Shenandoah GC

...and how it looks like in February 2018

Aleksey Shipilëv

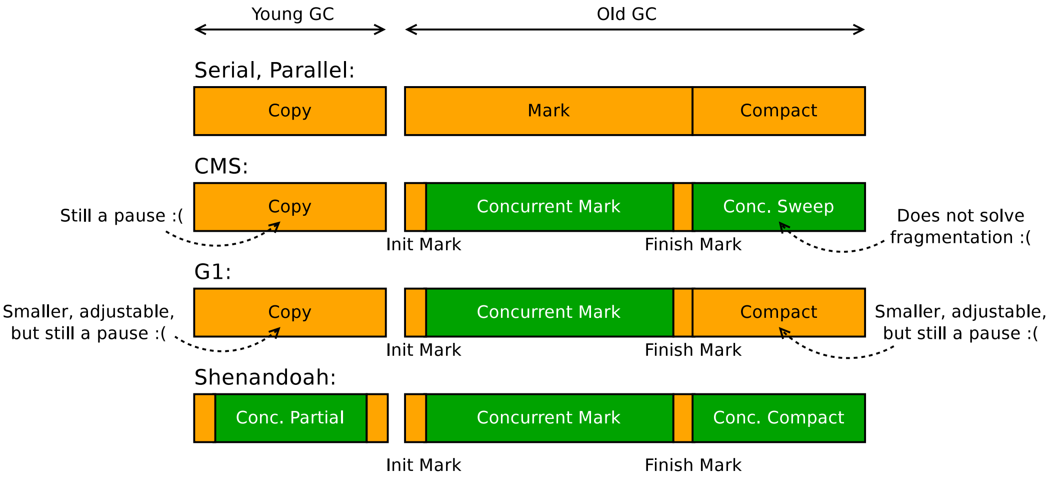
shade@redhat.com
@shipilev

Disclaimers First! 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



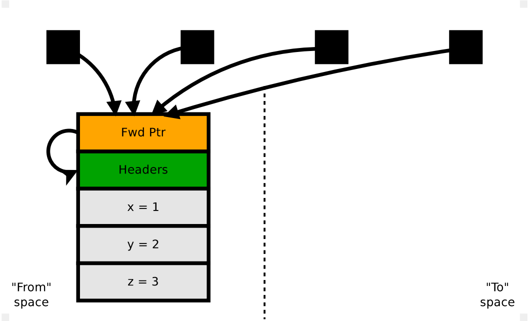
Overview: Key Idea (Java Analogy)

```
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...

Overview: Key Idea

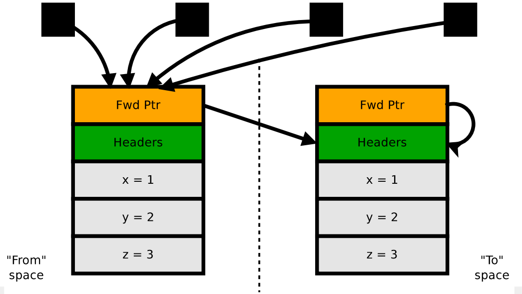
Brooks forwarding pointer to help concurrent copying:



fwdptr is attached to every instance,
all the times

Overview: Key Idea

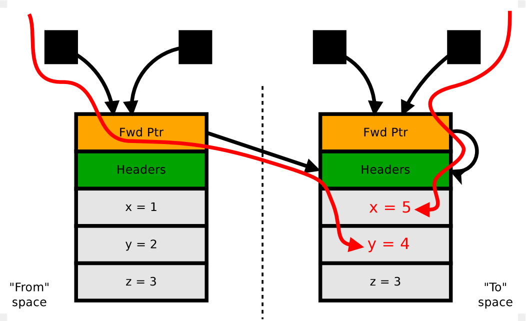
Brooks forwarding pointer to help concurrent copying:



fwdptr always points to most actual copy,
and gets atomically updated during evacuation

Overview: Key Idea

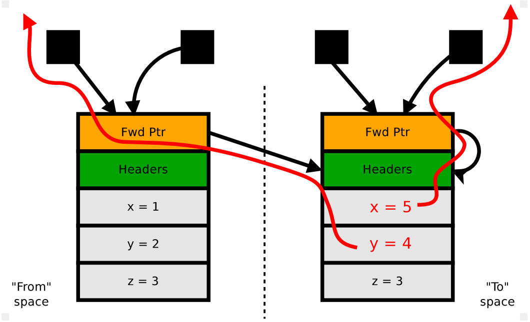
Brooks forwarding pointer to help concurrent copying:



Write barriers maintain **to-space invariant**:
«All writes happen into to-space copy»

Overview: Key Idea

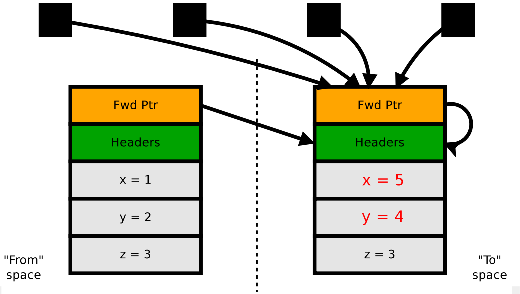
Brooks forwarding pointer to help concurrent copying:



Read barriers help to select the actual copy for reading
(**Not** the invariant: JLS allows reads from old copies)

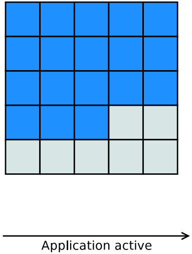
Overview: Key Idea

Brooks forwarding pointer to help concurrent copying:

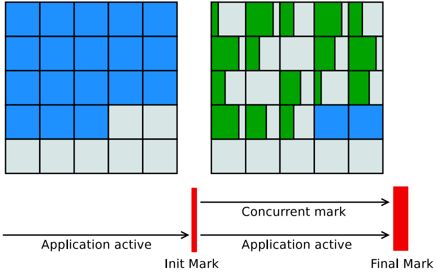


This mechanics allows to update
the heap references concurrently

Overview: Regular GC Cycle

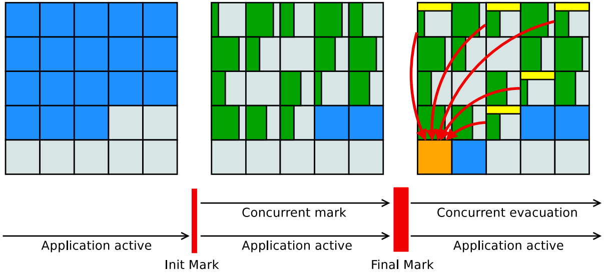


Overview: Regular GC Cycle



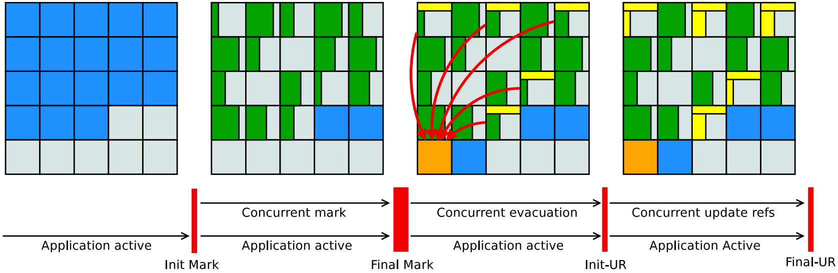
1. Snapshot-at-the-beginning concurrent mark

Overview: Regular GC Cycle



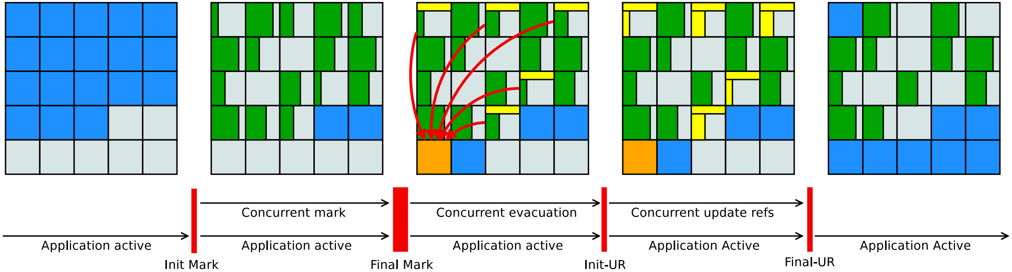
1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation

Overview: Regular GC Cycle



1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
3. Concurrent update references

Overview: Regular GC Cycle



1. Snapshot-at-the-beginning concurrent mark
2. Concurrent evacuation
3. Concurrent update references

   (optional, can be coalesced with upcoming cycle marking)

Basics

Basics: Concurrent GC 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
```

