Shenandoah GC

Part I: The Garbage Collector That Could

Aleksey Shipilëv

shade@redhat.com
@shipilev

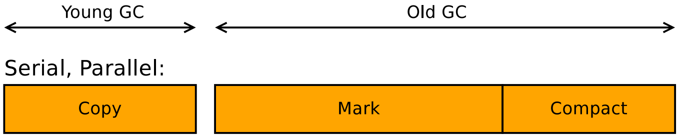
Safe Harbor / Тихая Гавань

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.

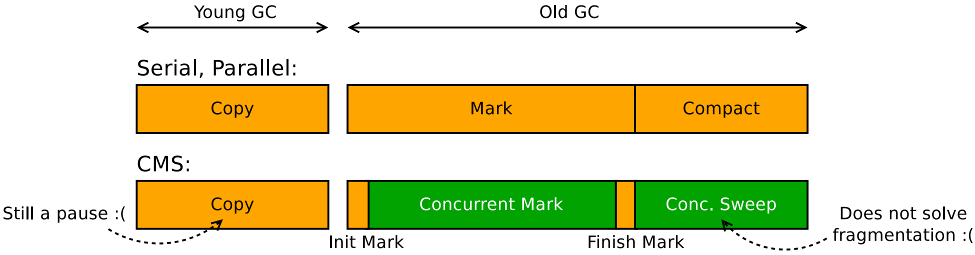
Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.

Basics

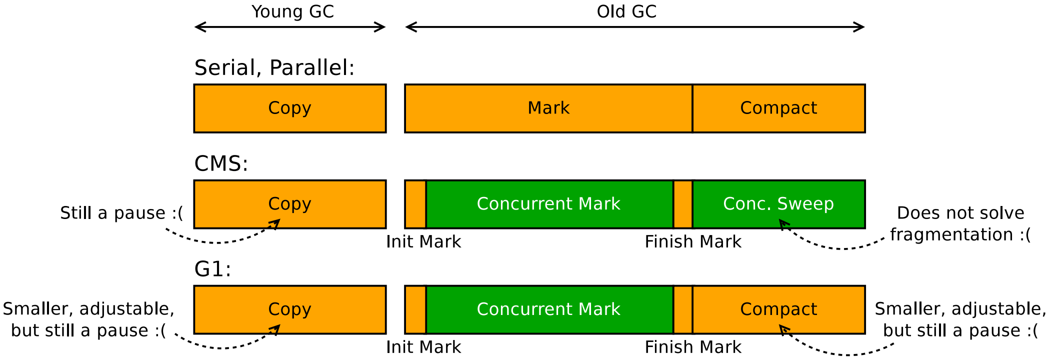
Basics: OpenJDK GCs Landscape



Basics: OpenJDK GCs Landscape

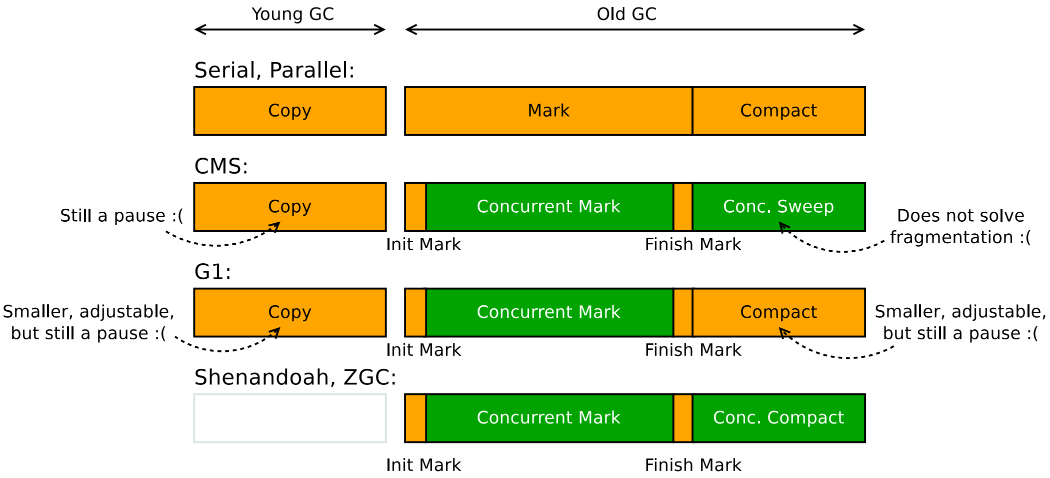


Basics: OpenJDK GCs Landscape



redhat.

Basics: OpenJDK GCs Landscape



Basics: Concurrent GC Only For Large Heaps?

Basics: Concurrent GC Only For Large Heaps?

$$Latency_{stw} = \alpha * Size_{heap} * MemRefs_{stw} * MemLatency_{avg}$$

Basics: Concurrent GC Only For Large Heaps?

$$Latency_{stw} = \alpha * Size_{heap} * MemRefs_{stw} * MemLatency_{avg}$$

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?

Observation	$Latency_{stw}$ components		
	$\alpha * Size_{heap}$	$MemRefs_{stw}$	$MemLatency_{avg}$
Large heap	$\uparrow\uparrow$	$\downarrow\downarrow$	$\approx$

- Large heap: large live data sets $\Rightarrow$ need concurrent GC

Basics: Concurrent GC Only For Large Heaps?

Observation	$Latency_{stw}$ components		
	$\alpha * Size_{heap}$	$MemRefs_{stw}$	$MemLatency_{avg}$
Large heap	$\uparrow\uparrow$	$\downarrow\downarrow$	$\approx$
Slow hardware	$\approx$	$\downarrow\downarrow$	$\uparrow\uparrow$

- Large heap: large live data sets $\Rightarrow$ need concurrent GC
- Slow hardware: memory is slow $\Rightarrow$ 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
```


redhat.

Basics: Releases

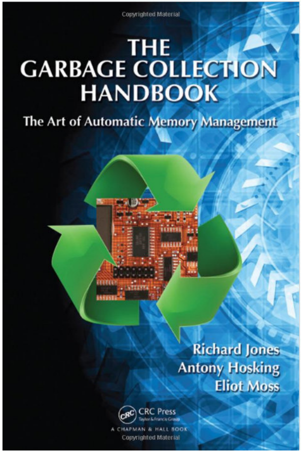
Easy to access (development) releases: try it now!

`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
```

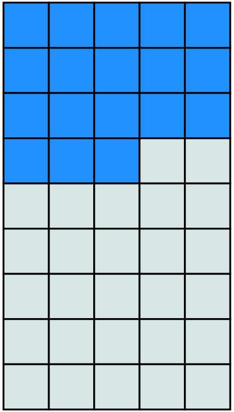
Basics: This Message Is Brought To You By



- 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

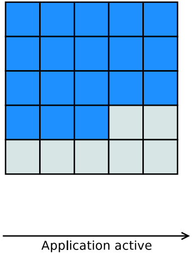
Overview: Heap Structure



Shenandoah is a *regionalized* GC

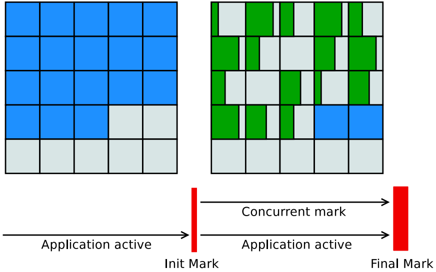
- Heap division, humongous regions, etc are similar to G1
- Collects garbage regions first by default
- Not generational by default, no young/old separation, even temporally
- Tracking inter-region references is not needed by default

Overview: Usual Cycle



Three major phases:

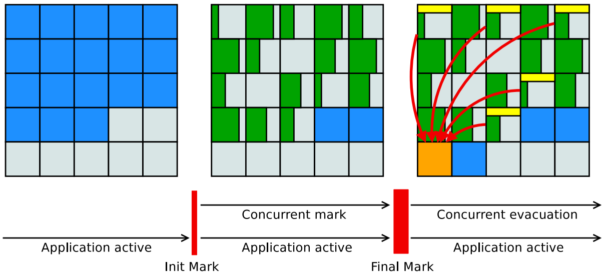
Overview: Usual Cycle



Three major phases:

1. Concurrent marking

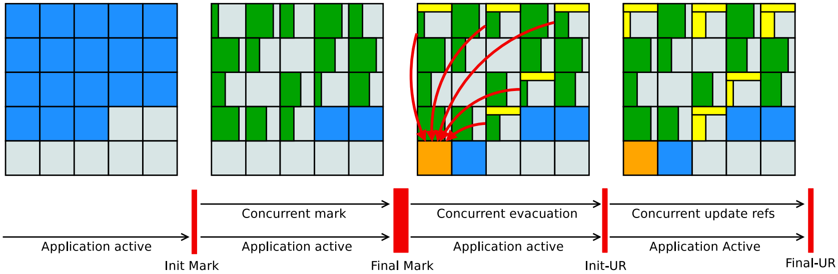
Overview: Usual Cycle



Three major phases:

1. Concurrent marking
2. Concurrent evacuation

Overview: Usual Cycle

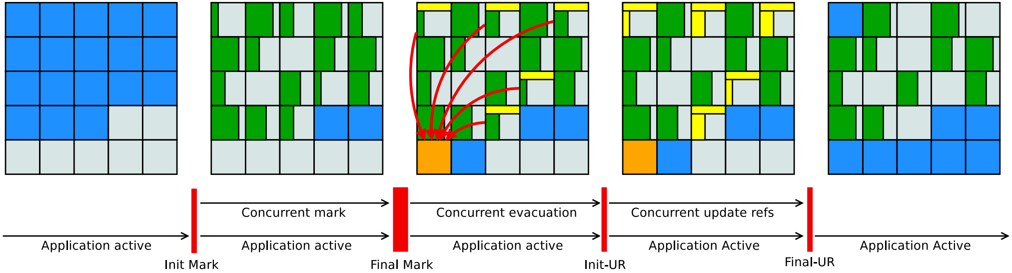


Three major phases:

1. Concurrent marking
2. Concurrent evacuation
3. Concurrent update references (optional)

redhat

Overview: Usual Cycle



Three major phases:

