td>
<td>28.5 ± 0.3</td>
</tr>
</tbody>
</table>
Trivia: Mark Solutions

Two classic approaches to solve this:

1. **Incremental Update**: intercept the stores, and process *insertions*, thus traversing new paths – good, but has weak termination guarantees

2. **Snapshot-at-the-Beginning**: intercept the stores, and process *deletions*, thus mitigating the destructive mutations – also good, but overestimates liveness

(there are also non-classic approaches, but not for this talk)