Basics: Concurrent GC 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
```

Basics: Concurrent Means Freedom

Concurrent collector runs GC cycles
without blocking application progress

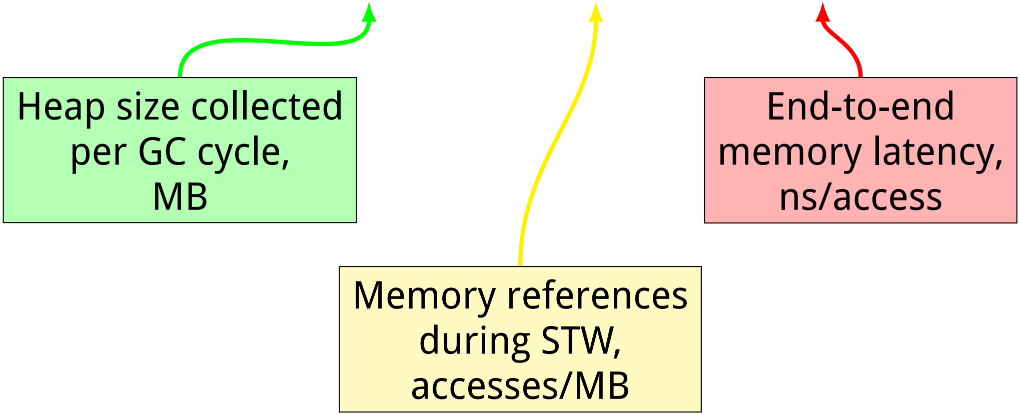
- Slow concurrent phase means higher GC duty cycle
  - Steal more cycles from application, not pause it extensively
  - Heuristics mistakes are (usually) much less painful
  - Control the GC cycle time budget: `-XX:ConcGCThreads=...`
- Testing:
  - **periodic** GCs without significant penalty
  - **continuous** GC (+ «back-to-back») gets the lowest footprint
  - **aggressive** GC (+ «move everything») aids testing a lot

Basics: Concurrent GC Only For Large Heaps?

$$Latency_{GC} = \alpha * Size_{heap} * MemRefs_{stw} * Latency_{mem}$$

Basics: Concurrent GC Only For Large Heaps?

$$Latency_{GC} = \alpha * Size_{heap} * MemRefs_{stw} * Latency_{mem}$$



Heap size collected per GC cycle, MB

End-to-end memory latency, ns/access

Memory references during STW, accesses/MB

Basics: Concurrent GC Only For Large Heaps?

$$Latency_{GC} = \alpha * Size_{heap} * MemRefs_{stw} * Latency_{mem}$$

«Large heap»:

- $Size_{heap}$ goes up, $MemRefs_{stw}$ must go down
- This assumes $Latency_{mem}$ is low

Basics: Concurrent GC Only For Large Heaps?

$$Latency_{GC} = \alpha * Size_{heap} * MemRefs_{stw} * Latency_{mem}$$

«Large heap»:

- $Size_{heap}$ goes up, $MemRefs_{stw}$ must go down
- This assumes $Latency_{mem}$ is low

«Slow hardware»:

- $Latency_{mem}$ goes up, $MemRefs_{stw}$ must go down!
- This assumes $Size_{heap}$ is low

Basics: Slow Hardware

Raspberry Pi 3, running springboot-petclinic:

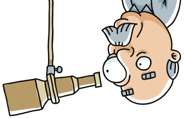
```
# -XX:+UseShenandoahGC
Pause Init Mark 8.991ms
Concurrent marking 409M->411M(512M) 246.580ms
Pause Final Mark 3.063ms
Concurrent cleanup 411M->89M(512M) 1.877ms

# -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!

- 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)

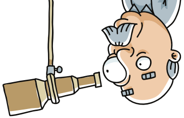
```
docker run -it --rm shipilev/openjdk-shenandoah \
   java -XX:+UseShenandoahGC -Xlog:gc -version
```



Basics: Observations

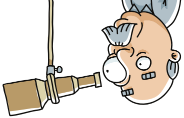
1. Concurrent GC works, and works fine
    - Figuring out throughput, latency hiccups, footprint features
    - Testing, refactoring, bugfixes are significant part of the story

Basics: Observations



1. Concurrent GC works, and works fine
    - Figuring out throughput, latency hiccups, footprint features
    - Testing, refactoring, bugfixes are significant part of the story
2. Adoption provides surprises
    - Small-to-mid heap sizes (below CompressedOops limit?)
    - Care about latencies only so much (<10 ms is okay)
    - Care about footprint a lot! (see next section)
    - Able to accept 10-20% throughput hit

Basics: Observations



1. Concurrent GC works, and works fine
    - Figuring out throughput, latency hiccups, footprint features
    - Testing, refactoring, bugfixes are significant part of the story
2. Adoption provides surprises
    - Small-to-mid heap sizes (below CompressedOops limit?)
    - Care about latencies only so much (<10 ms is okay)
    - Care about footprint a lot! (see next section)
    - Able to accept 10-20% throughput hit
3. Backports are very important part of the story
    - We have no adopters for sh/jdk10!
    - Real People (tm) are on sh/jdk8u, or RHEL/Fedora RPMs