1. Concurrent marking
2. Concurrent evacuation
3. Concurrent update references (optional)

redhat

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, ≈ 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
```


redhat

Phases

Mark: Reachability

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

redhat

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)*

redhat.

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

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

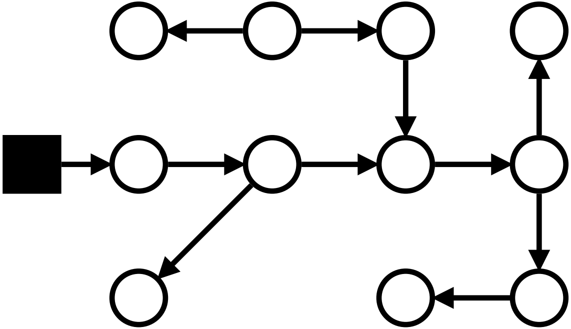
Mark: Three-Color Abstraction

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

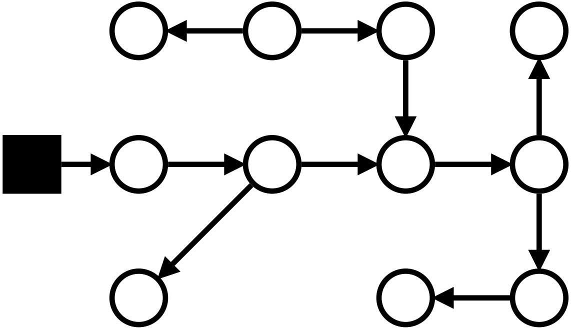
Daily Blues:
«All the marking algorithms do is
coloring white gray, and then coloring gray black»

Mark: Stop-The-World Mark



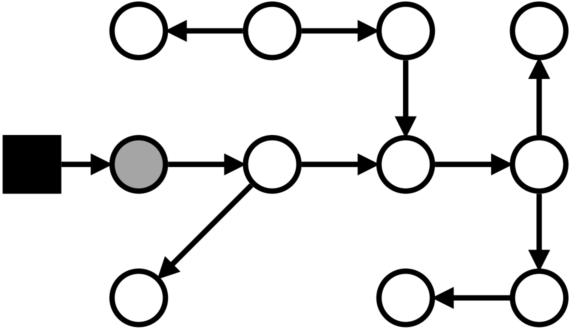
When application is stopped, everything is trivial!
Nothing messes up the scan...

Mark: Stop-The-World Mark



Found all roots, color them Black,
because they are implicitly reachable

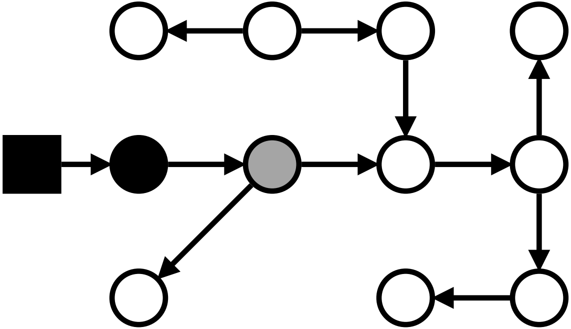
Mark: Stop-The-World Mark



References from Black are now Gray,
scanning Gray references

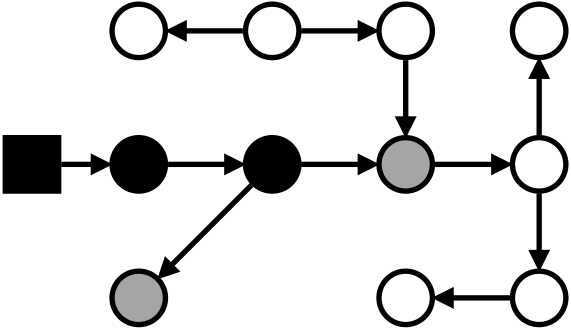
redhat.

Mark: Stop-The-World Mark



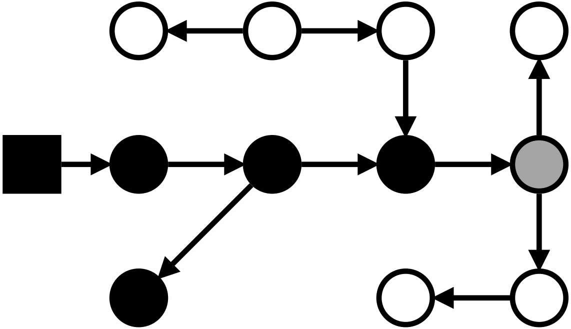
Finished scanning Gray, color them Black;
new references are Gray

Mark: Stop-The-World Mark



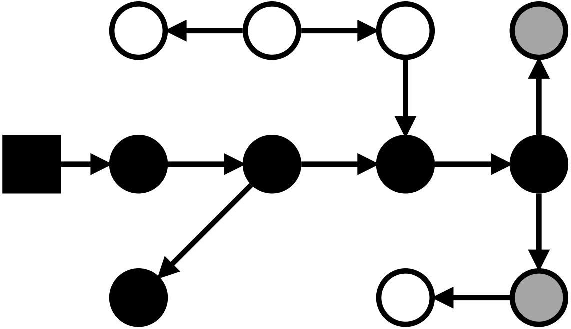
Gray → Black;
reachable from Gray → Gray

Mark: Stop-The-World Mark



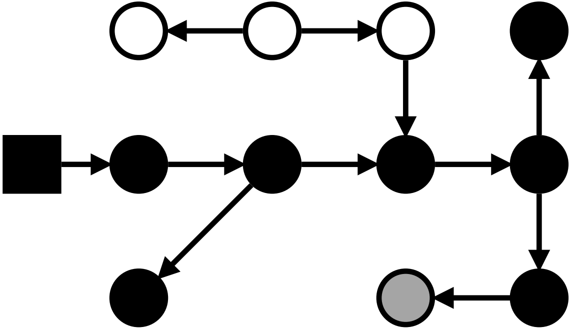
Gray → Black;
reachable from Gray → Gray

Mark: Stop-The-World Mark



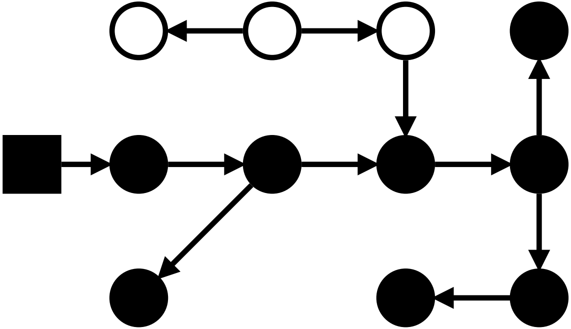
Gray → Black;
reachable from Gray → Gray

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

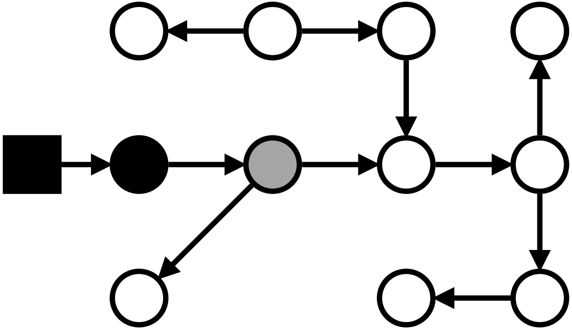
Concurrent Mark: Mutator Problems

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

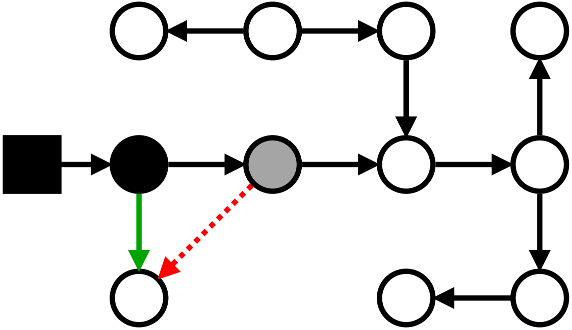
redhat

Concurrent Mark: Mutator Problems



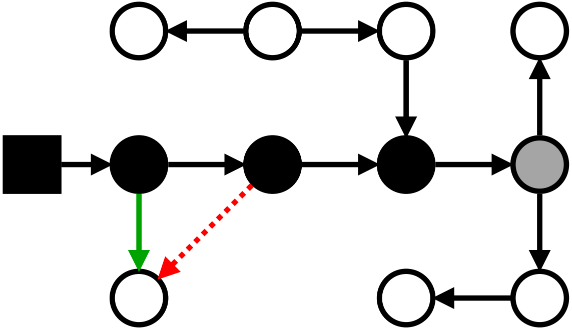
Wavefront is here,
and starts scanning the references in Gray object...

Concurrent Mark: Mutator Problems



Mutator removes the reference from Gray...
and inserts it to Black!

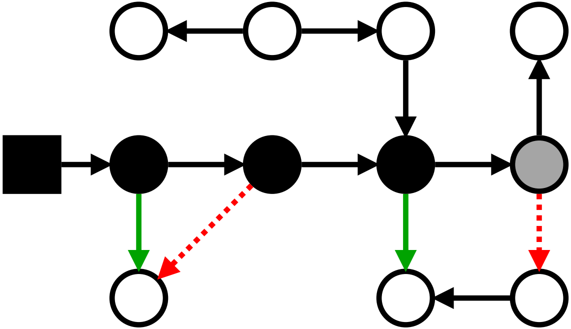
Concurrent Mark: Mutator Problems



...or mutator inserted the reference to
transitively reachable White object into Black

redhat

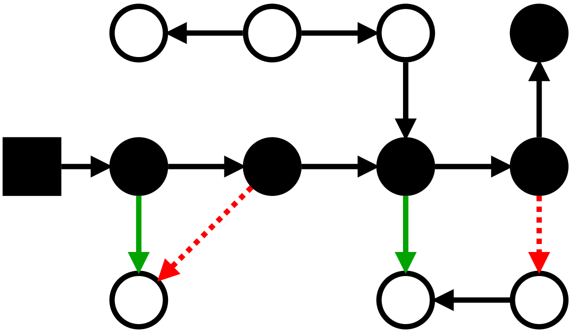
Concurrent Mark: Mutator Problems



...or mutator inserted the reference to
transitively reachable White object into Black

redhat.

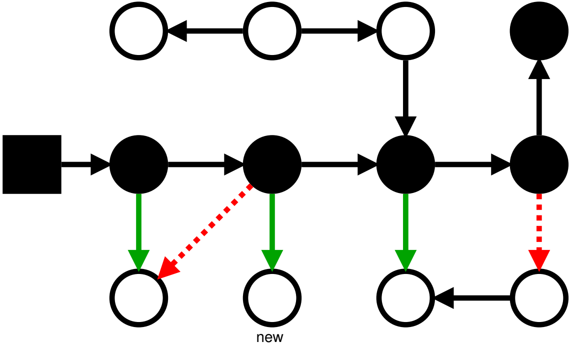
Concurrent Mark: Mutator Problems



Mark had finished, and boom: we have reachable **White** objects, which we will now reclaim, corrupting the heap

redhat.

Concurrent Mark: Mutator Problems

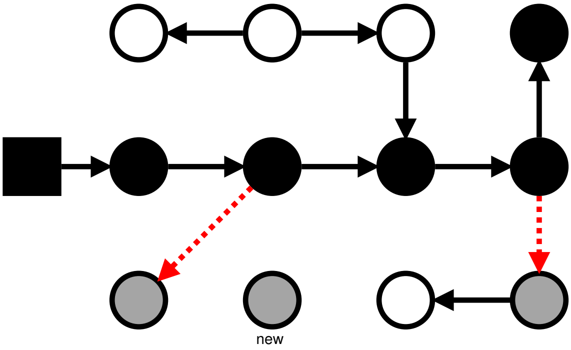


Another quirk: created new **new object**, and inserted it into Black

Concurrent Mark: Textbook Says

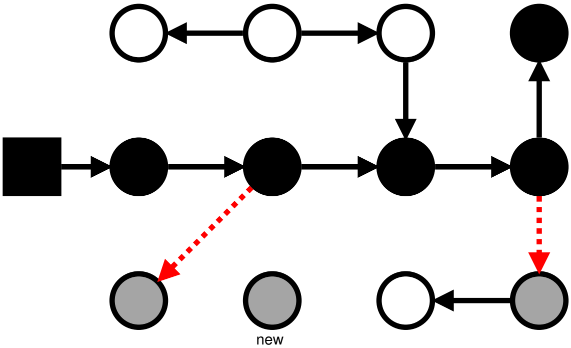
There are at least three approaches to solve this problem. All of them require intercepting heap accesses. Short on time, we shall discuss what G1 and Shenandoah are doing.

Concurrent Mark: SATB



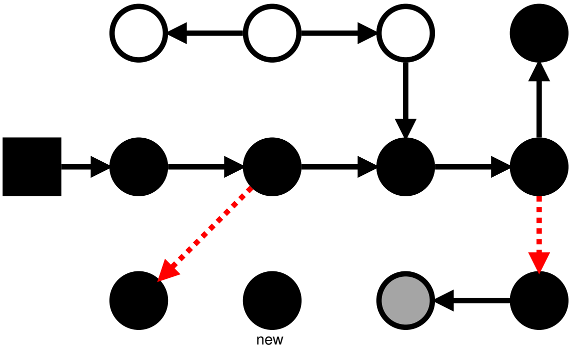
Color all **removed** referents Gray

Concurrent Mark: SATB



Color all new objects **Black**

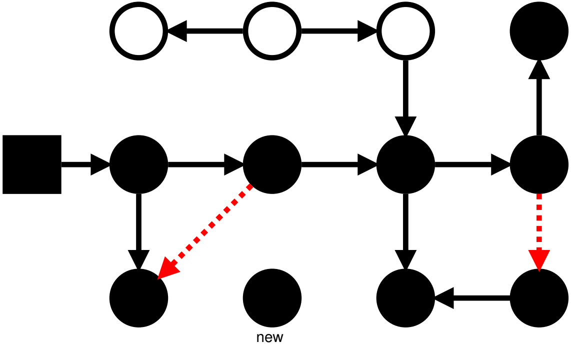
Concurrent Mark: SATB



Finishing...

redhat.

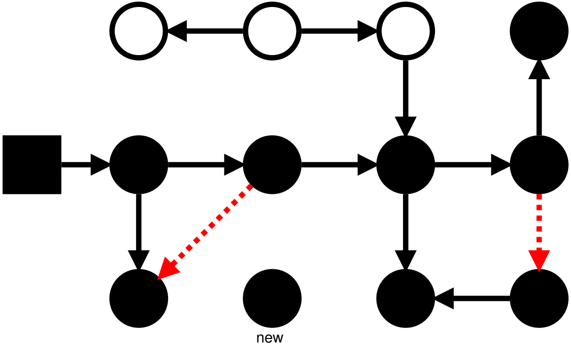
Concurrent Mark: SATB



Done!

redhat.

Concurrent Mark: SATB



«Snapshot At The Beginning»:
marked *all reachable at mark start*

redhat

Concurrent Mark: SATB Barrier

```
  # 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[1]

Init Mark: stop the mutator to avoid races

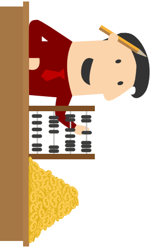
1. Walk and mark all roots ← most heavy-weight
2. Arm SATB barriers

Final Mark: stop the mutator to avoid races

1. Drain the thread buffers
2. Finish work from buffer updates ← most heavy-weight

[1] These can actually be concurrent, but that is not very practical

Concurrent Mark: Barriers Cost[2]

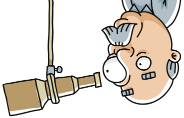


	Throughput hit, %
	SATB
Cmp	-1.6
Cps	-3.5
Cry	
Der	-1.6
Mpg	
Smk	
Ser	
Sfl	
Xml	-3.1

[2] Performance compared to STW Shenandoah with all barriers disabled

redhat.

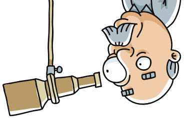
Concurrent Mark: Observations



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

redhat.

Concurrent Mark: Observations

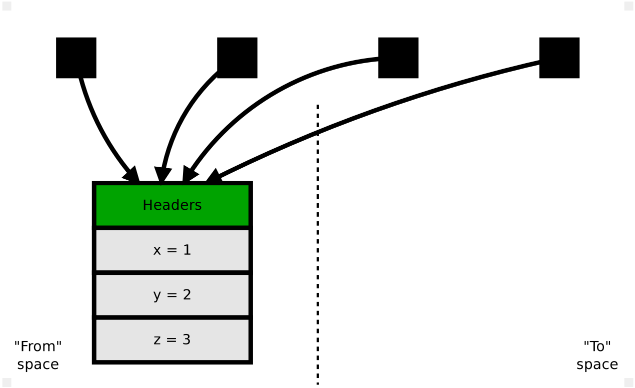


1. Extended concurrency needs to pay with more barriers
   - Ideal STW GC beats ideal concurrent GC on pure throughput
   - If you do not care about GC pauses, just use good STW GC
   - Empty GC log does not mean no GC overhead

2. Hiding references from mark prolongs final mark pause
   - Weak references with unreachable referents, **finalizers**
   - «Old» objects hidden in SATB buffers

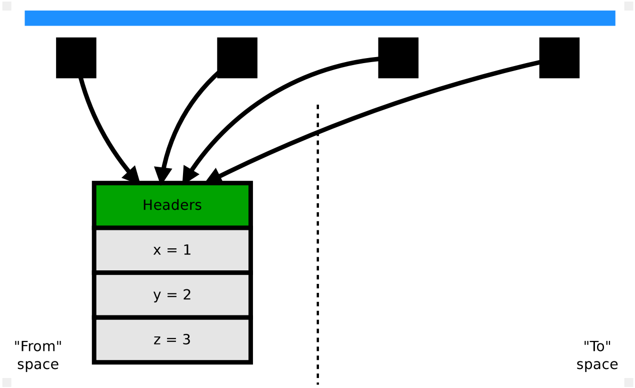
redhat.

Copy: Stop-The-World



Problem:
there is the object, the object is referenced from somewhere, need to move it to new location

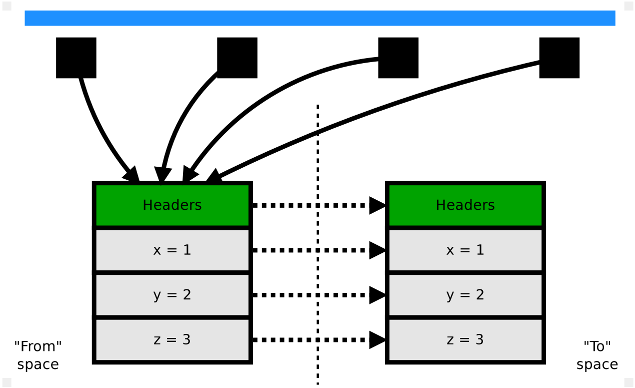
Copy: Stop-The-World



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

redhat

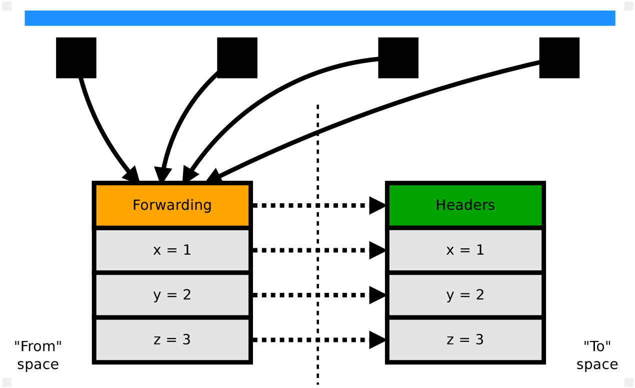
Copy: Stop-The-World



Step 2:
Copy the object with all its contents

redhat.

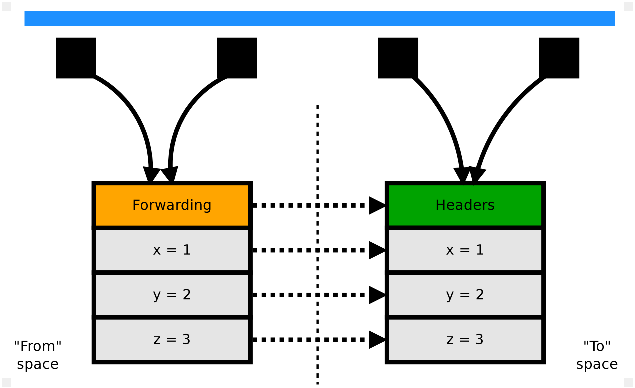
Copy: Stop-The-World



Step 3.1:
Update all references:
save the pointer that
forwards to the copy

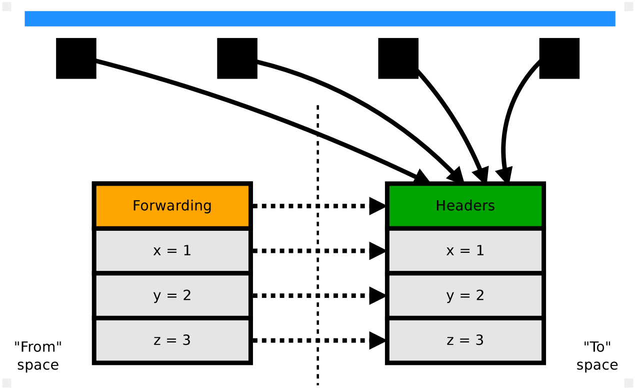
redhat.

Copy: Stop-The-World



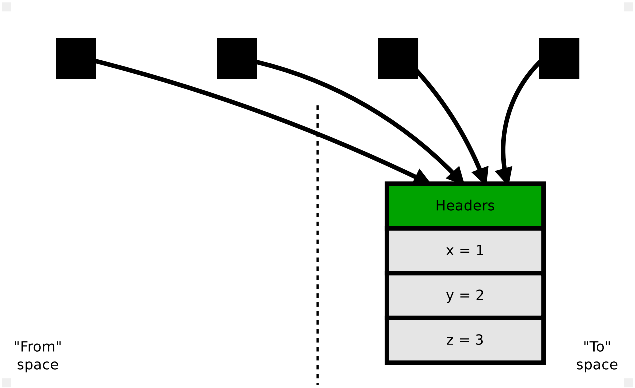
Step 3.2:
Update all references: walk the heap, replace all refs with fwdptr destination

Copy: Stop-The-World



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!

redhat.

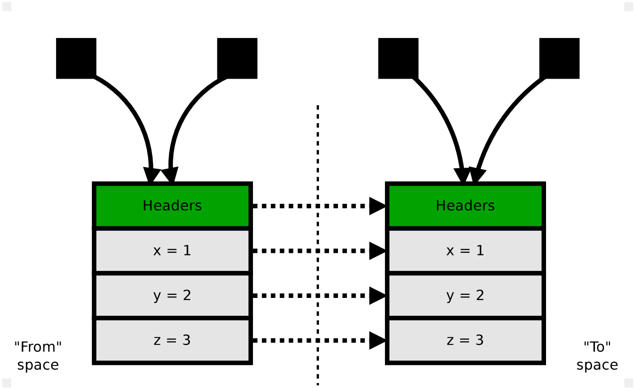
Concurrent Copy: Mutator Problems

http://vernova-dasha.livejournal.com/77066.html

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

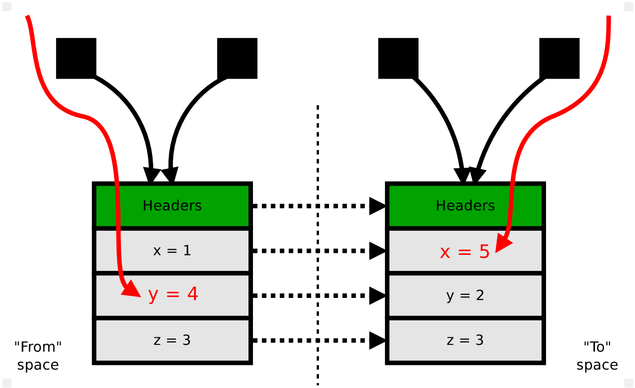
redhat.

Concurrent Copy: Mutator Problems



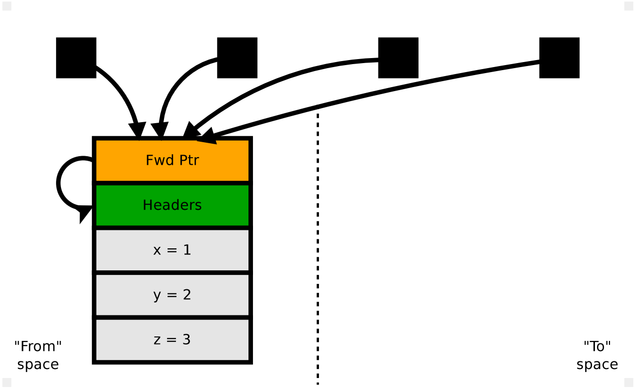
While object is being moved, there are *two* copies of the object, and both are reachable!

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?

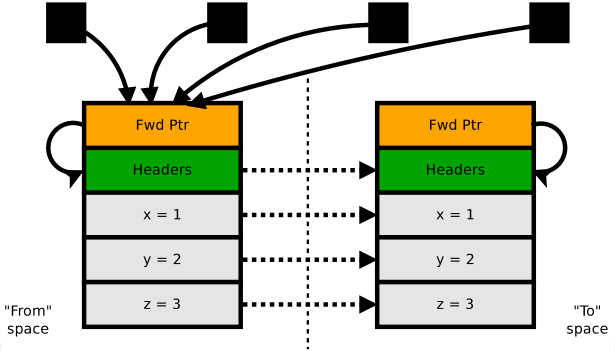
Concurrent Copy: Brooks Pointers



Idea:
Brooks pointer: object version change with additional atomically changed indirection

redhat.

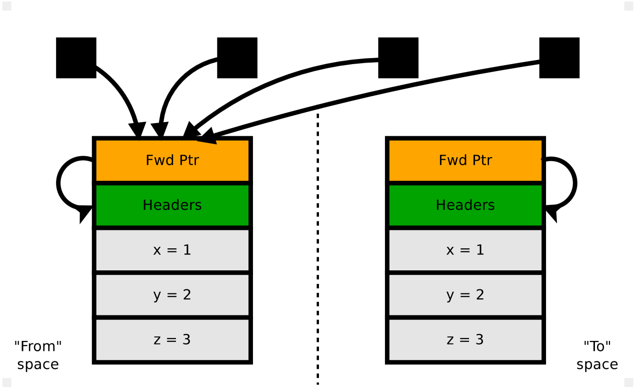
Concurrent Copy: Brooks Pointers



Step 1:
Copy the object, initialize its forwarding pointer to self

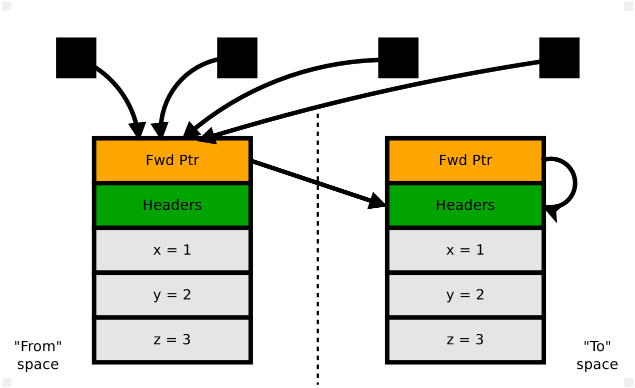
redhat.

Concurrent Copy: Brooks Pointers



We now have the copy of the object, but no one knows about it

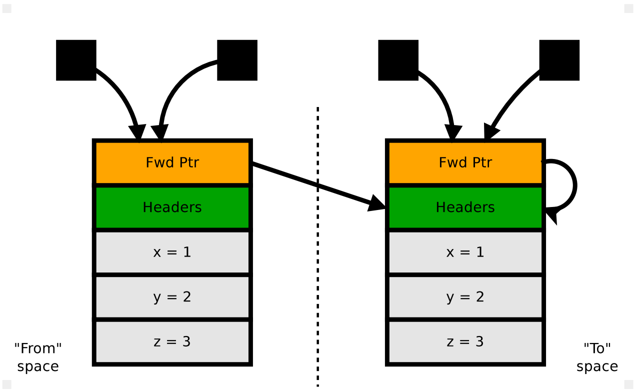
Concurrent Copy: Brooks Pointers



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

redhat

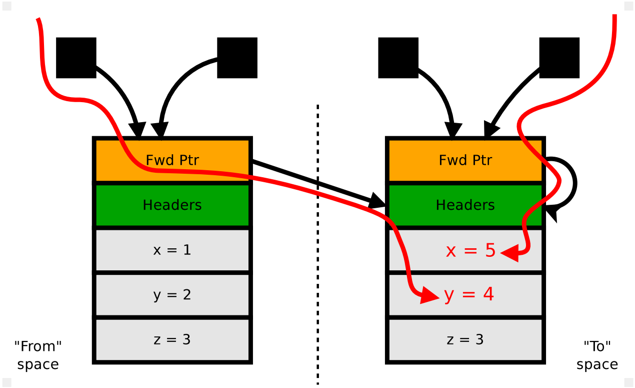
Concurrent Copy: Brooks Pointers



Step 3:
Rewrite the references at our own pace in the rest of the heap

redhat.