Footprint

Footprint: Overheads

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

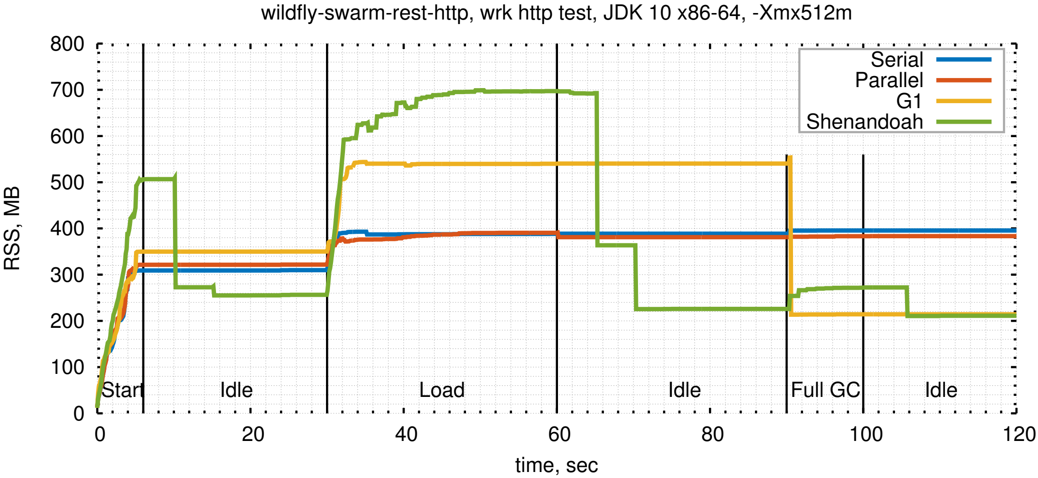
- Java heap: 1.5x worst and 1.05-1.10x avg overhead
  «−»: the overhead is non-static
  «+»: counted in Java heap – no surprise RSS inflation
- Native structures: 2x marking bitmaps, each 1/64 of heap
  «−»: -Xmx is still not close to RSS
  «+»: overhead is static: `-Xmx100g` means 103 GB RSS

Footprint: Overheads

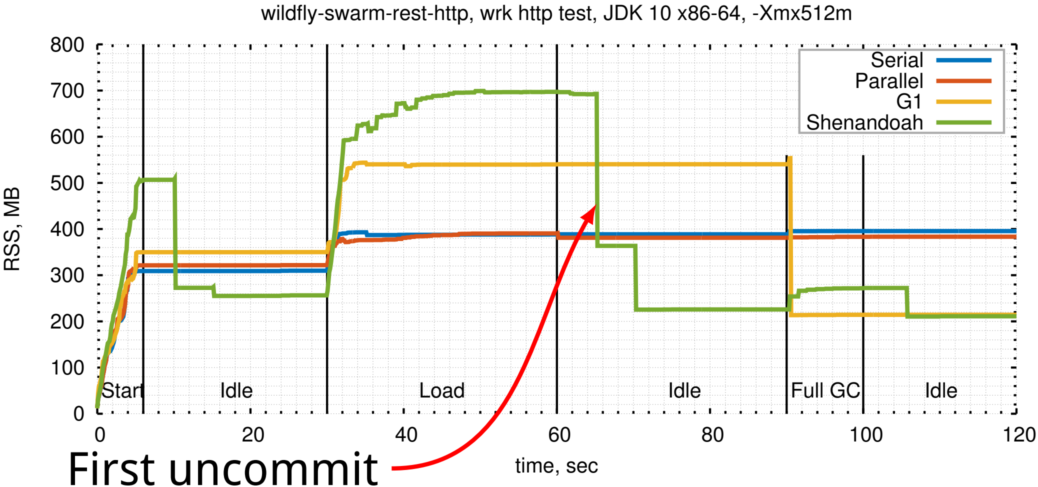
Shenandoah requires additional word per object
for forwarding pointer at all times, plus some native structs

- 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 Uncommit

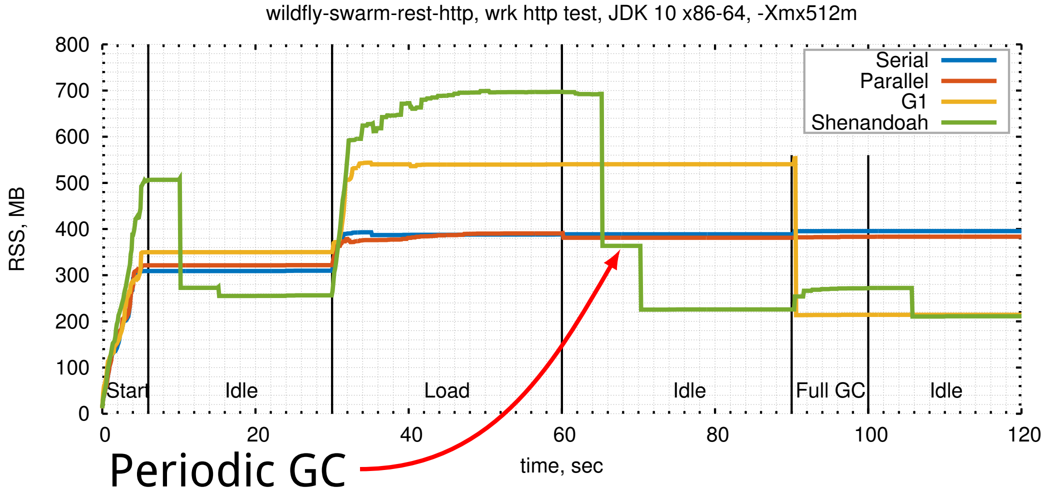


Footprint: Heap Uncommit



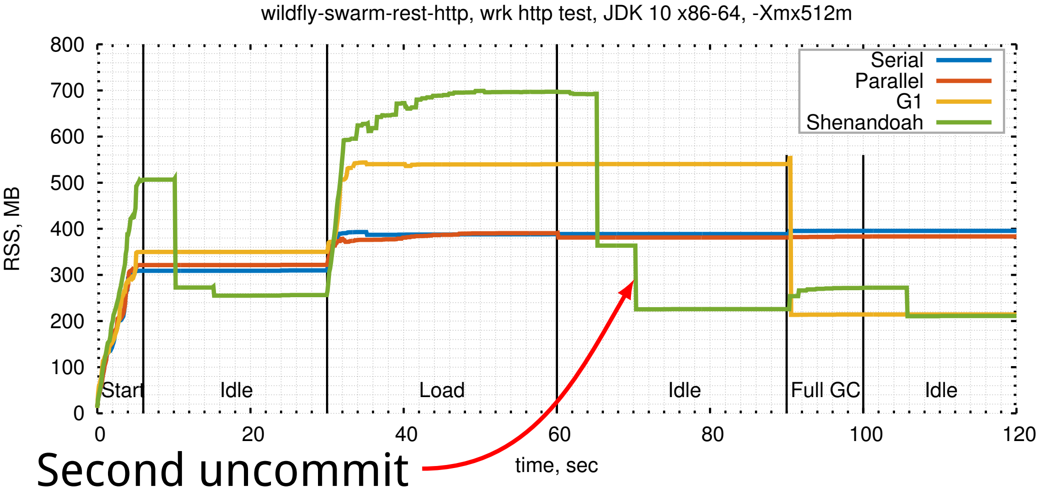
First uncommit

Footprint: Heap Uncommit



Periodic GC

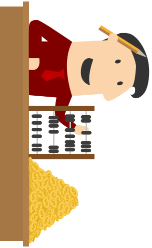
Footprint: Heap Uncommit



Second uncommit

Footprint: Enterprise Hello World

Start with `-Xmx100g`, allocate a terabyte of garbage, print «Hello World», wait for first customer to never come:

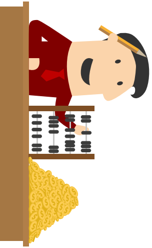


```
; After startup
Total: reserved=109842185KB, committed=108152925KB
 Heap: reserved=104857600KB, committed=104857600KB
   GC: reserved=  4917136KB, committed=  3278736KB

; 5 minutes later:
Total: reserved=109842307KB, committed=    52439KB
 Heap: reserved=104857600KB, committed=    32768KB
   GC: reserved=  4917186KB, committed=     3010KB
```

Footprint: Enterprise H

Easy cloud savings right there
(Cloud providers hate this guy![1])

Start with `-Xmx100g`, allocate a terabyte of garbage, print «Hello World», wait for first customer to never come:

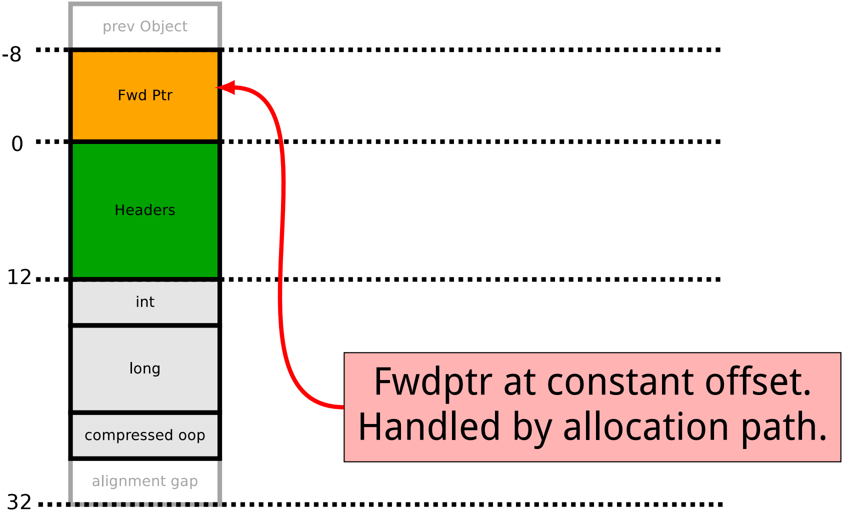


```
; After startup
Total: reserved=109842185KB, committed=108152925KB
 Heap: reserved=104857600KB, committed=104857600KB
   GC: reserved=  4917136KB, committed=  3278736KB

; 5 minutes later:
Total: reserved=109842307KB, committed=    52439KB
 Heap: reserved=104857600KB, committed=    32768KB
   GC: reserved=  4917186KB, committed=     3010KB
```

---

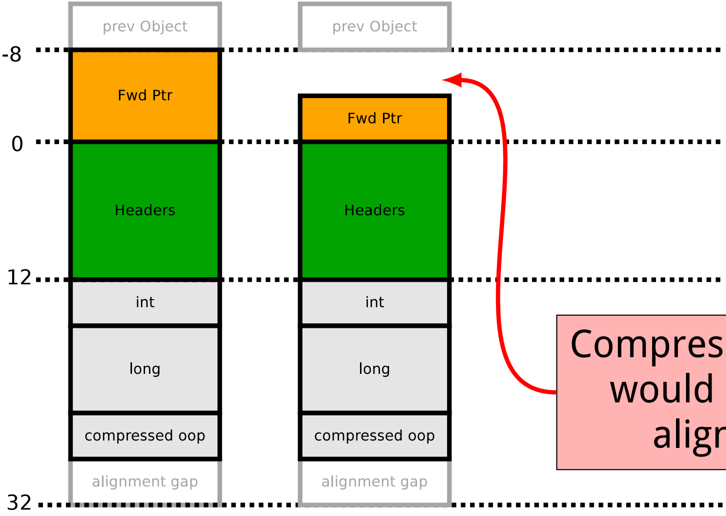
[1] Or not: `https://jelastic.com/blog/tuning-garbage-collector-java-memory-usage-optimization/`

Footprint: Future Improvements



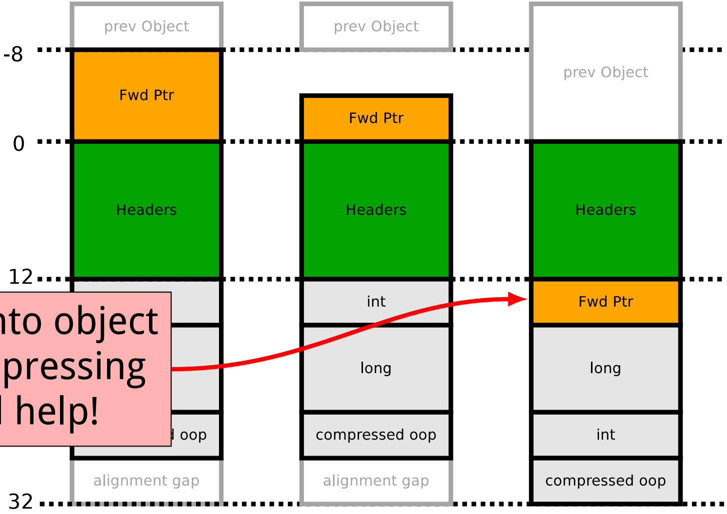
Fwdptr at constant offset.
Handled by allocation path.

Footprint: Future Improvements



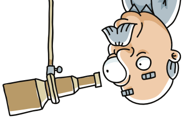
Compressing fwdptr would not help: alignment!

Footprint: Future Improvements



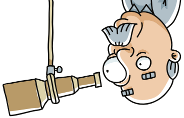
Moving into object and compressing would help!