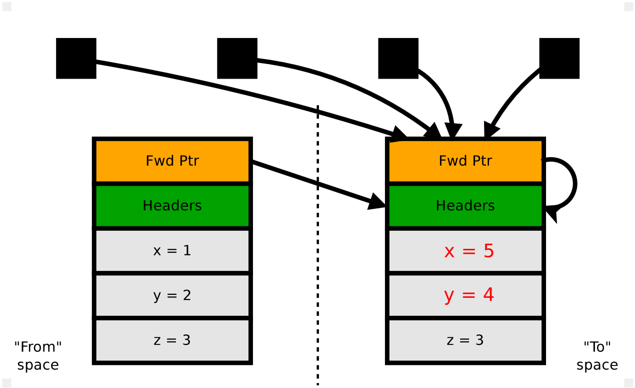
Concurrent Copy: Brooks Pointers



If somebody reaches the old copy via the old reference, it has to dereference via fwdptr and discover the actual object copy!

redhat.

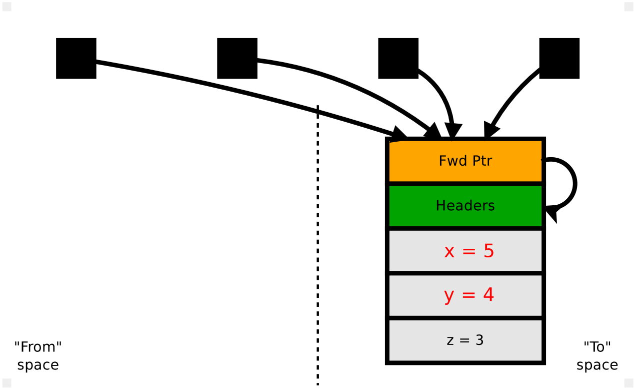
Concurrent Copy: Brooks Pointers



Step 4:
All references are updated, recycle the from-space copy

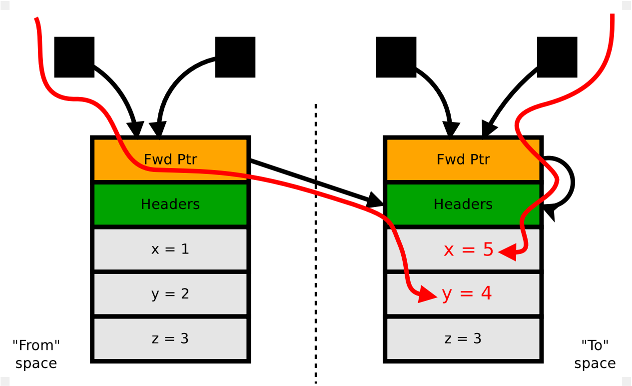
redhat

Concurrent Copy: Brooks Pointers



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

```
  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
      fwd-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
    }
  }
}
```

redhat.

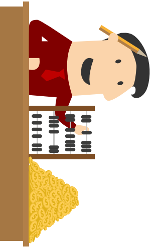
Write Barriers: GC Evacuation Code

```
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.

redhat.

Write Barriers: Barriers Cost[2]

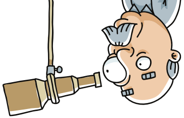


	Throughput hit, %	
	SATB	WB
Cmp	-1.6	-3.5
Cps	-3.5	
Cry		-1.1
Der	-1.6	
Mpg		-2.1
Smk		-0.5
Ser		-4.0
Sfl		-2.7
Xml	-3.1	-3.5

[2] Performance compared to STW Shenandoah with all barriers disabled

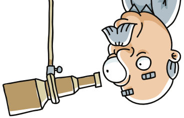
redhat.

Write Barriers: Observations



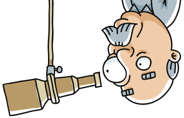
1. Shenandoah needs WB on **all** stores
   - Field stores – obviously
   - Locking the object – changes header ⇒ needs WB
   - Computing identity hash code – changes header ⇒ needs WB

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

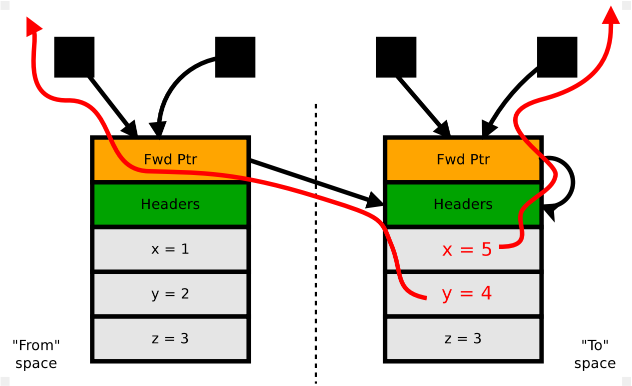
Write Barriers: Observations



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

redhat.

Read Barriers: Motivation



Heap reads have to (?) dereference via the forwarding pointer, to discover the actual object copy

Read Barriers: Implementation

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

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

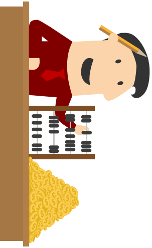
Read Barriers: Implementation

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

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

Benchmark	Score		Units
	base	+3 RBs	
time	4.6 ± 0.1	5.3 ± 0.1	ns/op
L1-dcache-loads	12.3 ± 0.2	15.1 ± 0.3	#/op
cycles	18.7 ± 0.3	21.6 ± 0.3	#/op
instructions	26.6 ± 0.2	30.3 ± 0.3	#/op

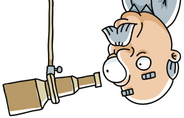
Read Barriers: Barriers Cost[2]



	Throughput hit, %		
	SATB	WB	RB
Cmp	-1.6	-3.5	-7.7
Cps	-3.5		-11.4
Cry		-1.1	
Der	-1.6		-7.4
Mpg		-2.1	-12.4
Smk		-0.5	-4.9
Ser		-4.0	-7.1
Sfl		-2.7	-6.7
Xml	-3.1	-3.5	-9.5

[2] Performance compared to STW Shenandoah with all barriers disabled

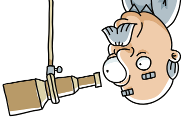
Read Barriers: Observations



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

redhat.

Read Barriers: Observations

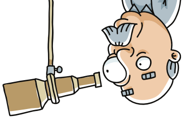


1. Shenandoah needs RBs before **most** loads
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains

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

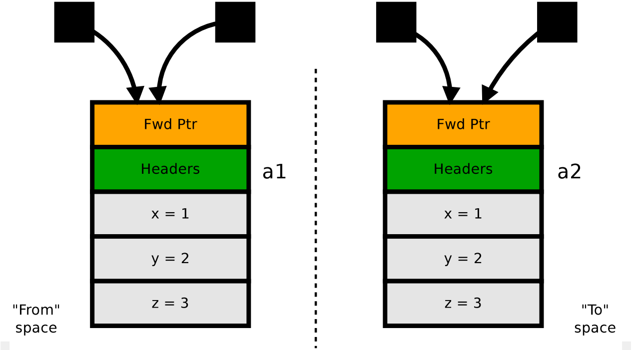
redhat.

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

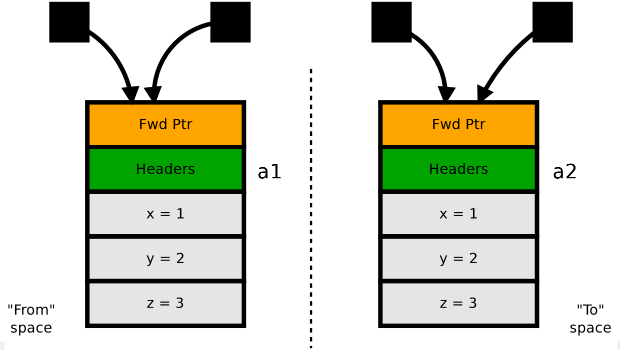
CMP: Trouble



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

(a1 == a2) → ???

CMP: Trouble



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

(a1 == a2) → ???

But *machine ptrs* are not equal... Oops.

redhat.

CMP: Exotic Barriers

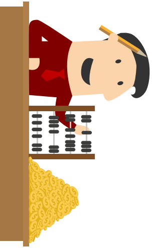
Having two *physical* copies of the same *logical* object,
«==» has to compare *logical* objects