Footprint: Observations



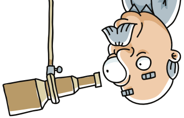
1. Footprint story is nuanced
    - Blindly counting bytes taken by Java heap and GC does not cut it

Footprint: Observations



1. Footprint story is nuanced
    - Blindly counting bytes taken by Java heap and GC does not cut it
2. Fwdptr overhead is substantial and manageable
    - Comparing with per-oop-field cost is hard!
    - More intrusive fwdptr injection cuts the overhead down

Footprint: Observations



1. Footprint story is nuanced
   - Blindly counting bytes taken by Java heap and GC does not cut it
2. Fwdptr overhead is substantial and manageable
   - Comparing with per-oop-field cost is hard!
   - More intrusive fwdptr injection cuts the overhead down
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

Barriers

Barriers: Sadness Distilled

Sad part of barriers story:
Shenandoah needs much more barriers

1. SATB barriers for **reference** stores
2. Write barriers on **all stores**, not only reference stores
3. Read barriers on **almost all heap reads**
4. Other exotic barriers: acmp, CAS, clone, ...

Barriers: SATB Barriers

```
# Read TLS flag and see if mark is enabled
cmpb   0x2, 0x3d8(%r15)
jnz    OMG-MARKING

# ...actual ref store follows...
```

- Incidence: covers all reference stores
- Reason: captures destructive stores that break marking
- Impact: `0..3%` throughput hit
- Optimizeability: medium, requires raw memory slices

Barriers: Read Barriers

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

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

- Incidence: before almost every heap read
- Reason: support concurrent copying
- Impact: `0..15%` throughput hit
- Optimizeability: good, barriers move with heap accesses

Barriers: Write Barriers

```
# Read TLS flag and see if evac is enabled
cmpb    0x4, 0x3d8(%r15)
jne     OMG-EVAC-ENABLED    # Oh my...

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

# ...actual store follows...
```

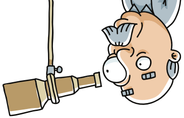
- Incidence: before almost every heap write
- Reason: support to-space invariant
- Impact: `0..5%` throughput hit
- Optimizeability: medium, requires weird voodoo magic

Barriers: ACMP, CAS, etc

```
# 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)
cmp     %rcx,%rdx            # if (a1 == a2) ...
```

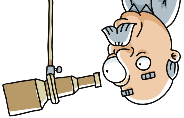
- Incidence: on many reference comparisons (acmp, CAS)
- Reason: unequal machine ptrs $\neq$ unequal Java refs!
- Impact: `0..5%` throughput hit
- Optimizeability: good, comparisons with `null` are trivial

Barriers: Observations



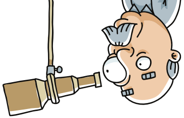
1. Easily portable across HW architectures
    - Special needs: `CAS` (performance largerly irrelevant)
    - x86_64 and AArch64 are major implemented targets

Barriers: Observations



1. Easily portable across HW architectures
   - Special needs: `CAS` (performance largely irrelevant)
   - x86_64 and AArch64 are major implemented targets
2. Trivially portable across OSes
   - Special needs: none
   - Linux is major target
   - Adopters build on Windows and Mac OS X without problems

Barriers: Observations



1. Easily portable across HW architectures
   - Special needs: `CAS` (performance largerly irrelevant)
   - x86_64 and AArch64 are major implemented targets
2. Trivially portable across OSes
   - Special needs: none
   - Linux is major target
   - Adopters build on Windows and Mac OS X without problems
3. VM interactions are simple enough
   - Play well with compressed oops: separate fwdptr
   - OS/CPU-specific things only for barriers codegen
   - Throughput overheads get better with compiler opts (see later)

Partial

Partial: 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

Partial: Obvious Shortcut

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

Partial: Obvious Shortcut

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

1. Mark is fast, because most things are dead

Partial: Obvious Shortcut

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

Partial: Obvious Shortcut

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

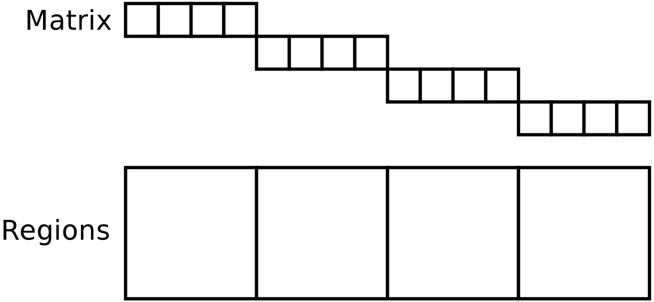
Partial: Partials

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 Tables, finer grained Remembered Sets, etc)
- Differs from regular cycle: selects the collection set without prior marking, thus more conservative
- *Generational* is the special case of partial

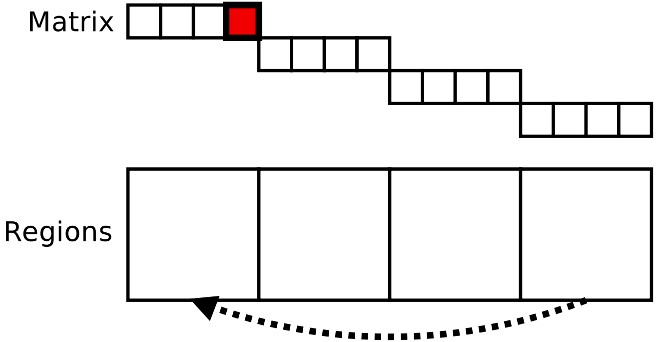
Partial: Partials, Connection Matrix

Concurrent collector allows for the very coarse «connection matrix»: the 2D incidence matrix for region connection graph



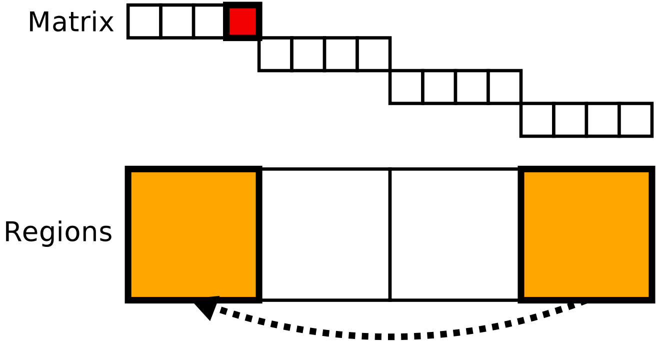
Partial: Partials, Connection Matrix

Concurrent collector allows for the very coarse «connection matrix»: the 2D incidence matrix for region connection graph



Partial: Partials, Connection Matrix

Concurrent collector allows for the very coarse «connection matrix»: the 2D incidence matrix for region connection graph

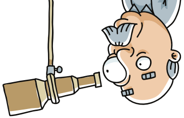


Partial: Example

```
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

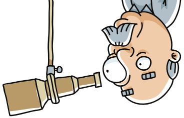
```
GC(193) Pause Init Partial 1.913ms
GC(193) Concurrent partial 27062M->27082M(51200M) 0.108ms
GC(193) Pause Final Partial 0.570ms
GC(193) Concurrent cleanup 27086M->17092M(51200M) 15.241ms
```



Partial: Observations

1. Immediate garbage shortcuts approximate generational
   - Catch-22: Most workloads are fully young

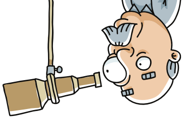
Partial: Observations



1. Immediate garbage shortcuts approximate generational
   - Catch-22: Most workloads are fully young

2. Partial collections help when LDS is low-to-mid
   - Maintaining the connectivity data means more barriers!
   - Increased GC efficiency need to offset more overhead
   - Optionality helps where barriers overhead is too much

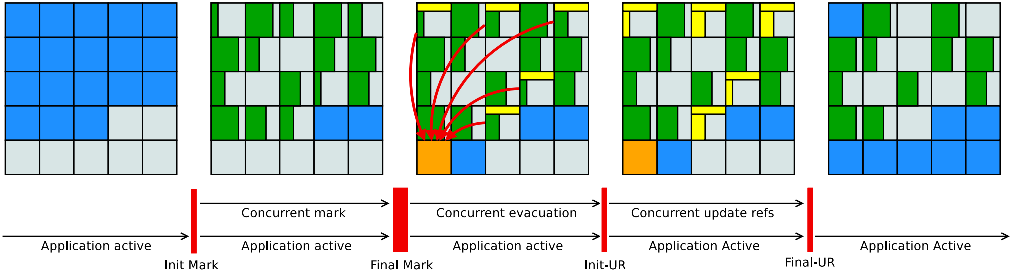
Partial: Observations



1. Immediate garbage shortcuts approximate generational
    - Catch-22: Most workloads are fully young
2. Partial collections help when LDS is low-to-mid
    - Maintaining the connectivity data means more barriers!
    - Increased GC efficiency need to offset more overhead
    - Optionality helps where barriers overhead is too much
3. Nothing helps when LDS is high
    - Generational becomes actively harmful
    - Some partial policies may help to unclutter heap
    - Need to handle concurrent GC failures (see later)

Traversal Order

Traversal Order: Spot The Trouble



Concurrent mark

Concurrent evacuation

Concurrent update refs

Application active

Application active

Application active

Application Active

Application Active

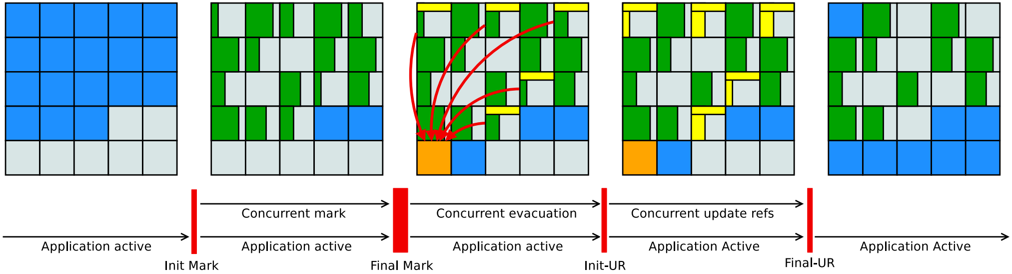
Init Mark

Final Mark

Init-UR

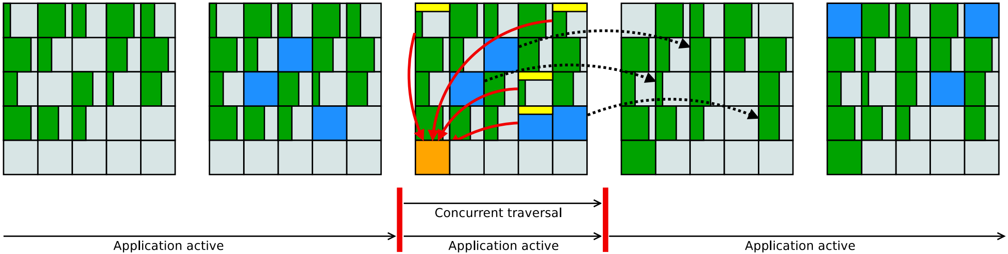
Final-UR

Traversal Order: Spot The Trouble



Separate marking and evacuation phases mean collector maintains the *allocation* order, not the *traversal* order

Traversal Order: Traversal GC



```
GC(57) Pause Init Traversal 1.705ms
GC(57) Concurrent traversal 14967M->15288M(16384M) 200.259ms
GC(57) Pause Final Traversal 4.028ms
GC(57) Concurrent cleanup 15311M->5563M(16384M) 16.431ms
```

Traversal Order: Layout-Sensitive Test

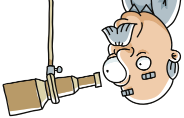
```
@Param({"1", "100", "10000", "1000000"})
int size;

// map of "size" keys/values
// backing array is Object[]
Map<String, String> map = ...;

@Benchmark
public void test(Blackhole bh) {
  for (Map.Entry<String, String> kv : map.entrySet()) {
    bh.consume(kv.getKey());
    bh.consume(new Object());
  }
}
```

Traversal Order: Layout-Sensitive Test

Reference locality FTW in some cases:

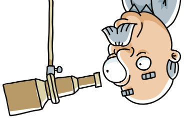
map size	time, us/op				Impr
	default		traversal		
1	0.02	± 0.01	0.02	± 0.01	+0%
100	1.06	± 0.02	0.93	± 0.01	+13%
10000	207.25	± 2.74	185.52	± 0.36	+11%
1000000	48499.42	± 479.39	43066.18	± 343.03	+13%

Traversal Order: Observations



1. Allocation order is not always perfect
    - Sometimes it is – the only thing that user can control
    - Traversal order seems to be a fair approximation of most uses

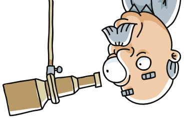
Traversal Order: Observations



1. Allocation order is not always perfect
    - Sometimes it is – the only thing that user can control
    - Traversal order seems to be a fair approximation of most uses

2. Unintended consequence: merging all phases in one
    - Makes us walk the heap once, not thrice

Traversal Order: Observations



1. Allocation order is not always perfect
   - Sometimes it is – the only thing that user can control
   - Traversal order seems to be a fair approximation of most uses

2. Unintended consequence: merging all phases in one
   - Makes us walk the heap once, not thrice

3. Unintended consequence: fewer barriers
   - Binary GC state: «idle» + «traversal»
   - Barrier optimization story gets easier (see later)

Handling Failures

Handling Failures: Practicals

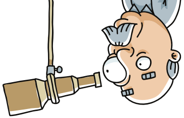
Happy concurrent GC relies on *collecting faster than applications allocate*: applications **always** see there is available memory

- 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, application allocations outpace GC work – what do we do then?

Handling Failures: Approaches

- **Fail Hard**: crash the VM
  (Epsilon)
- **Fail Hard to STW**: assume the worst, dive into Full GC
  (Shenandoah, beginning 2017)
- **Fail Soft to STW**: dive to STW, complete the cycle there
  (Shenandoah: mid/end 2017, aka «Degenerated GC»)
- **Fail Wait**: wait until memory is available
  (Shenandoah experiments, discontinued)

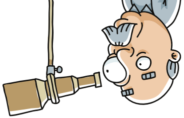
Handling Failures: Degenerated GC