```
# 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) ...
```

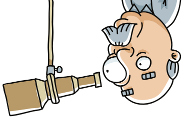
CMP: Barriers Cost[2]



	Throughput hit, %			
	SATB	WB	RB	CMP*
Cmp	-1.6	-3.5	-7.7	
Cps	-3.5		-11.4	
Cry		-1.1		
Der	-1.6		-7.4	
Mpg		-2.1	-12.4	
Smk		-0.5	-4.9	
Ser		-4.0	-7.1	
Sfl		-2.7	-6.7	
Xml	-3.1	-3.5	-9.5	

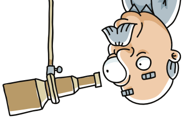
[2] Performance compared to STW Shenandoah with all barriers disabled

CMP: Observations



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

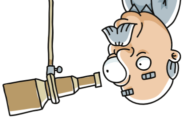
CMP: Observations



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



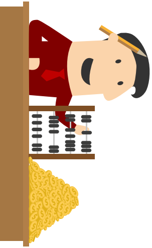
1. Shenandoah needs to handle ref comparisons specially
   - Cannot make RBs much heavier
   - Optimizing compilers move and coalesce RB – massive gains

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

redhat

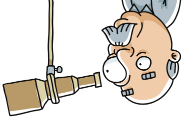
Overall: Barriers Cost[2]



	Throughput hit, %				
	SATB	WB	RB	CMP*	TOTAL
Cmp	-1.6	-3.5	-7.7		-14.3
Cps	-3.5		-11.4		-13.7
Cry		-1.1			-4.3
Der	-1.6		-7.4		-9.3
Mpg		-2.1	-12.4		-14.8
Smk		-0.5	-4.9		-2.6
Ser		-4.0	-7.1		-11.1
Sfl		-2.7	-6.7		-11.3
Xml	-3.1	-3.5	-9.5		-15.6

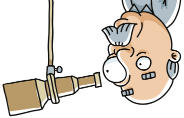
[2] Performance compared to STW Shenandoah with all barriers disabled

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)

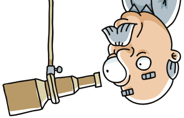
Overall: Observations



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

redhat.

Overall: Observations

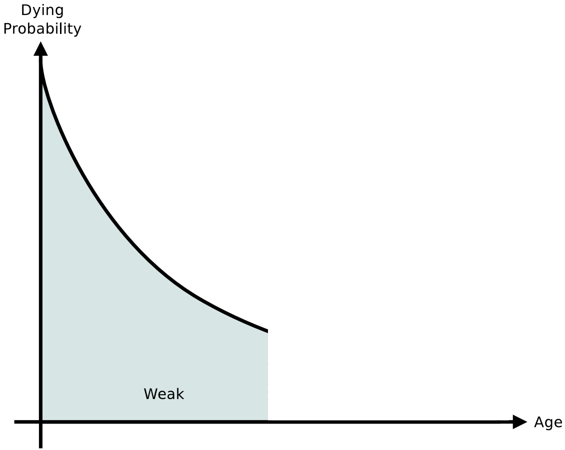


1. Easily portable across HW architectures
   - Special needs: `CAS` (performance is important, but not critical)
   - x86_64 and AArch64 are major implemented targets
   - Theoretically works with 32-bit arches (but not ported yet)
2. Trivially portable across OSes
   - Special needs: none
   - Linux 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

redhat.

Intermezzo

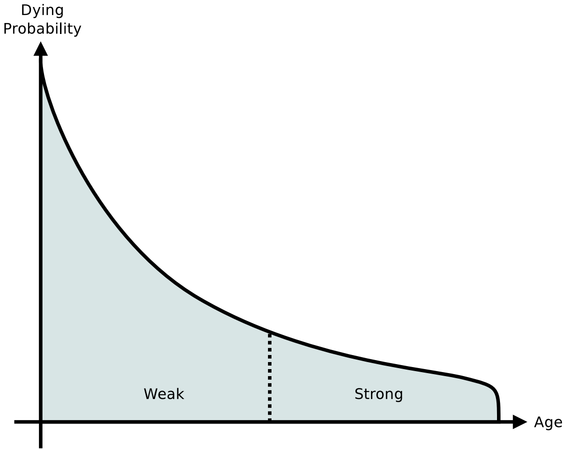
Intermezzo: Generational Hypotheses



Weak hypothesis:
most objects die young

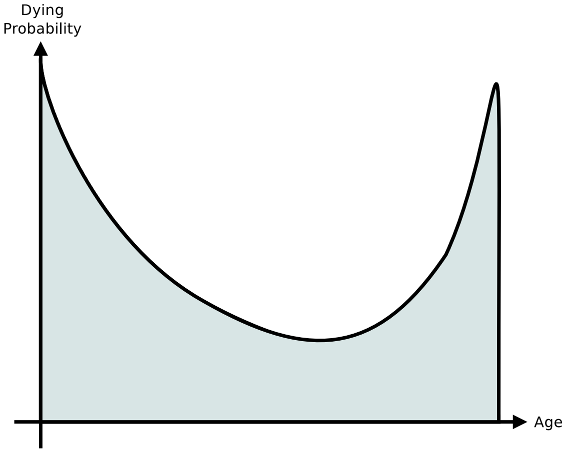
redhat.

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

redhat.

Intermezzo: The Simplest LRU

The simplest LRU implementation in Java?

redhat.

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

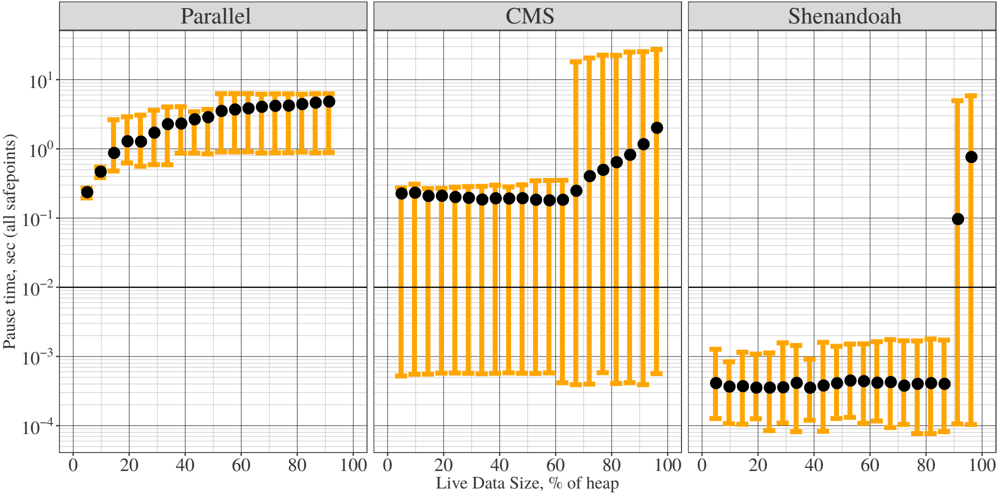
Boring config:

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`

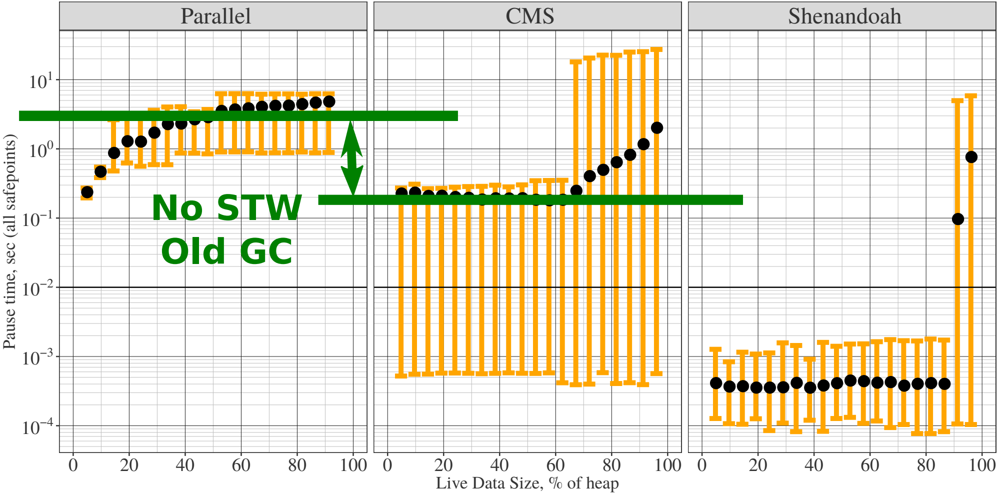
Varying cache size $\Rightarrow$ varying LDS $\Rightarrow$ make GC uncomfortable

redhat

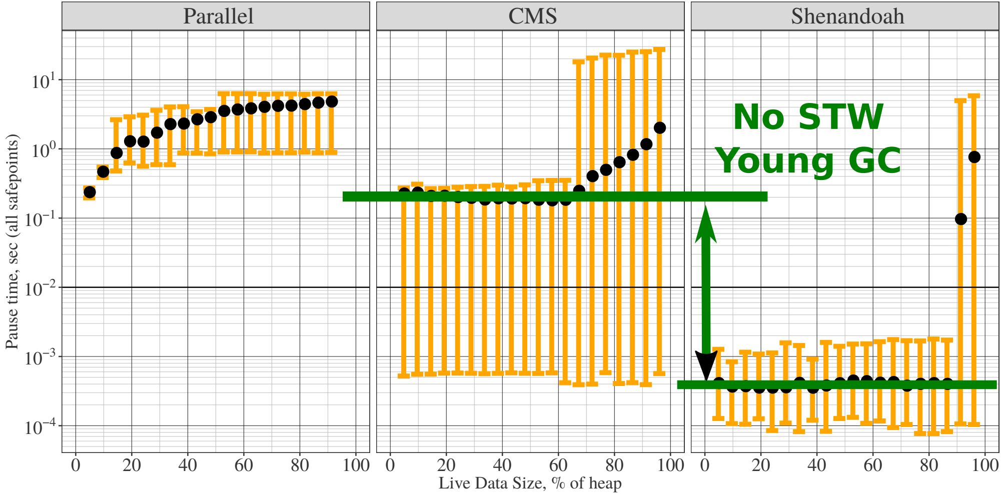
Intermezzo: Pauses vs. LDS



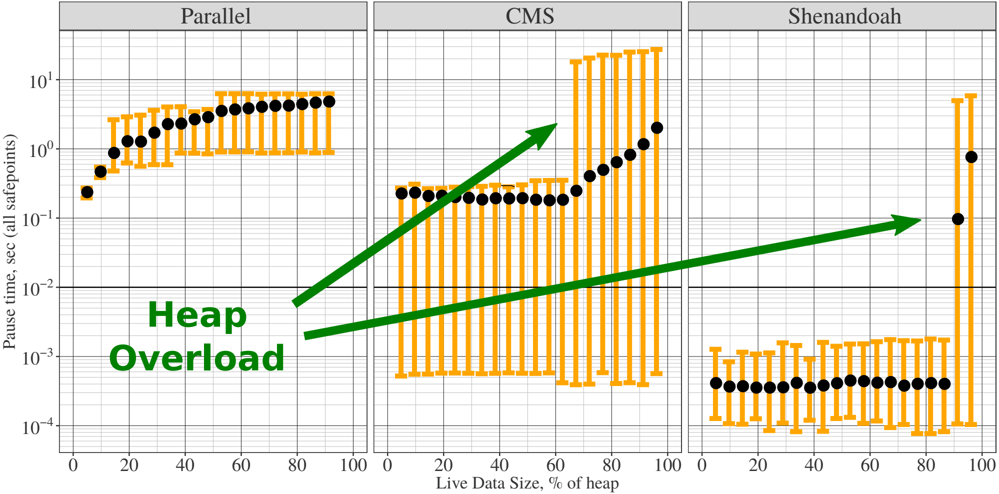
Intermezzo: Pauses vs. LDS



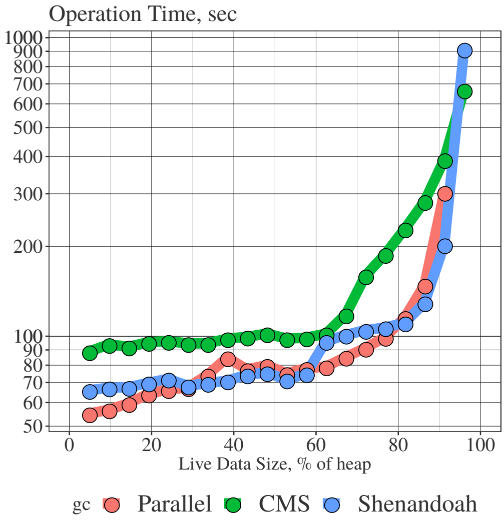
Intermezzo: Pauses vs. LDS

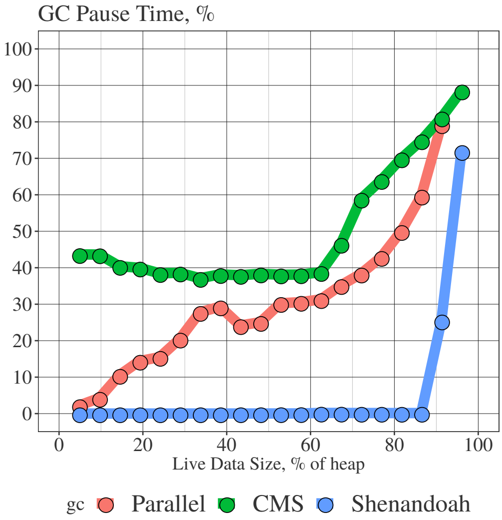


Intermezzo: Pauses vs. LDS



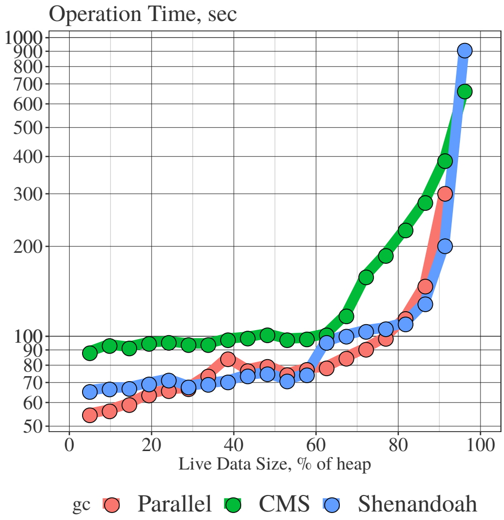
Intermezzo: Perf vs. LDS

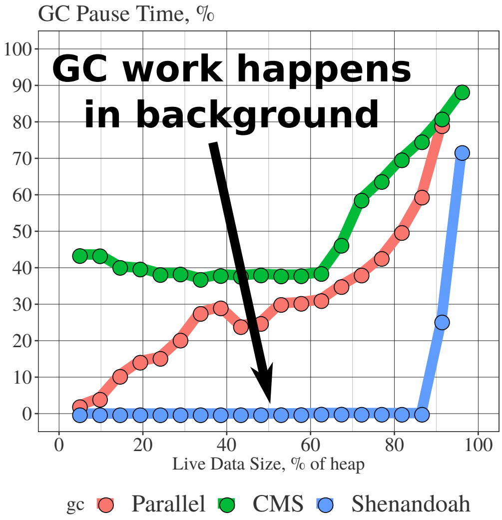




redhat

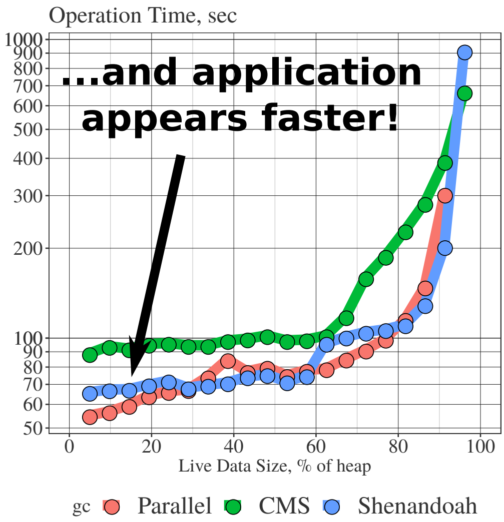
Intermezzo: Perf vs. LDS

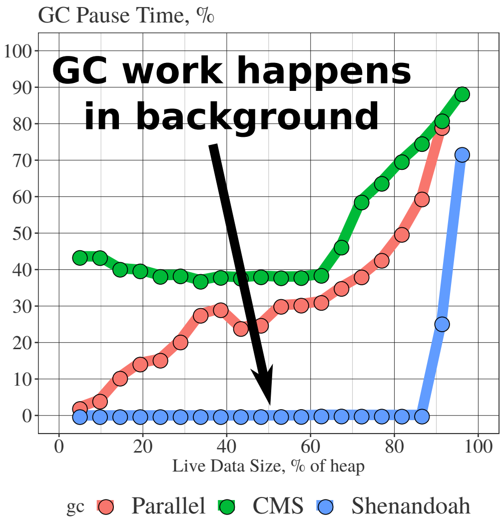




redhat.

Intermezzo: Perf vs. LDS





Command and Control

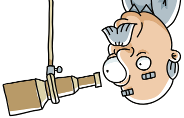
Command and Control: Central Dogma

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

redhat.

Command and Control: Off To The Races



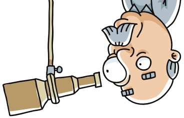
```
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is
above the time for allocation rate (3254.90 MB/s) to
deplete free headroom (13071M)
```

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

redhat.

Command and Control: Off To The Races

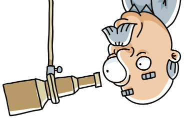


```
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is
above the time for allocation rate (3254.90 MB/s) to
deplete free headroom (13071M)
```

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: Off To The Races



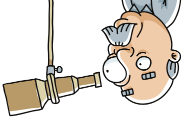
```
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is
above the time for allocation rate (3254.90 MB/s) to
deplete free headroom (13071M)
```

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

redhat.

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

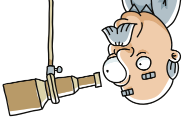
Problem:

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

Things that help:

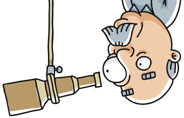
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade

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
```

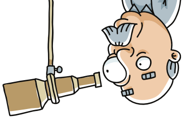
Immediates: Obvious Shortcut



```
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

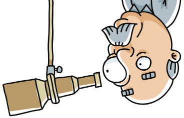
Immediates: Obvious Shortcut



```
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...

redhat

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

redhat.