```
Pause Init Update Refs 0.034ms
Cancelling concurrent 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 Degenerated GC
- Degenerated GC *continues* the cycle
- Second allocation failure may upgrade to Full GC

Handling Failures: Degenerated GC



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

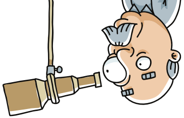
- First allocation failure dives into Degenerated GC
- Degenerated GC *continues* the cycle
- Second allocation failure may upgrade to Full GC

Handling Failures: Full GC

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

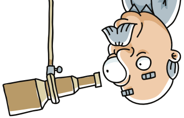
- Designed to recover from anything: 99% full regions, heavy (humongous) fragmentation, abort from any point in concurrent GC, etc.
- Parallel: Multi-threaded, runs on-par with Parallel GC
- Sliding: No additional memory needed + reuses fwdptr slots to store forwarding data

Handling Failures: Observations



1. Handling GC failures is important part of the story
    - Few people care when GC performs well. When it fails? Oh my!
    - Most tuning guides would talk about avoiding failures

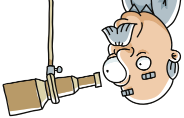
Handling Failures: Observations



1. Handling GC failures is important part of the story
   - Few people care when GC performs well. When it fails? Oh my!
   - Most tuning guides would talk about avoiding failures
2. Graceful degradation is key
   - Observability is the part of grace
   - If you are stalling the application threads, honestly say so!

redhat

Handling Failures: Observations



1. Handling GC failures is important part of the story
   - Few people care when GC performs well. When it fails? Oh my!
   - Most tuning guides would talk about avoiding failures
2. Graceful degradation is key
   - Observability is the part of grace
   - If you are stalling the application threads, honestly say so!
3. Failure paths performance is important
   - «Your system melted down because you have misconfigured our oh-so-perfect product» flies only so much...
   - Unconditionally failing to STW is performance diagnostics tool!

Compiler Support

Compiler Support: Overview

The key thing to achieve low pauses
with decent throughput performance
are compiler optimizations[2]

---

[2] Also the major source of interesting bugs

Compiler Support: Overview

The key thing to achieve low pauses
with decent throughput performance
are compiler optimizations[2]

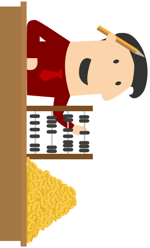
Several categories:

1. Generic optimizations that help all GCs
2. Semi-generic optimizations that unblock GC-specific fixes
3. Special optimizations for specific GCs

---

[2] Also the major source of interesting bugs

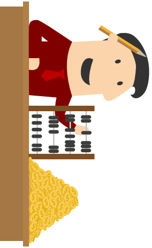
Compiler Support: In Numbers



	C1			C2		
Test	**Par**	**Shen**	**%diff**	**Par**	**Shen**	**%diff**
Compiler*	753	634	-16%	1178	1009	-14%
Compress	1265	832	-34%	1533	1334	-13%
Crypto*	649	509	-22%	2273	2210	-3%
Derby	742	649	-12%	1609	1475	-8%
MpegAudio	291	199	-32%	475	416	-12%
Scimark*	303	232	-23%	521	486	-7%
Serial	14473	11272	-22%	21890	19604	-10%
Sunflow	255	196	-23%	285	264	-7%
Xml*	510	430	-16%	1821	1568	-14%

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

Compiler Support: Long Loops



```
int[] arr;

@Benchmark
public int test() throws InterruptedException {
  int r = 0;
  for (int i : arr)
  r = (i * 1664525 + 1013904223 + r) % 1000;
  return r;
}
```

```
# java -XX:+UseShenandoahGC -Dsize=10'000'000
Performance: 35.832 +- 1.024  ms/op
Total Pauses (G) =  0.69 s (a = 26531 us)
Total Pauses (N) =  0.02 s (a =   734 us)
```

Compiler Support: Loop Strip Mining[3]

Make a smaller bounded loop without the safepoint polls inside the original one:

```
for (c : [0, L]) {
  use(c);
  <safepoint poll>
}
```

⇒

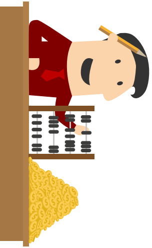
```
for (c : [0, L] by M) {
  for (k : [0: M]) {
    use(c + k);
  }
  <safepoint poll>
}
```

Amortize safepoint poll costs without sacrificing TTSP!

---

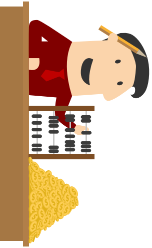
[3] https://bugs.openjdk.java.net/browse/JDK-8186027

Compiler Support: Loop Strip Mining



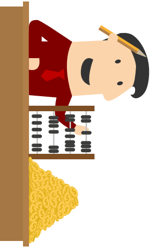
```
# -XX:+UseShenandoahGC -XX:-UseCLS
Performance: 35.832 +- 1.024  ms/op
Total Pauses (G) =  0.69 s (a = 26531 us)
Total Pauses (N) =  0.02 s (a =   734 us)
```

Compiler Support: Loop Strip Mining



```
# -XX:+UseShenandoahGC -XX:-UseCLS
Performance: 35.832 +- 1.024  ms/op
Total Pauses (G) =  0.69 s (a = 26531 us)
Total Pauses (N) =  0.02 s (a =   734 us)

# -XX:+UseShenandoahGC -XX:+UseCLS -XX:LSM=1
Performance: 38.043 +- 0.866  ms/op
Total Pauses (G) =  0.02 s (a =   811 us)
Total Pauses (N) =  0.02 s (a =   670 us)
```

Compiler Support: Loop Strip Mining



```
# -XX:+UseShenandoahGC -XX:-UseCLS
Performance: 35.832 +- 1.024  ms/op
Total Pauses (G) =  0.69 s (a = 26531 us)
Total Pauses (N) =  0.02 s (a =   734 us)

# -XX:+UseShenandoahGC -XX:+UseCLS -XX:LSM=1
Performance: 38.043 +- 0.866  ms/op
Total Pauses (G) =  0.02 s (a =   811 us)
Total Pauses (N) =  0.02 s (a =   670 us)

# -XX:+UseShenandoahGC -XX:+UseCLS -XX:LSM=1000
Performance: 34.660 +- 0.657  ms/op
Total Pauses (G) =  0.03 s (a =   842 us)
Total Pauses (N) =  0.02 s (a =   682 us)
```

Compiler Support: Switch Profiling[4]

```
for (int pos = 0; pos < size; pos++) {
  int b1 = buf[pos] & 0xFF;
  switch (b1 >> 4) {
    case 0: case 1: case 2: case 3:
    case 4: case 5: case 6: case 7:
      cbuf[cpos++] = ...; break;
    case 12: case 13:
      cbuf[cpos++] = ...; break;
    case 14:
      cbuf[cpos++] = ...; break;
    default: throw new IllegalStateException();
  }
}
```

---

[4] http://mail.openjdk.java.net/pipermail/shenandoah-dev/2018-February/004886.html