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

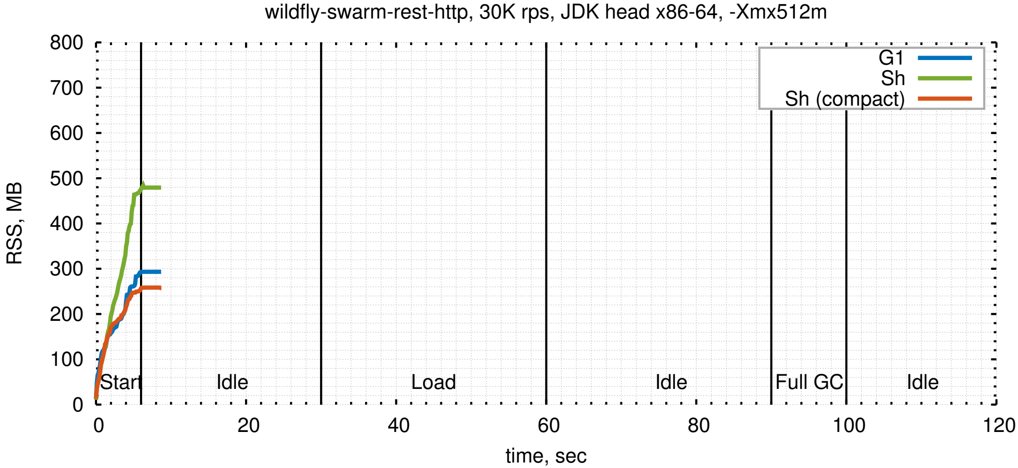
redhat.

Footprint: Shenandoah Overheads

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

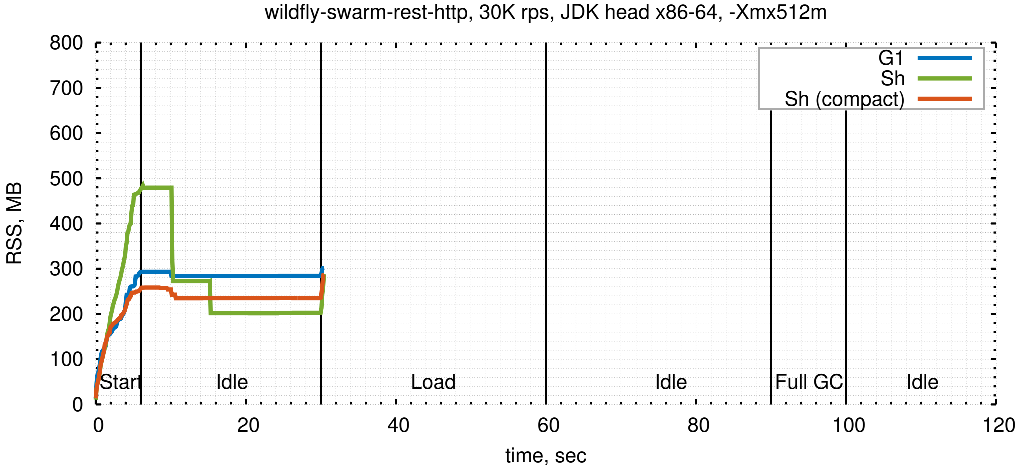
- Java heap: 1.5x worst and 1.05-1.10x avg overhead
  «−»: the overhead is non-static
  «+»: counted in Java heap – no surprise RSS inflation
- Native structures: 2x marking bitmaps, each 1/64 of heap
  «−»: -Xmx is still not close to RSS
  «+»: overhead is static: `-Xmx100g` means 103 GB RSS
- Surprise: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

Footprint: Heap Sizing

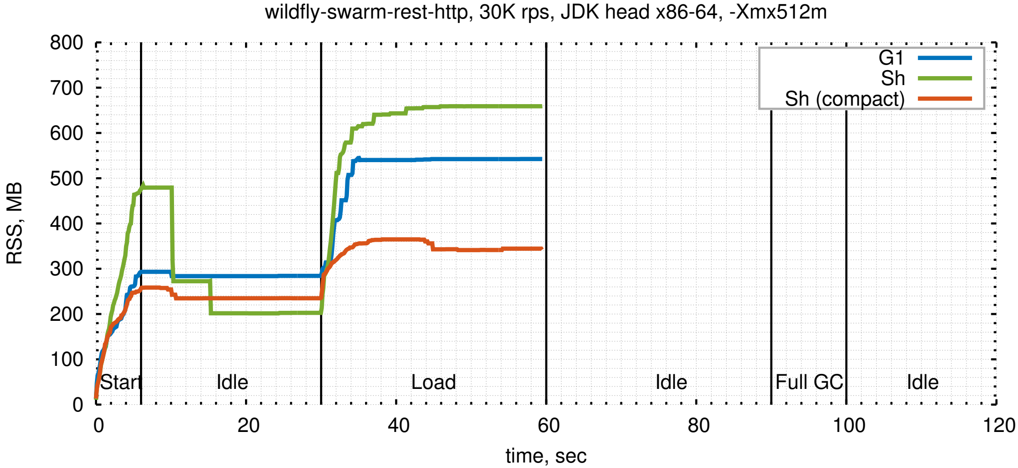


redhat.

Footprint: Heap Sizing

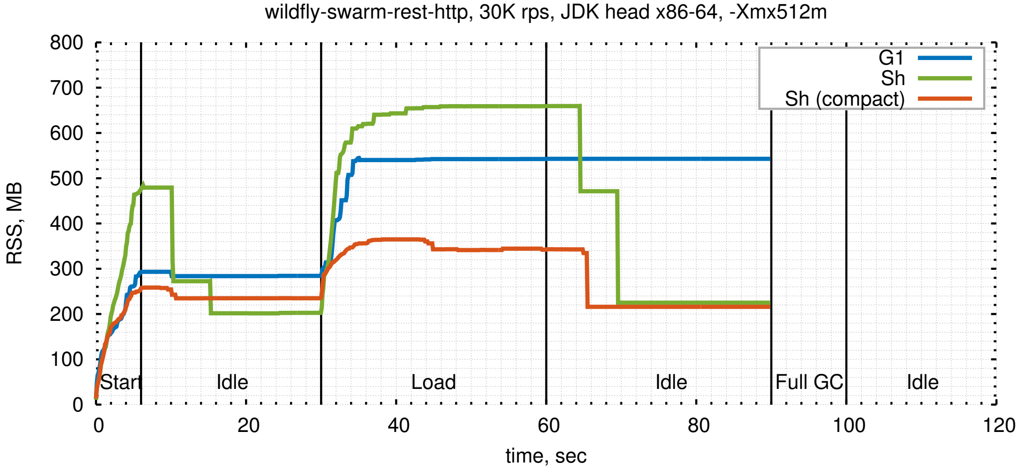


Footprint: Heap Sizing

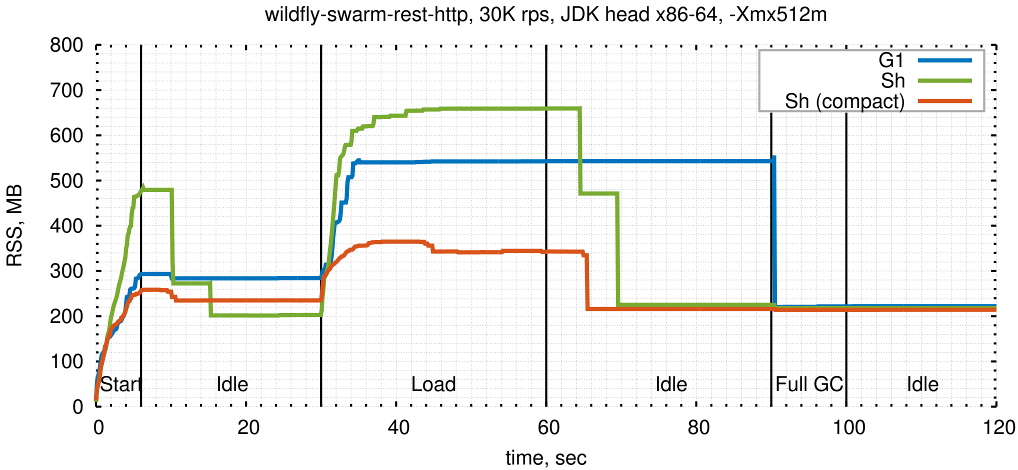


redhat.

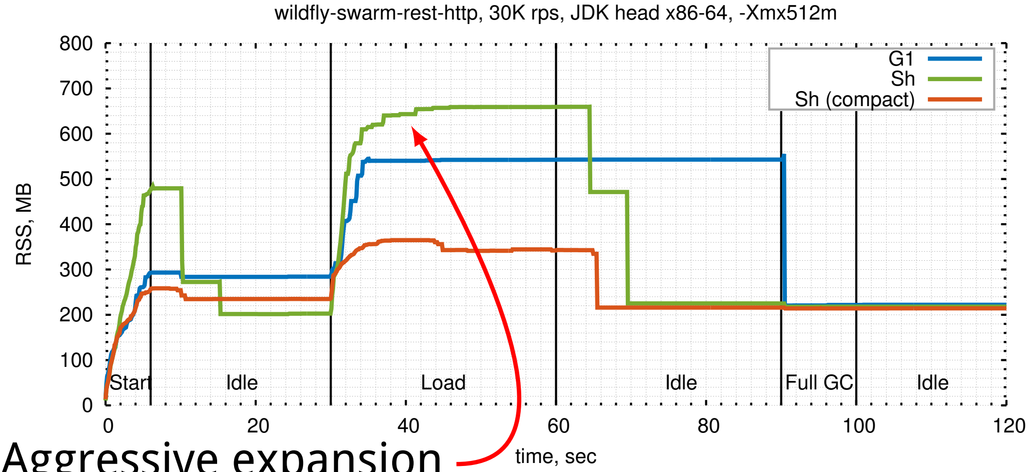
Footprint: Heap Sizing



Footprint: Heap Sizing



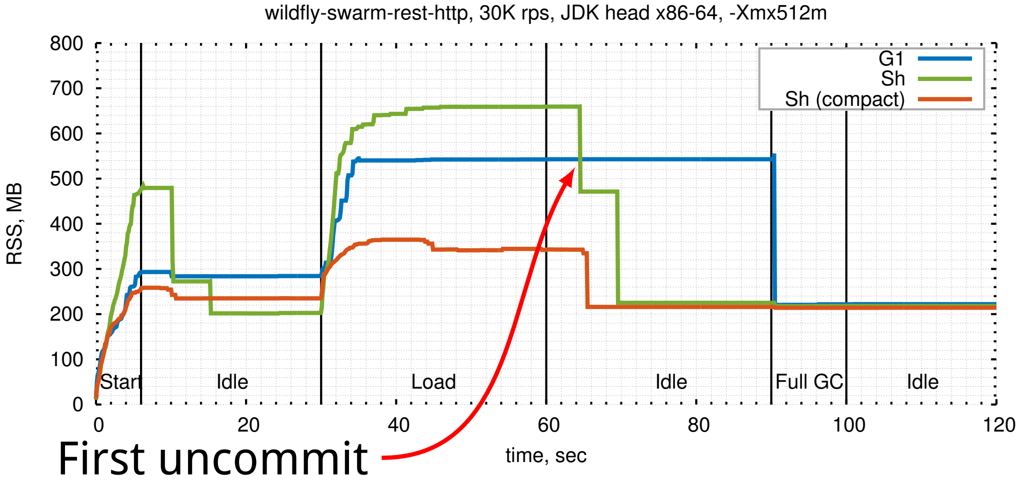
Footprint: Heap Sizing



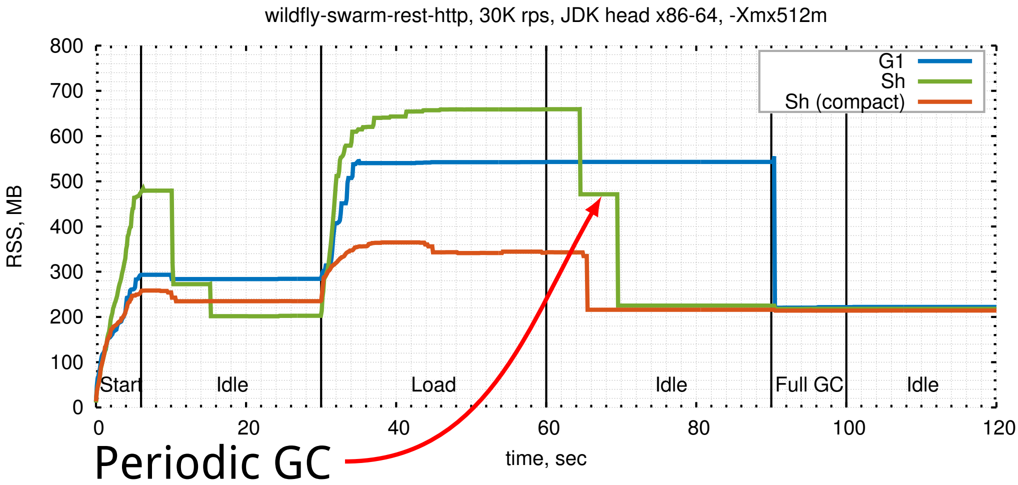
Aggressive expansion

redhat.

Footprint: Heap Sizing

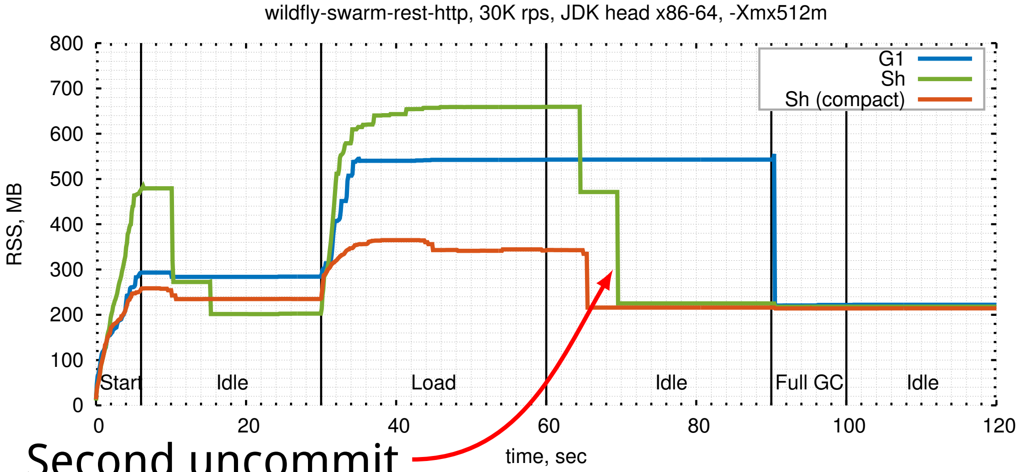


Footprint: Heap Sizing



redhat.

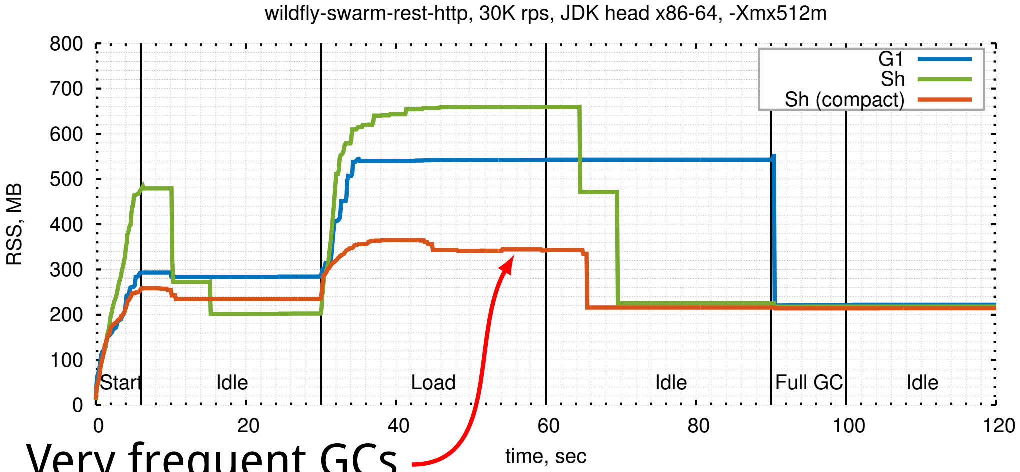
Footprint: Heap Sizing



Second uncommit

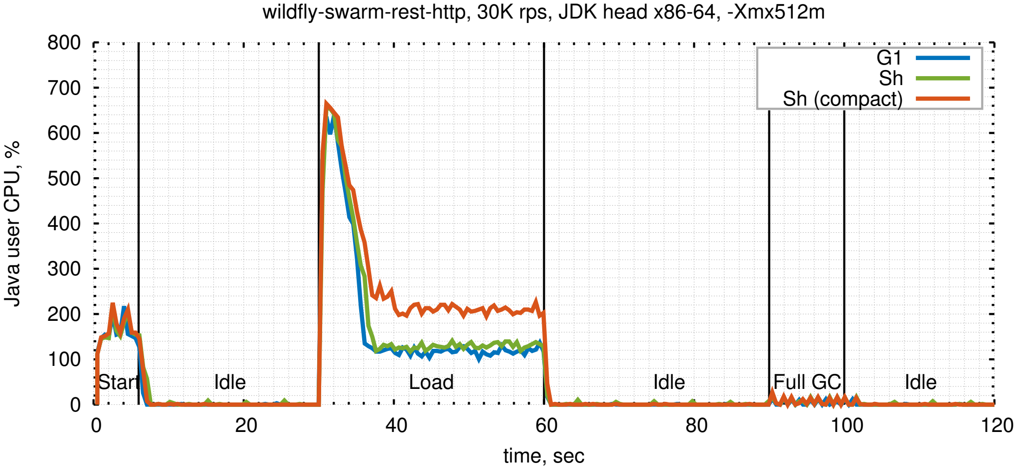
redhat.

Footprint: Heap Sizing



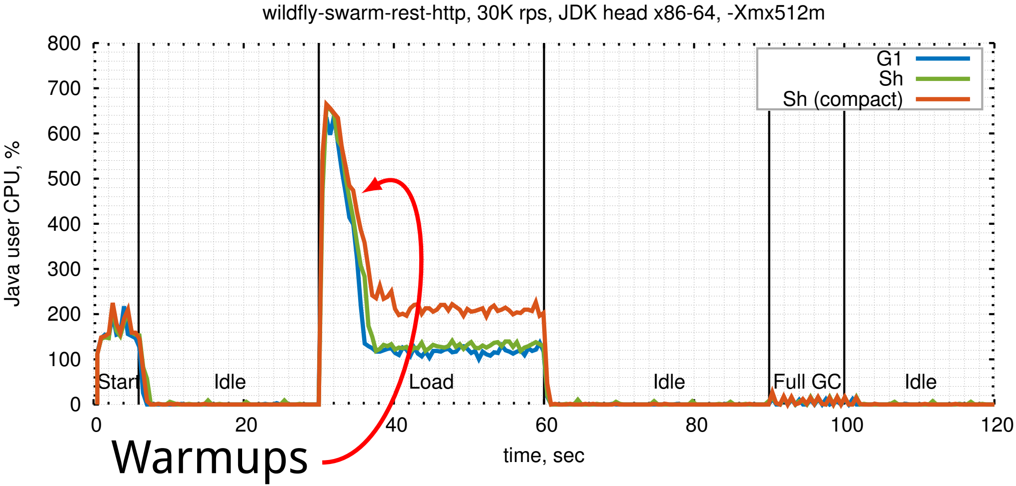
Very frequent GCs

Footprint: CPU Time Tradeoffs



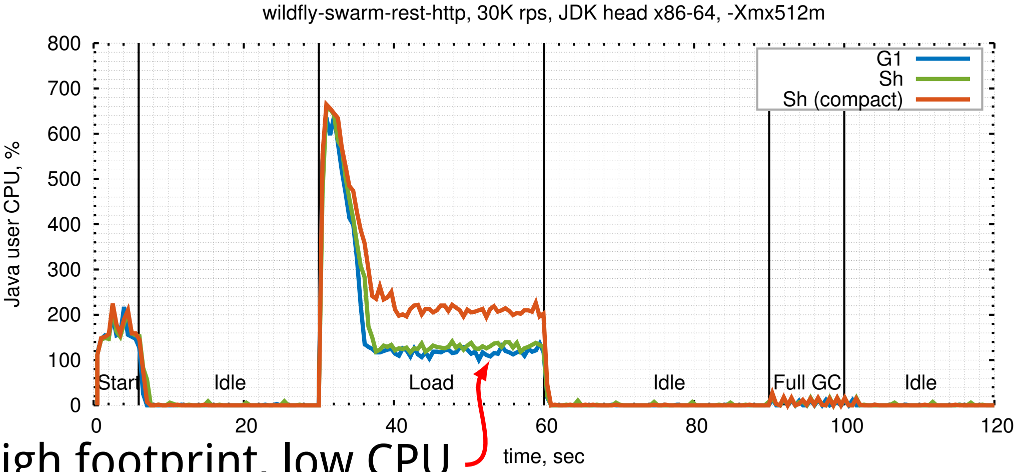
redhat.

Footprint: CPU Time Tradeoffs



redhat.

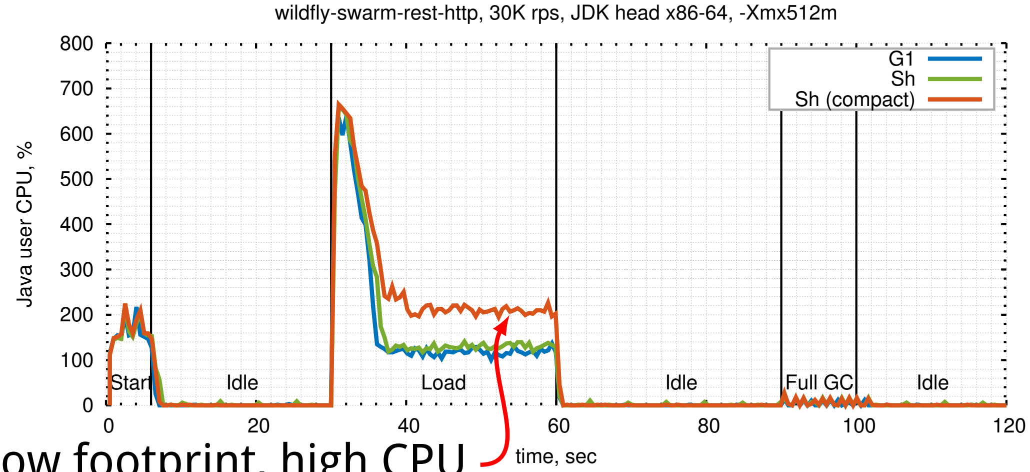
Footprint: CPU Time Tradeoffs



High footprint, low CPU

redhat.

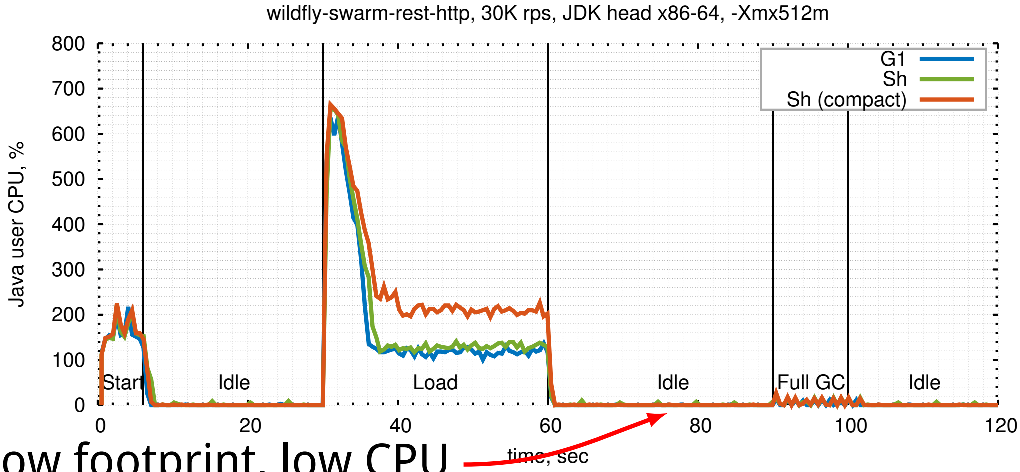
Footprint: CPU Time Tradeoffs



Low footprint, high CPU

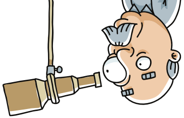
redhat.

Footprint: CPU Time Tradeoffs



Low footprint, low CPU

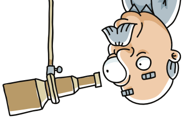
Footprint: Observations



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

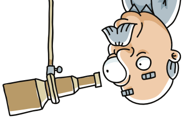
redhat.

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

redhat.

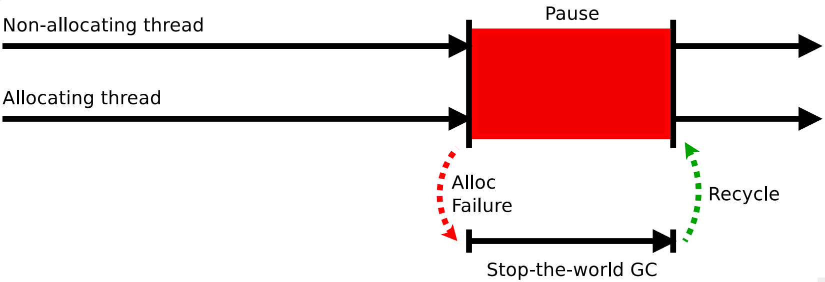
Pacing: Living Space

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

Things that help:

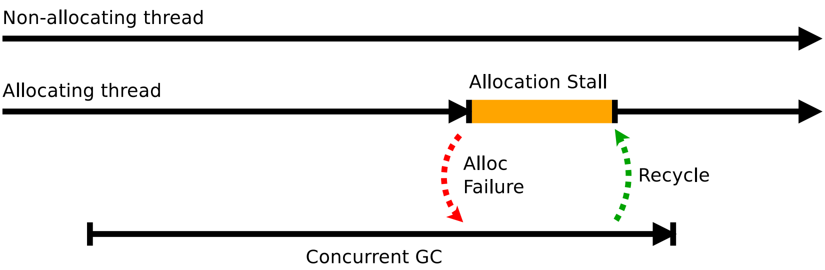
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade

Pacing: STW GC Control Loop



- Once memory is exhausted, perform GC
- Natural feedback loop: STW is the nominal mode
- Not really accessible for concurrent GC?

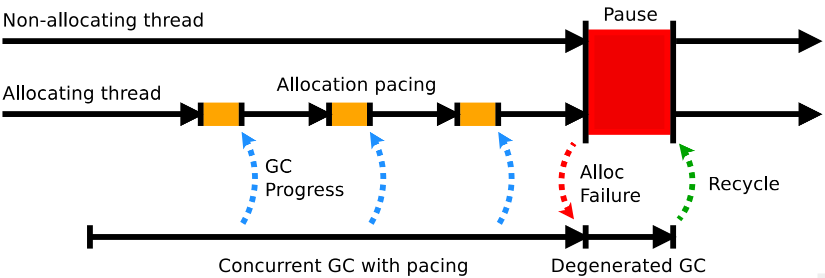
Pacing: Naive Conc GC Control Loop



- 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*

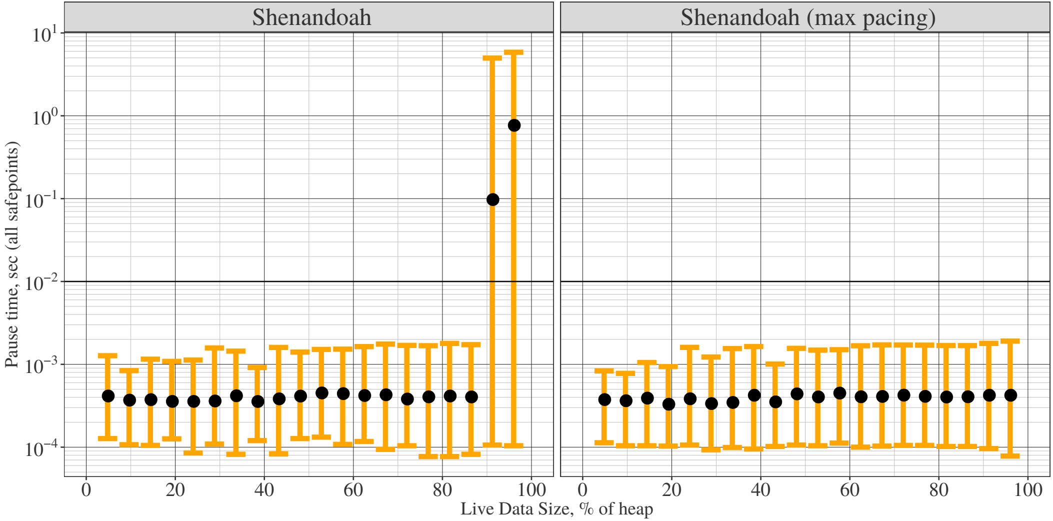
redhat.

Pacing: Shenandoah Control Loop



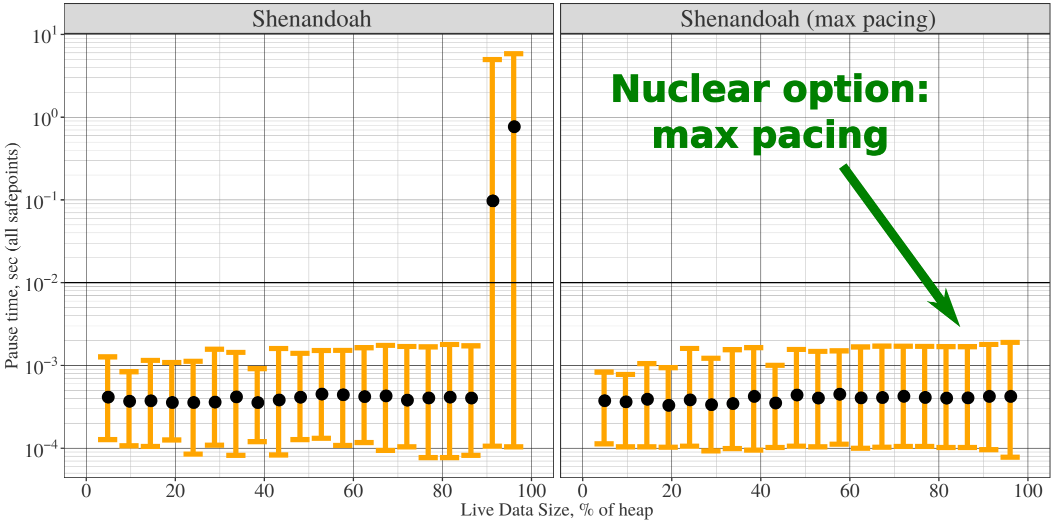
- Incremental pacing stalls allocations a bit at a time
- If AF happens, «degenerates»: completes under STW
- Pacing introduces latency, but the capped one

Pacing: Max Pacing, Pauses



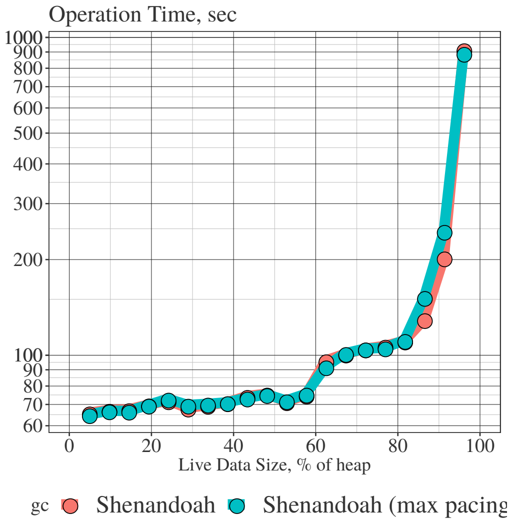
redhat.

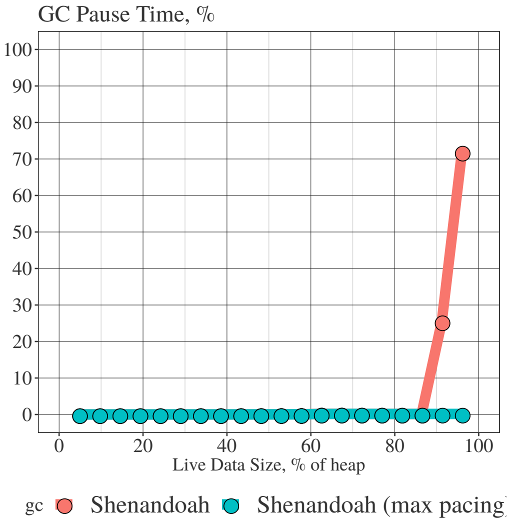
Pacing: Max Pacing, Pauses



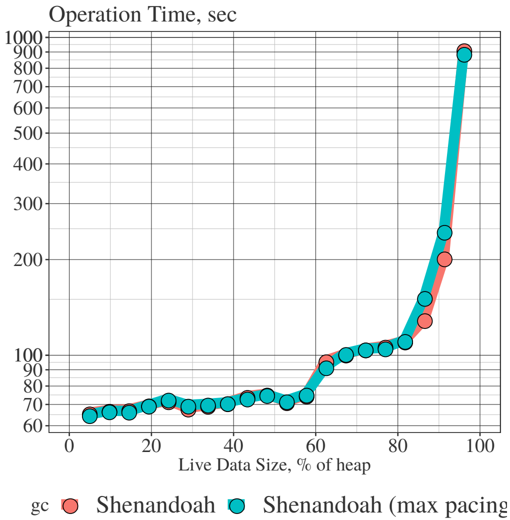
redhat.

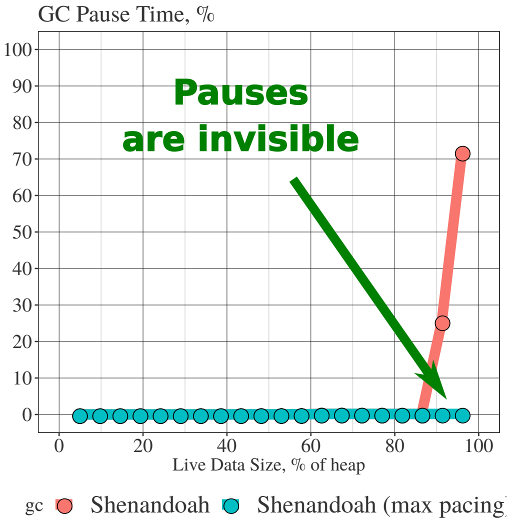
Pacing: Max Pacing, Times



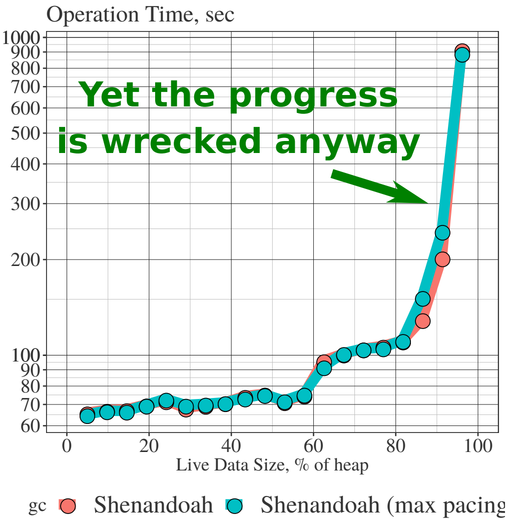


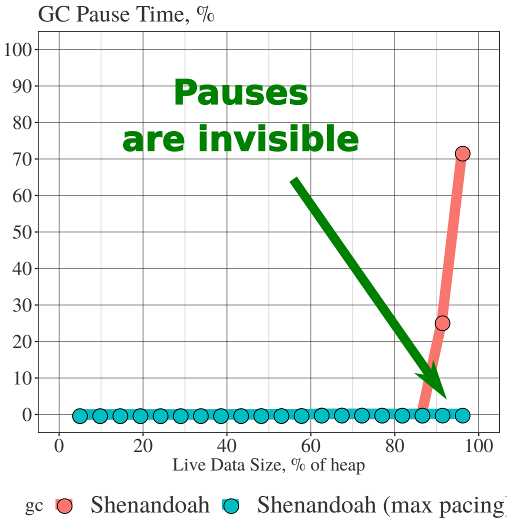
Pacing: Max Pacing, Times



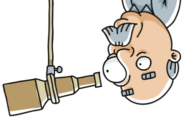


Pacing: Max Pacing, Times





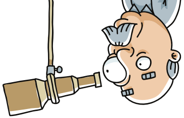
Pacing: Observations



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

redhat.

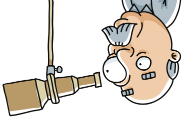
Pacing: Observations



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

redhat.

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

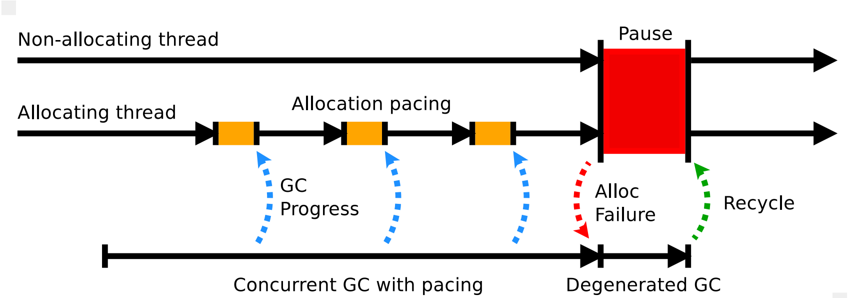
Problem:
Concurrent GC needs breathing room to succeed,
while applications allocate like madmen

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

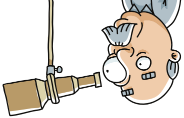
redhat.

Handling Failures: Shenandoah Control Loop



- If AF happens, «degenerates»: completes under STW

Handling Failures: Degenerated GC

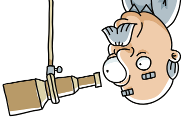


```
Pause Init Update Refs 0.034ms
Cancelling GC: Allocation Failure
Concurrent update references 7265M->8126M(8192M) 248.467ms
Pause Degenerated GC (Update Refs) 8126M->2716M(8192M) 29.787ms
```

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

redhat.

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

redhat.

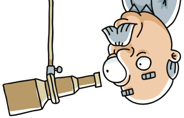
Handling Failures: Full GC

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

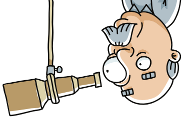
redhat.

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

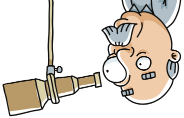
Handling Failures: Observations



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

redhat.

Handling Failures: Observations



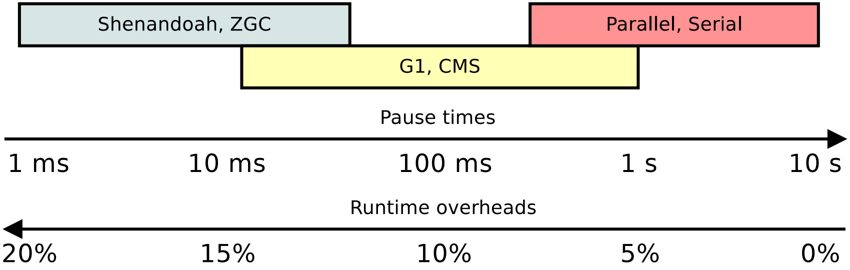
1. Being fully concurrent is nice, but own the failures
   - The failures will happen, accept it
   - «Our perfect GC melted down, because you forgot this magic VM option(, stupid)» flies only that far
2. Graceful and observable degradation is key
   - Getting worse incrementally is better than falling 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

redhat.

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!

redhat

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
```


redhat.