Compiler Support: Switch Profiling[4]

```
for (int pos = 0; pos < size; pos++) {
  int b1 = buf[pos] & 0xFF;
  switch (b1 >> 4) {
    case 0: case 1: case 2: case 3:
    case 4: case 5: case 6: case 7:
      cbuf[cpos++] = ...; break;
    case 12: case 13:
      cbuf[cpos++] = ...; break;
    case 14:
      cbuf[cpos++]
    default: thro
  }
}
```

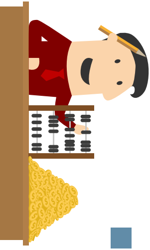
Most frequent branch,
but the absence of profiling
messes everything up

---

[4] http://mail.openjdk.java.net/pipermail/shenandoah-dev/2018-February/004886.html

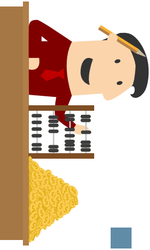
Compiler Support: Switch Profiling, #2

GC	Score, ns/op		Improv
	Baseline	Switch Prof	
Shenandoah	3963 ± 10	681 ± 10	5.8x



- Very profitable optimization

Compiler Support: Switch Profiling, #2

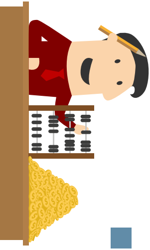


GC	Score, ns/op		Improv
	Baseline	Switch Prof	
Parallel	3084 ± 10	600 ± 10	5.1x
Shenandoah	3963 ± 10	681 ± 10	5.8x

- Very profitable optimization
- Generic optimization: helps everyone

Compiler Support: Switch Profiling, #2

GC	Score, ns/op				Improv
	Baseline		Switch Prof		
Parallel	3084	± 10	600	± 10	5.1x
Shenandoah	3963	± 10	681	± 10	5.8x
	-28%		-13%		



- Very profitable optimization
- Generic optimization: helps everyone
- Helps some GCs better: e.g. barrier moves

Compiler Support: Common Up Happy Paths

```
void m(Holder hld) { this.obj = hld.obj; }
```

We have:

```
mov  -0x8(%HLD), %HLD
mov  0x10(%HLD), %V
cmpb 0x2, (GC-STATE)
jnz  SATB-ENABLED
cmpb 0x4, (GC-STATE)
jnz  EVAC-ENABLED
mov  -0x8(%THIS), %THIS
mov  %V, 0x10(%THIS)
test 0x13371337(%rip), %rax
ret
```

Compiler Support: Common Up Happy Paths

```
void m(Holder hld) { this.obj = hld.obj; }
```

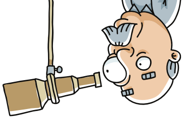
We have:

```
mov  -0x8(%HLD), %HLD
mov  0x10(%HLD), %V
cmpb 0x2, (GC-STATE)
jnz  SATB-ENABLED
cmpb 0x4, (GC-STATE)
jnz  EVAC-ENABLED
mov  -0x8(%THIS), %THIS
mov  %V, 0x10(%THIS)
test 0x13371337(%rip), %rax
ret
```

We can do:

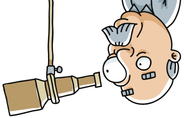
```
cmpb 0x0, (GC-STATE)
jnz  HEAP-UNSTABLE
mov  0x10(%HLD), %V
mov  %V, 0x10(%THIS)
test 0x13371337(%rip), %rax
ret
```

Compiler Support: Observations



1. Compiler optimizations make barrier overheads better
   - The hope is to get it down to low-single-digit percents

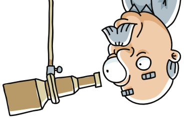
Compiler Support: Observations



1. Compiler optimizations make barrier overheads better
    - The hope is to get it down to low-single-digit percents

2. Compiler optimizations are high-level
    - No need to care about OS/CPU specific things
    - Helps things beyond Shenandoah

Compiler Support: Observations



1. Compiler optimizations make barrier overheads better
   - The hope is to get it down to low-single-digit percents
2. Compiler optimizations are high-level
   - No need to care about OS/CPU specific things
   - Helps things beyond Shenandoah
3. Compiler diffs makes perf comparisons uber-hard
   - Different baselines! Parallel GC is faster where: `jdk/jdk`, `jdk/hs`, `shenandoah/jdk10`, or `zgc/zgc`?
   - The way out is to put everything into single repo?

Conclusion

Conclusion: Ready for Experimental Use

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

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

Backup

Backup: VM Support

Pauses $\leqslant 1\ ms$ require more runtime support

Some examples:

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

Backup: STW Woes

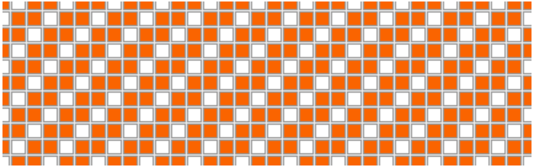
Pauses $\approx 1\ ms$ leave little time budget to deal with, but need to scan roots, cleanup runtime stuff, walk over regions...

Consider:

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

Backup: Humongous and 2^K allocs

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

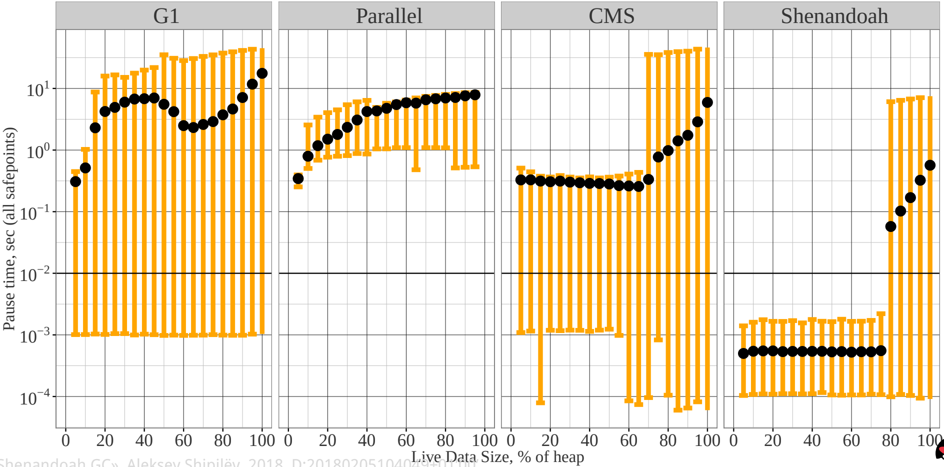


- Actually, in G1-style humongous allocs, the **worst** fit: objects have headers, and 2^K-sized alloc would barely **not** fit, wasting one of the regions

Q: Can be redone with segregated-fits freelist maintained separately?

Backup: Almost Concurrent Works Fine!

LRUFragger, 100 GB heap, varying LDS:



Backup: Almost Concurrent Works Fine!

LRUFragger, 100 GB heap, varying LDS: