Shenandoah GC

Version 2.0 (2019): The Great Revolution

Aleksey Shipilëv

shade@redhat.com
@shipilev

Safe Harbor / Тихая Гавань

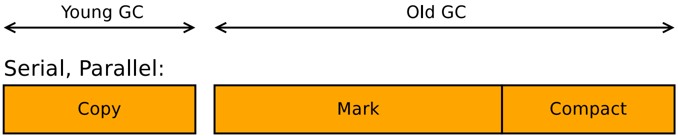
Anything on this or any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

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

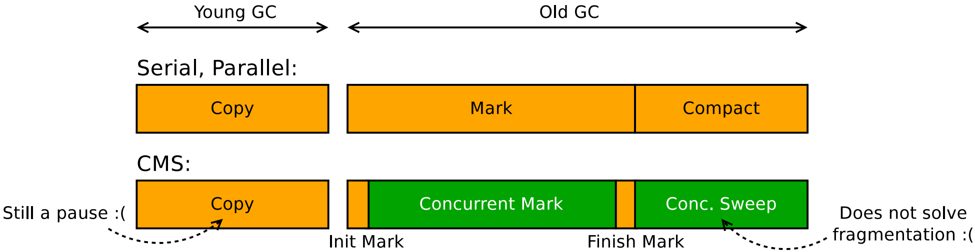
Basics: OpenJDK GCs Landscape

Basics: OpenJDK GCs Landscape

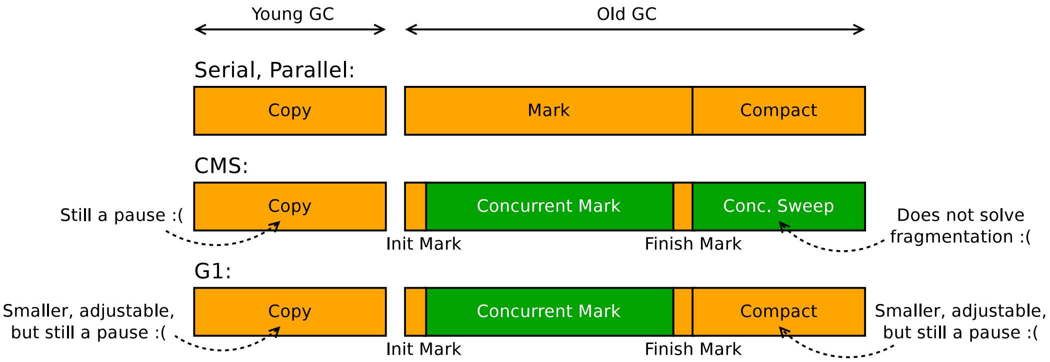


redhat

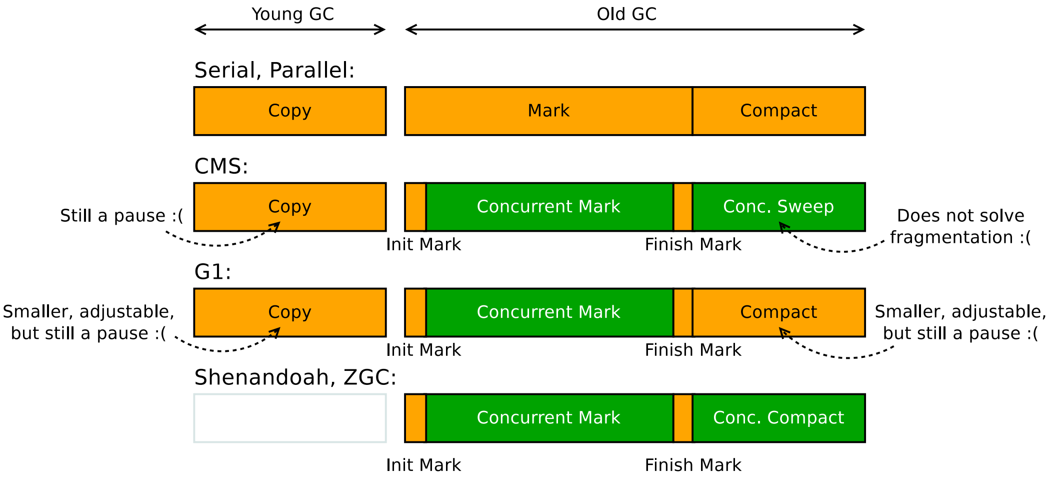
Basics: OpenJDK GCs Landscape



Basics: OpenJDK GCs Landscape



Basics: OpenJDK GCs Landscape



Basics: Concurrent GC Only For Large Heaps?

Basics: Concurrent GC Only For Large Heaps?

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

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

redhat

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, AArch64, 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 13, **11u, 8u**
- 8u backport ships in RHEL 7.4+, Fedora 24+
- 11u 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: Shenandoah 2.0

Major differences from older talks:

1. Load reference barriers
2. Elimination of separate fwdptr slot
3. Extended platform support

Basics: Shenandoah 2.0

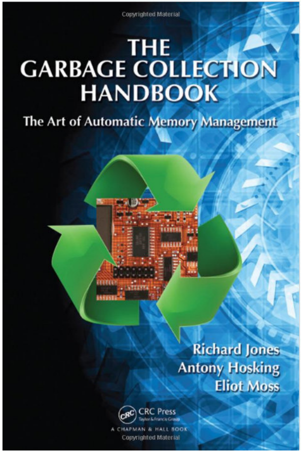
Major differences from older talks:

1. Load reference barriers
2. Elimination of separate fwdptr slot
3. Extended platform support

Status:

- In JDK 13 GA
- In JDK 11.0.5+ Red Hat dowstreams
- Backporting to JDK 8u is in progress (Shenandoah 1.0 is there already)

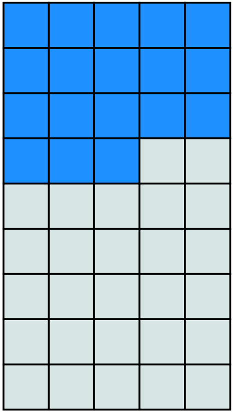
Basics: This Message Is Brought To You By



- IMHO, discussing the 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

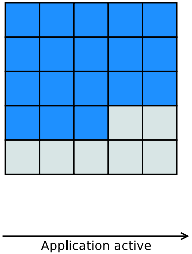
Overview: Heap Structure



Shenandoah is a *regionalized* GC:
heap split into equally-sized regions

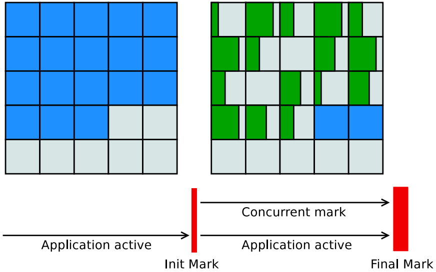
- Heap organization is similar to G1
- Collects most garbage regions first
- Not generational (yet), single heap
- Requires little auxiliary metadata

Overview: Usual GC Cycle



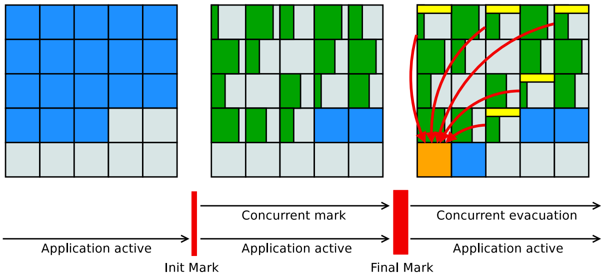
redhat.

Overview: Usual GC Cycle



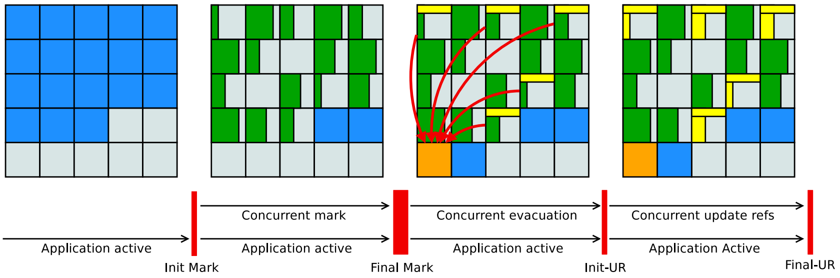
1. Concurrent marking

Overview: Usual GC Cycle



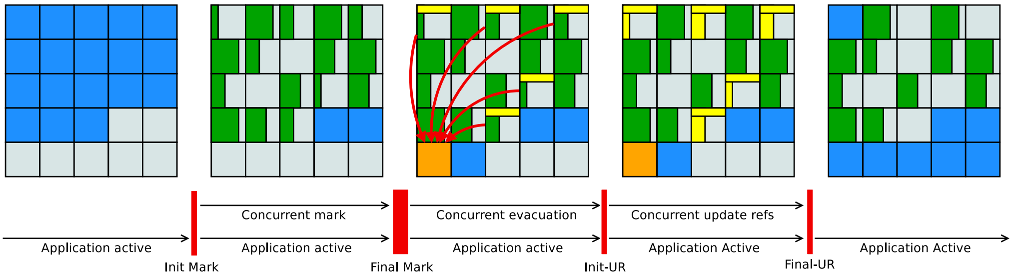
1. Concurrent marking
2. Concurrent evacuation

Overview: Usual GC Cycle



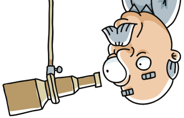
1. Concurrent marking
2. Concurrent evacuation
3. Concurrent update references (optional)

Overview: Usual GC Cycle



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

Overview: Usual GC Log

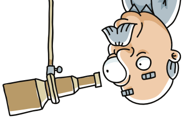


LRUFragger, 100 GB heap, $\approx$ 80 GB live data:

```
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
```


redhat.

Overview: Usual GC Log



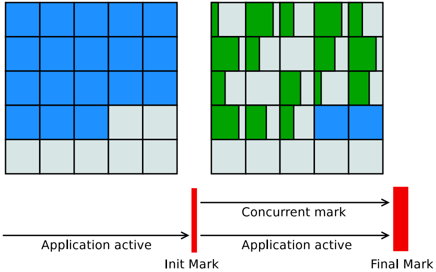
LRUFragger, 100 GB heap, $\approx$ 80 GB live data:

```
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
```


redhat

Phases

Mark: Usual GC Cycle



1. Concurrent marking

redhat.

Mark: Reachability

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

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 *(see: 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 *(see: Epsilon GC)*
2. **Reference counting**: track the number of incoming refs, treat `RC=0` 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 *(see: Epsilon GC)*
2. **Reference counting**: track the number of incoming refs, treat `RC=0` as garbage
3. **Tracing**: walk the object graph, find reachable objects, treat *everything else* as garbage

redhat.

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

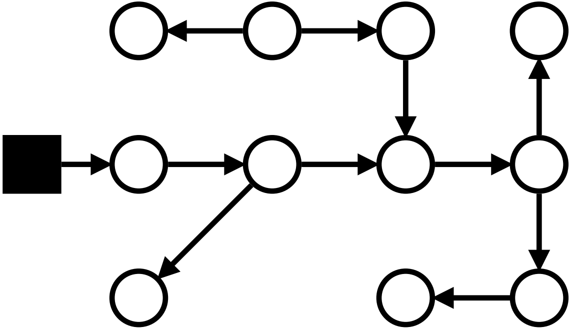
Mark: Three-Color Abstraction

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

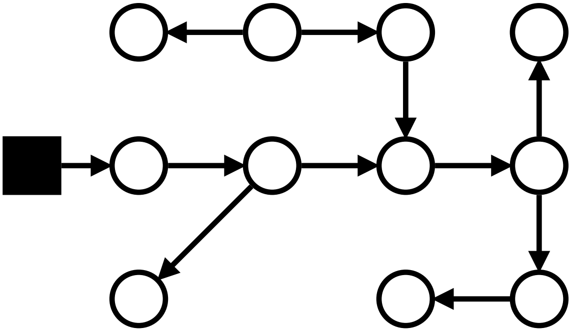
Daily Blues:
«All the marking algorithms do is
coloring white gray, and then coloring gray black»

Mark: Stop-The-World Mark



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

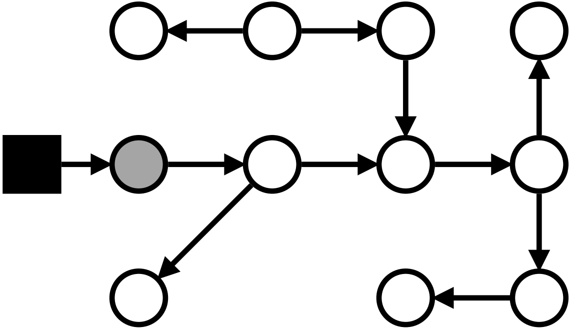
Mark: Stop-The-World Mark



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

redhat

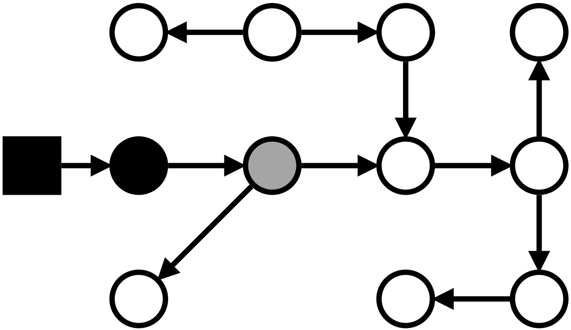
Mark: Stop-The-World Mark



References from Black are now Gray,
scanning Gray references

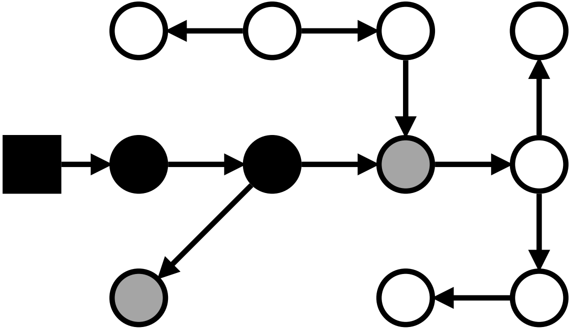
redhat.

Mark: Stop-The-World Mark



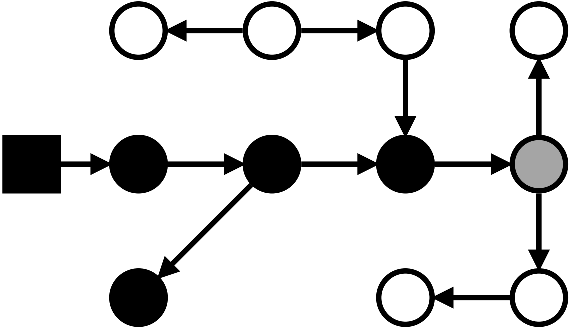
Finished scanning Gray, color them Black;
new references are Gray

Mark: Stop-The-World Mark



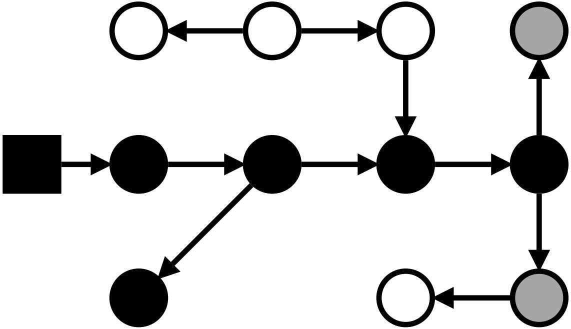
Gray → Black;
reachable from Gray → Gray

Mark: Stop-The-World Mark



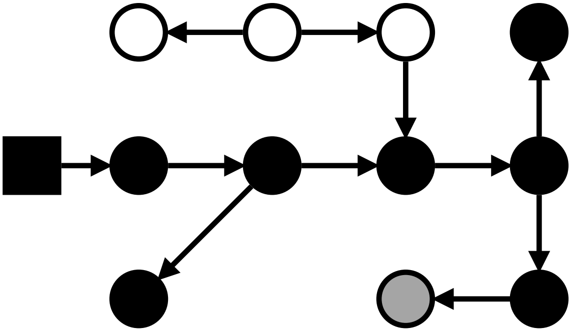
Gray → Black;
reachable from Gray → Gray

Mark: Stop-The-World Mark



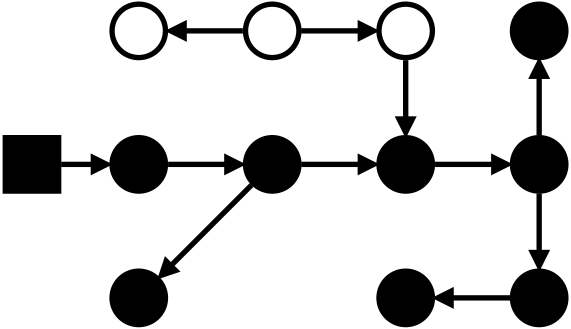
Gray $\rightarrow$ Black;
reachable from Gray $\rightarrow$ Gray

Mark: Stop-The-World Mark



Gray $\rightarrow$ Black;
reachable from Gray $\rightarrow$ Gray

Mark: Stop-The-World Mark



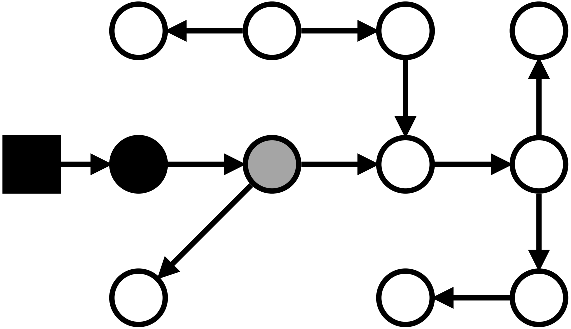
Finished: everything reachable is Black;
all garbage is White

Concurrent Mark: Mutator Problems

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

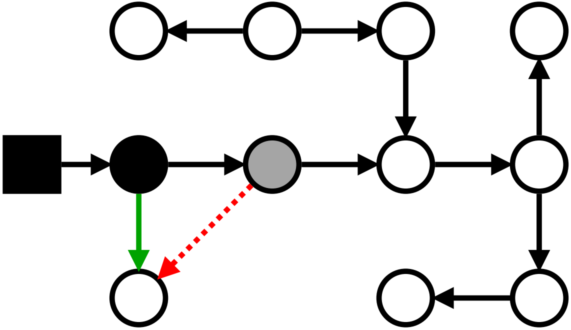
We contemptuously call it *mutator* because of that.

Concurrent Mark: Mutator Problems



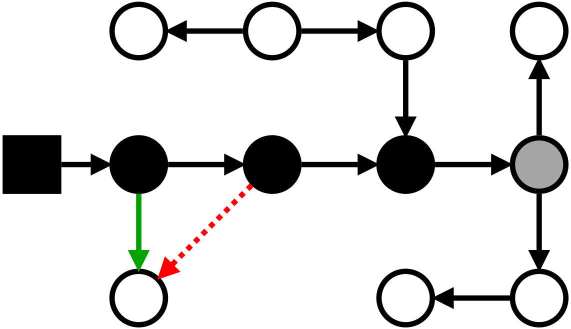
Wavefront is here,
and starts scanning the references in Gray object...

Concurrent Mark: Mutator Problems



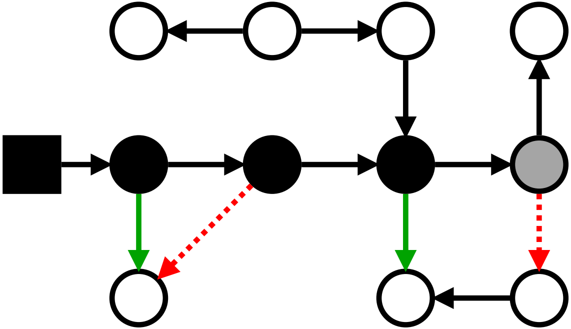
Mutator removes the reference from Gray...
and inserts it to Black!

Concurrent Mark: Mutator Problems



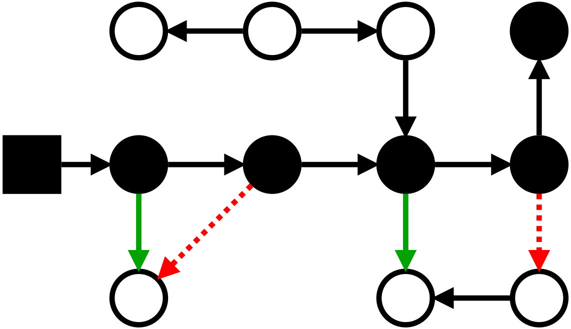
...or mutator inserted the reference to
transitively reachable White object into Black

Concurrent Mark: Mutator Problems



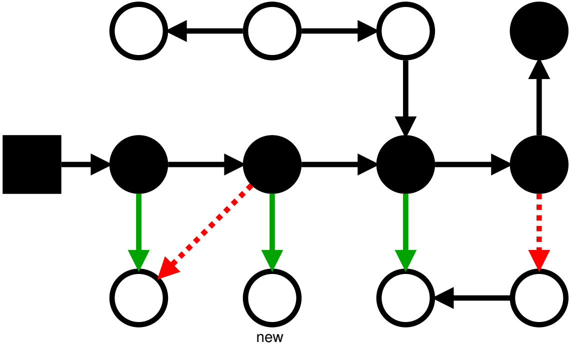
...or mutator inserted the reference to
transitively reachable White object into Black

Concurrent Mark: Mutator Problems



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

Concurrent Mark: Mutator Problems

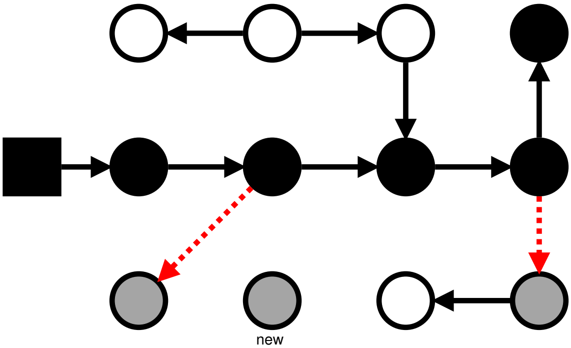


Another quirk: created new **new object**,
and inserted it into Black

Concurrent Mark: Textbook Says

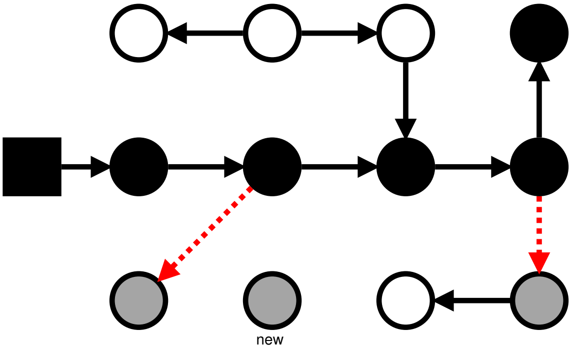
There are at least three approaches to solve this problem. All of them require intercepting heap accesses. Short on time, we shall discuss what G1 and Shenandoah are doing today.

Concurrent Mark: SATB



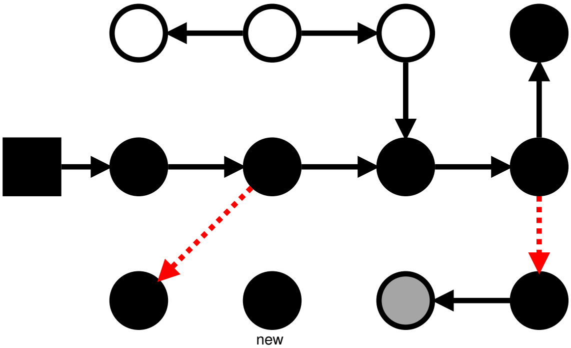
Color all **removed** referents **Gray**

Concurrent Mark: SATB



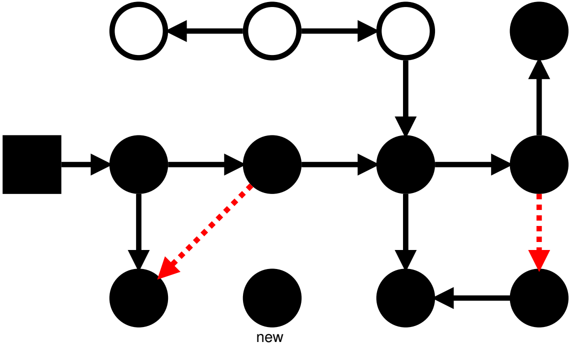
Color all new objects **Black**

Concurrent Mark: SATB



Finishing...

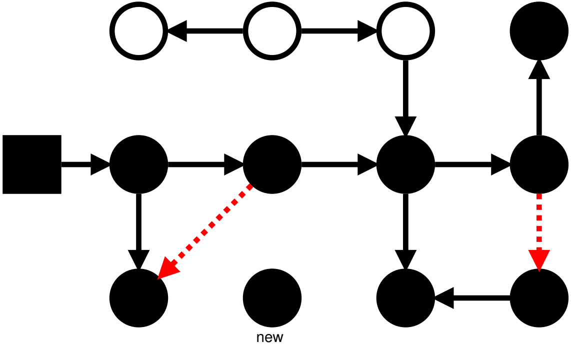
Concurrent Mark: SATB



Done!

redhat

Concurrent Mark: SATB



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

Concurrent Mark: SATB Writes (pseudocode)

Concurrent Mark: SATB Writes (pseudocode)

```
void WriteWithSATB(Obj* loc, Obj val):
  // SATB write pre-barrier:
  if (HEAP->is_marking()): // check thread-local flag
    Obj old = *loc;          // read old value
    if (old != NULL):
      // put the value in buffer
      THREAD->addToSATBBuffer(old);
      // maybe deliver full buffer to GC
      THREAD->maybeFlushSATBBuffer();

  // Barrier is done. Do the actual write:
  *loc = val;
```

Concurrent Mark: SATB Barrier (inline)

```
  # 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 fully concurrent, but that is not very practical today

redhat

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

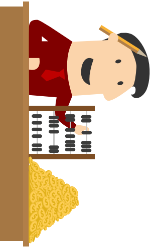
Final Mark: stop the mutator to avoid races

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

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

redhat.

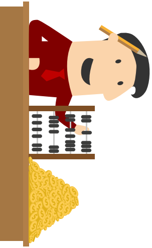
Concurrent Mark: Barriers Cost[2]



	Throughput hit, %
Cmp	
Cps	
Cry	
Der	
Mpg	
Smk	
Ser	
Xml	

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

Concurrent Mark: Barriers Cost[2]

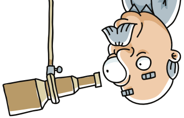


	Throughput hit, %
	SATB
Cmp	-2.1
Cps	ϵ
Cry	ϵ
Der	-1.7
Mpg	ϵ
Smk	-0.8
Ser	-1.6
Xml	-2.4

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

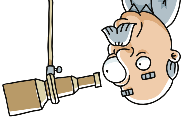
redhat

Concurrent Mark: Observations



1. Extended concurrency needs to pay with more barriers
   - Ideal STW GC beats ideal concurrent GC on pure throughput
   - Unless there are spare CPUs to offload the concurrent GC

Concurrent Mark: Observations

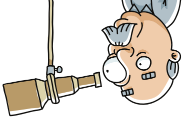


1. Extended concurrency needs to pay with more barriers
   - Ideal STW GC beats ideal concurrent GC on pure throughput
   - Unless there are spare CPUs to offload the concurrent GC

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

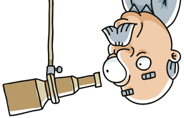
redhat

Concurrent Mark: Tips



1. High load, don’t care about pauses? Prefer STW GC!
   - No need to pay for concurrency when you cannot exploit it
   - Empty GC log does not mean no GC overhead

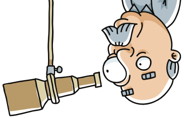
Concurrent Mark: Tips



1. High load, don't care about pauses? Prefer STW GC!
   - No need to pay for concurrency when you cannot exploit it
   - Empty GC log does not mean no GC overhead
2. #objects and #references define conc mark performance
   - Flatter object graphs are quicker to walk
   - Primitive fields/arrays are no-brainer for GC
   - Generally, lots of references is tolerable when parallelisable

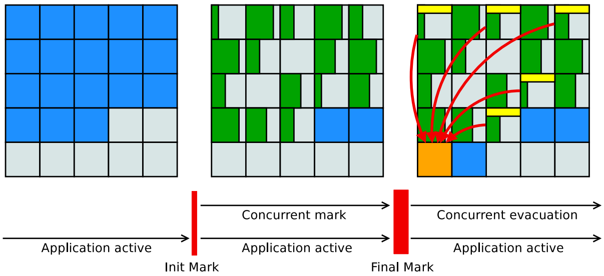
redhat

Concurrent Mark: Tips



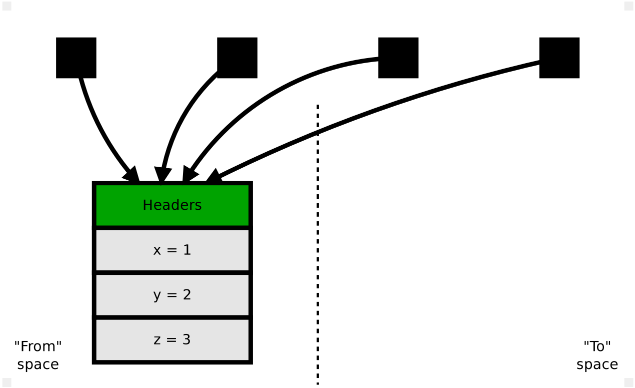
1. High load, don't care about pauses? Prefer STW GC!
   - No need to pay for concurrency when you cannot exploit it
   - Empty GC log does not mean no GC overhead
2. #objects and #references define conc mark performance
   - Flatter object graphs are quicker to walk
   - Primitive fields/arrays are no-brainer for GC
   - Generally, lots of references is tolerable when parallelisable
3. Long chains of references hurt tracing GCs
   - Long linked lists are GC nemesis: unparallelisable
   - Arrays (and derivatives, e.g. hash tables) are perfect
   - Trees seem to be the sane middle ground

Evac: Usual GC Cycle



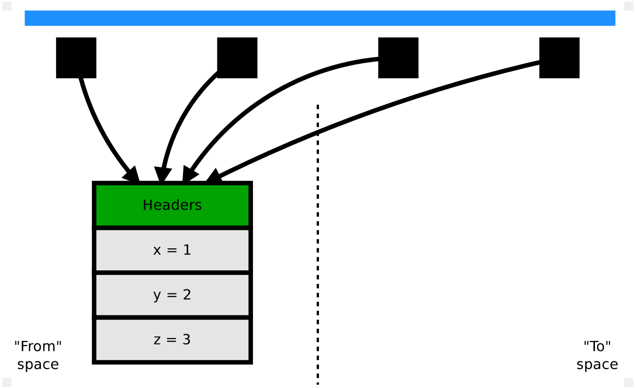
1. Concurrent marking
2. Concurrent evacuation

Evac: Stop-The-World



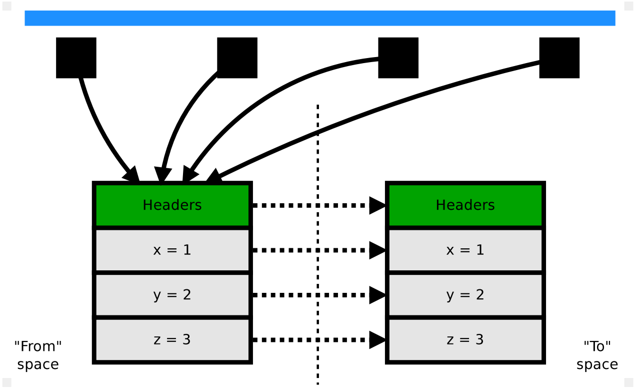
Problem:
there is the object, the object is referenced from somewhere, need to move it to new location

Evac: Stop-The-World



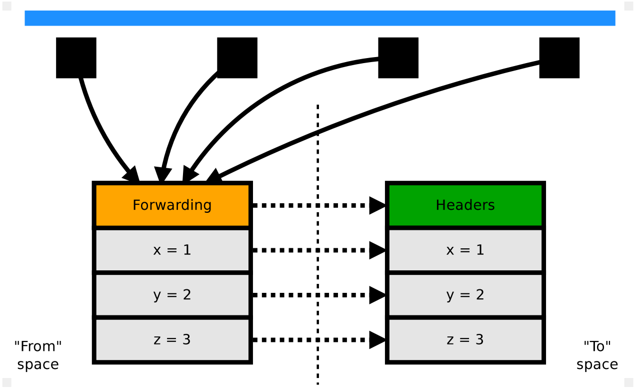
Step 1: Stop The World, evasive maneuver to distract mutator from looking into our mess

Evac: Stop-The-World



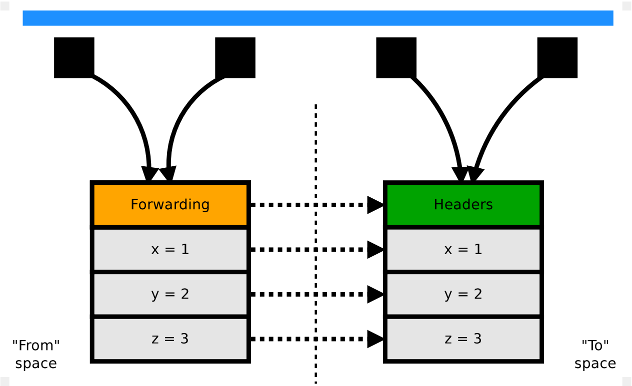
Step 2:
Copy the object with all its contents

Evac: Stop-The-World



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

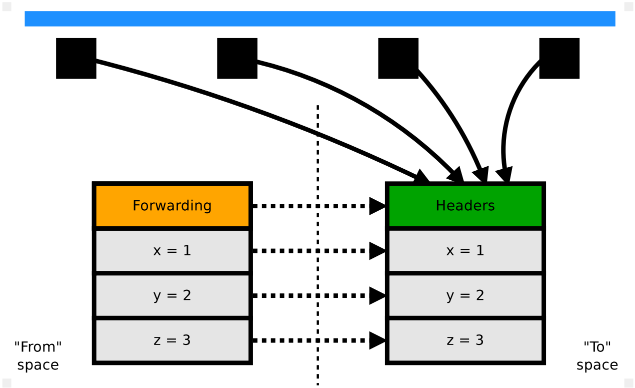
Evac: Stop-The-World



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

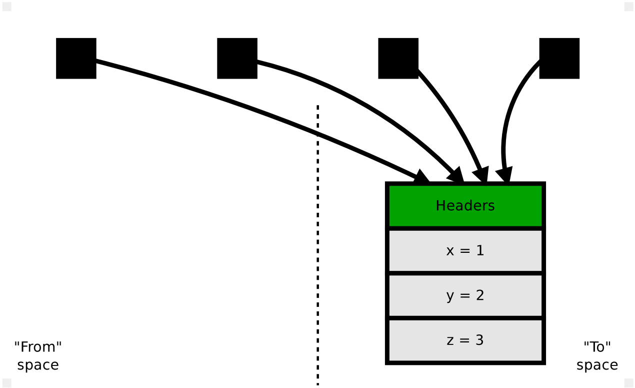
redhat.

Evac: Stop-The-World



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

Evac: Stop-The-World



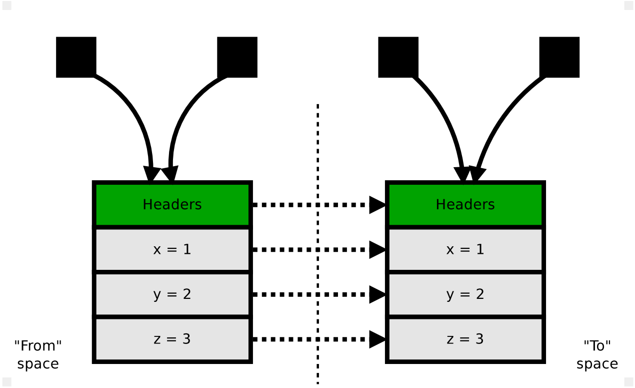
Everything is fine in the world, set the mutators free! Done!

Concurrent Evac: Mutator Problems

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

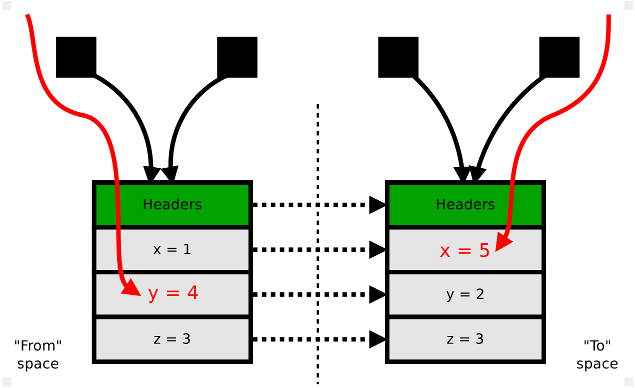
With **concurrent** copying everything gets is significantly harder: the application writes into the objects while we are moving the same objects!

Concurrent Evac: Mutator Problems



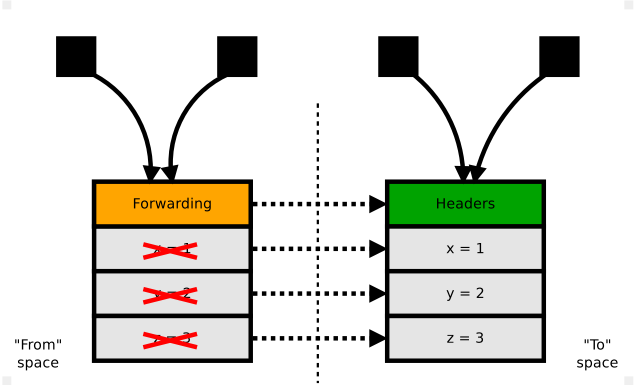
While object is being moved, there are *two* copies of the object, and both are reachable!

Concurrent Evac: Mutator Problems



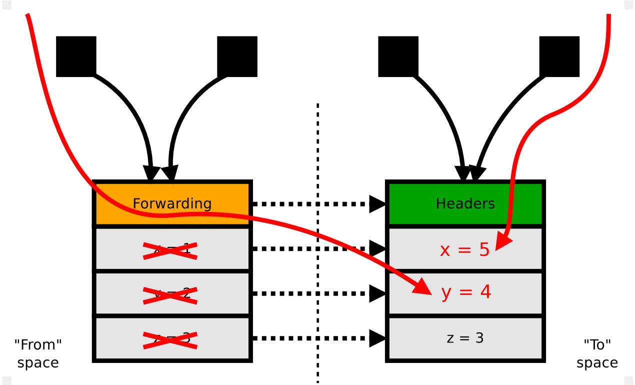
Thread A writes $y = 4$ to one copy, and Thread B writes $x = 5$ to another. Which copy is correct now, huh?

Concurrent Evac: Load Reference Barriers



Idea:
If we need to copy objects, do it before any use. Let application act when loading the object (when *loading reference to it*)

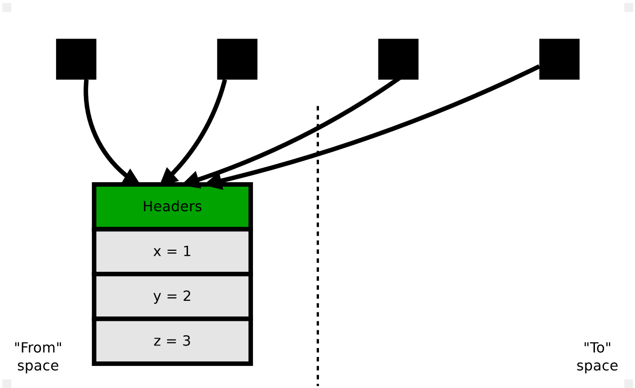
Concurrent Evac: Load Reference Barriers



If copy exists:
Resolve it and use it

redhat.

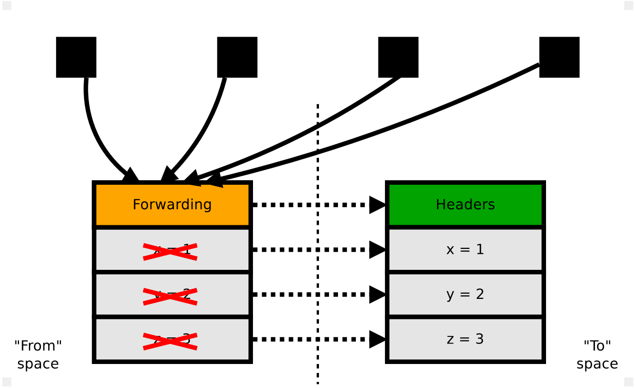
Concurrent Evac: Load Reference Barriers



If copy does not exist: Can read/write from the current copy? Problematic...

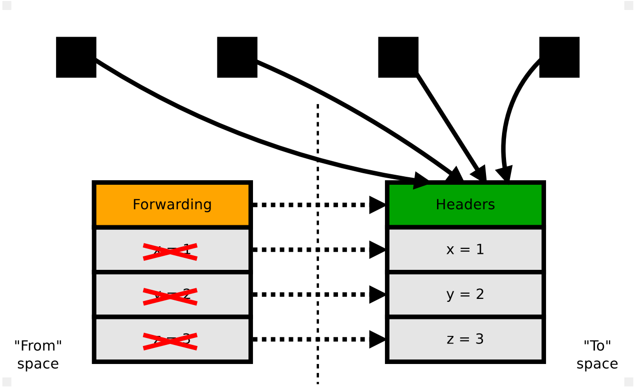
redhat

Concurrent Evac: Load Reference Barriers



Copy ourselves: make sure every access is done to actual copy. Agree on which copy is actual by changing the forwarding *atomically*

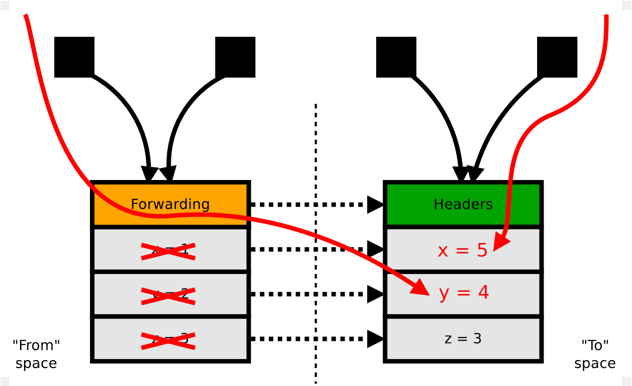
Concurrent Evac: Load Reference Barriers



After GC did its work updating the references, we can recycle the old objects, along with forwarding pointers

redhat.

Load Ref Barrier: Motivation



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

Load Ref Barrier: Pseudocode

```
Obj ReadWithLRB(Obj* loc):
  // Read the reference
  Obj obj = *loc

  // Load reference barrier:
  if (HEAP->has_forwarded_objects()):   // single byte
    if (HEAP->in_collection_set(obj)):  // dense bytemap
      if (obj->is_forwarded()):         // object header
        obj = obj->forwardee()
      else:
        obj = LRB_Slowpath(loc, obj)

  // Barrier is done. Here's our actual object:
  return obj
```

Load Ref Barrier: Fastpath

```
  mov %r10, ...              # Load reference
  testb $0x1, 0x20(%r15) # Has forwarded objects?
  jne LRB-MIDPATH
BACK:
  # normal access happens afterwards...

  # somewhere later
LRB-MIDPATH:
    mov %r11, %r10
    shr %r11, 16             # Compute region ID
    testb $0, (CSBM, %r11) # Test cset bytemap
    je BACK
    # decode and test fwdptr
    # ...and maybe jump to slowpath
```

Load Ref Barrier: Slowpath

```
Obj LRB_Slowpath(Obj* loc, Obj obj):
  assert(HEAP->has_forwarded_objects(), "fastpath")
  assert(HEAP->in_collection_set(obj), "fastpath")

  Obj copy = copy(obj)

  // Try to install new copy as actual:
  if (obj->cas_forwardee(NULL, copy)):
    // success, this is our new copy
    return copy
  else:
    // someone else did it: pick up the result
    return obj->forwardee()
```

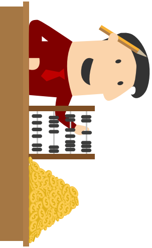
Load Ref Barrier: aside, GC Evacuation Code

```
void GC_Evacuate():
  for (Region r : HEAP->collection_set()):
    for (Obj obj : r->live_objects()):
      if (!obj->is_forwarded()):
        // Not copied yet, do it now:
        Obj copy = copy(obj);

        // Try to install new copy. Don't care if failed.
        obj->cas_forwardee(NULL, copy);
```

Roll over collection set and copy all live objects out.
Skip objects that LRB evacuated itself.

redhat.

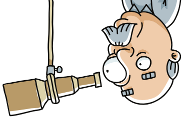
Load Ref Barrier: Barriers Cost[2]



	Throughput hit, %	
	SATB	LRB
Cmp	-2.1	-11.3
Cps	ϵ	-9.2
Cry	ϵ	ϵ
Der	-1.7	-5.9
Mpg	ϵ	-11.7
Smk	-0.8	-1.8
Ser	-1.6	-6.0
Xml	-2.4	-11.6

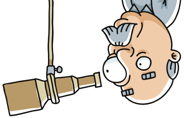
[2]Performance compared to STW Shenandoah with all barriers disabled

Load Ref Barrier: Observations



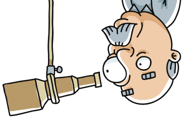
1. Shenandoah needs LRB on every reference load
   - The frequency is optimized: optimizers try to avoid heap read
   - Pretty much everywhere you expect compressed oops decoding

Load Ref Barrier: Observations



1. Shenandoah needs LRB on every reference load
   - The frequency is optimized: optimizers try to avoid heap read
   - Pretty much everywhere you expect compressed oops decoding
2. Passive LRB cost is low
   - Single thread-local load and predicted branch
   - Still has non-zero costs: instructions, optimizations interference

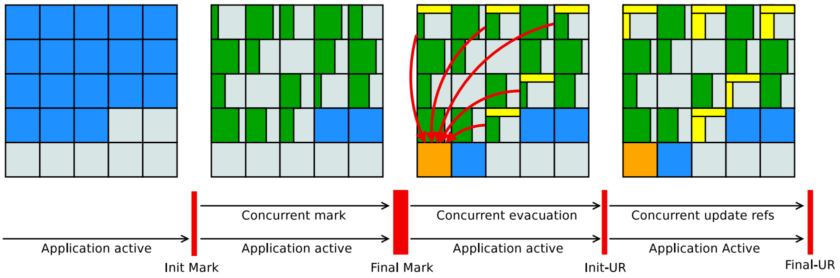
Load Ref Barrier: Observations



1. Shenandoah needs LRB on every reference load
   - The frequency is optimized: optimizers try to avoid heap read
   - Pretty much everywhere you expect compressed oops decoding
2. Passive LRB cost is low
   - Single thread-local load and predicted branch
   - Still has non-zero costs: instructions, optimizations interference
3. Active LRB cost is moderate
   - Most exits from LRB midpath: not in cset or already forwarded
   - GC does the bulk of the evacuation work

redhat.

Update References: Usual GC Cycle



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

Update References: GC Code (pseudocode)

```
void GC_UpdateRefs():
  for (Region r : HEAP->regions_snapshot()):
    for (Obj obj : r->live_objects()):
      for (Obj* loc : obj->fields()):
        Obj f = *loc;
        if (f->is_forwarded()):
          CAS_raw(loc, f, f->forwardee())
```

Roll over all live objects,
and update all forwarded references.

Update References: GC Code (pseudocode)

```
void GC_UpdateRefs():
  for (Region r : HEAP->regions_snapshot()):
    for (Obj obj : r->live_objects()):
      for (Obj* loc : obj->fields()):
        Obj f = *loc;
        if (f->is_forwarded()):
          CAS_raw(loc, f, f->forwardee())
```

CAS fails?

$\Rightarrow$ some other store happened

$\Rightarrow$ stored reference already passed LRB

$\Rightarrow$ guaranteed to be the reference to actual copy

Update References: Nasty Corner Case

There are special operations that bypass normal LRB:

Update References: Nasty Corner Case

There are special operations that bypass normal LRB:
arraycopy/clone

- Usually copy raw memory underneath
  - Because, performance! Vectorized copy FTW
  - Fine for primitive data, catastrophic for Java references
- Can therefore overwrite already updated refs
  - Update-refs progress guarantees out of the window
  - Unless, we fix up source/destination before/after the copy!

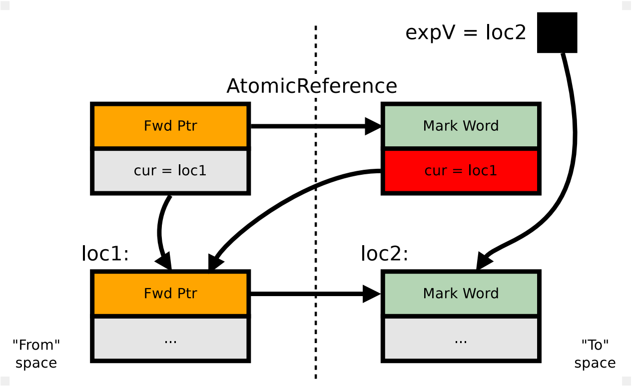
Exotic Barriers: Arraycopy (pseudocode)

```
void ArraycopyBarrier(Obj* src, Obj* dst,
                      int beg, int len):
  // Fixup before copy:
  for (int c : [0, len]):
    Obj* loc = (src + beg + c)
    Obj elem = *loc
    if (elem->is_forwarded()):
      CAS_raw(loc, elem, elem->forwardee())

  // Do the actual copy of known good stuff
  arraycopy(src, dst, deg, len);
```

Or: «Oh, no LRB for Bender?
Fine, I am going to build my own barrier...»

Exotic Barriers: CAS

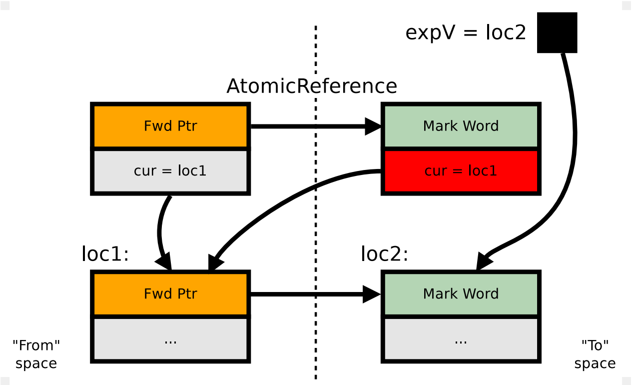
```
atomicRef.compareAndSwap(expV, newV)
```



LRB makes sure we never get exposed to old copies...

Exotic Barriers: CAS

`atomicRef.compareAndSwap(expV, newV)`



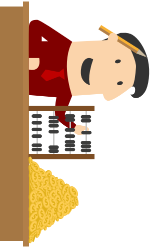
LRB makes sure we never get exposed to old copies...

Which breaks when we do CAS that compares new copy with old copy for the same object!

Exotic Barriers: CAS Barrier (pseudocode)

```
bool ConcGC_CAS(Obj* loc, Obj expV, Obj newV):
  // Optimistic attempt:
  if (CAS_raw(loc, expV, newV)):
    return true;

  // False negative? Fix up:
  Obj old = *loc;
  if (old->is_forwarded()):
    CAS_raw(loc, old, old->forwardee())

  // Try again:
  return CAS_raw(loc, expV, newV);
```

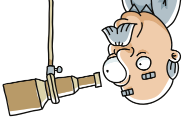
Exotic Barriers: Barriers Cost[2]



	Throughput hit, %		
	SATB	LRB	CAS, AC
Cmp	-2.1	-11.3	ϵ
Cps	ϵ	-9.2	ϵ
Cry	ϵ	ϵ	ϵ
Der	-1.7	-5.9	ϵ
Mpg	ϵ	-11.7	ϵ
Smk	-0.8	-1.8	ϵ
Ser	-1.6	-6.0	ϵ
Xml	-2.4	-11.6	ϵ

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

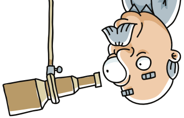
Exotic Barriers: Observations



1. CAS barriers are important for performance
   - Reference CASes are relatively rare
   - Most of the time CASes succeed, so fixup is not needed
   - When CAS fails, you have larger problem: retries, fallbacks

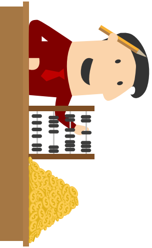
redhat.

Exotic Barriers: Observations



1. CAS barriers are important for performance
   - Reference CASes are relatively rare
   - Most of the time CASes succeed, so fixup is not needed
   - When CAS fails, you have larger problem: retries, fallbacks
2. Arraycopy barriers are sometimes critical
   - Before or after the actual raw copy blazes through...
   - GC would need to pre/post-handle the reference arrays

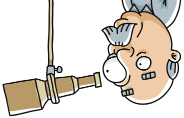
Overall: Barriers Cost[2]



	Throughput hit, %			
	SATB	LRB	CAS, AC	TOTAL
Cmp	-2.1	-11.3	ϵ	-12.9
Cps	ϵ	-9.2	ϵ	-9.2
Cry	ϵ	ϵ	ϵ	ϵ
Der	-1.7	-5.9	ϵ	-6.6
Mpg	ϵ	-11.7	ϵ	-12.7
Smk	-0.8	-1.8	ϵ	-2.5
Ser	-1.6	-6.0	ϵ	-7.5
Xml	-2.4	-11.6	ϵ	-13.5

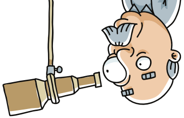
[2] Performance compared to STW Shenandoah with all barriers disabled

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
   - Works with 32-bit arches: x86_32 is done, ARM32 prototyping

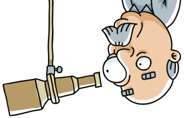
Overall: Observations



1. Easily portable across HW architectures
   - Special needs: `CAS` (performance is important, but not critical)
   - x86_64 and AArch64 are major implemented targets
   - Works with 32-bit arches: x86_32 is done, ARM32 prototyping
2. Trivially portable across OSes
   - Special needs: none
   - Linux is a major target, Windows is minor target
   - Vendors build and ship Mac OS and Solaris without problems

redhat.

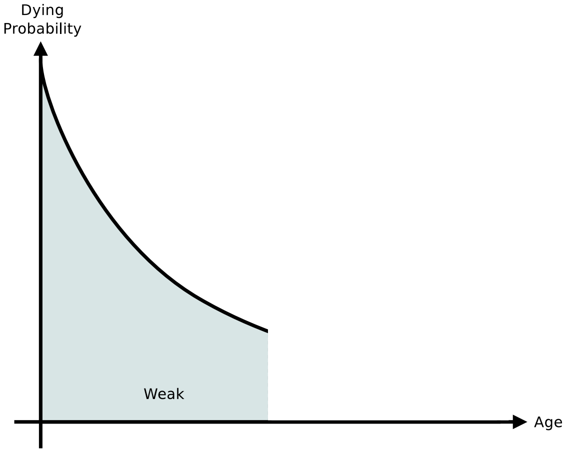
Overall: Observations



1. Easily portable across HW architectures
   - Special needs: `CAS` (performance is important, but not critical)
   - x86_64 and AArch64 are major implemented targets
   - Works with 32-bit arches: x86_32 is done, ARM32 prototyping
2. Trivially portable across OSes
   - Special needs: none
   - Linux is a major target, Windows is minor target
   - Vendors build and ship Mac OS and Solaris without problems
3. VM interactions are simple enough
   - Play well with compressed oops: pointers untouched
   - OS/CPU-specific things only for barriers codegen

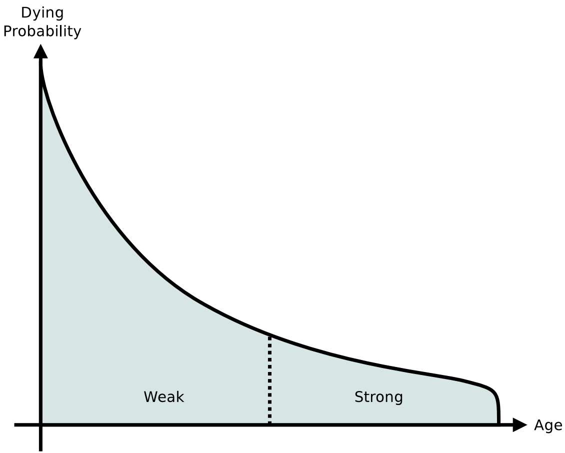
Example

Example: Generational Hypotheses



Weak hypothesis:
most objects die young

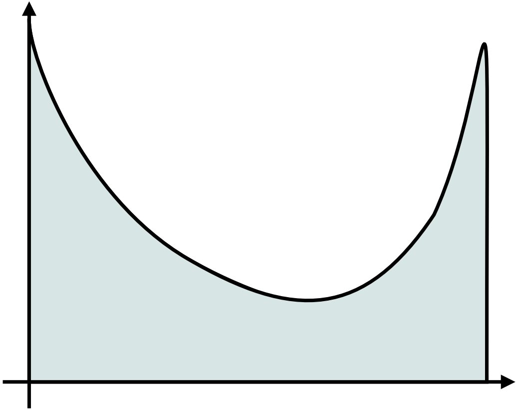
Example: Generational Hypotheses



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

redhat.

Example: Generational Hypotheses



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

In-memory LRU-like caches are the prime counterexamples

redhat.

Example: 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

Example: The Simplest LRU

The simplest LRU implementation in Java?

Example: 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;
  }
};
```

Example: Testing

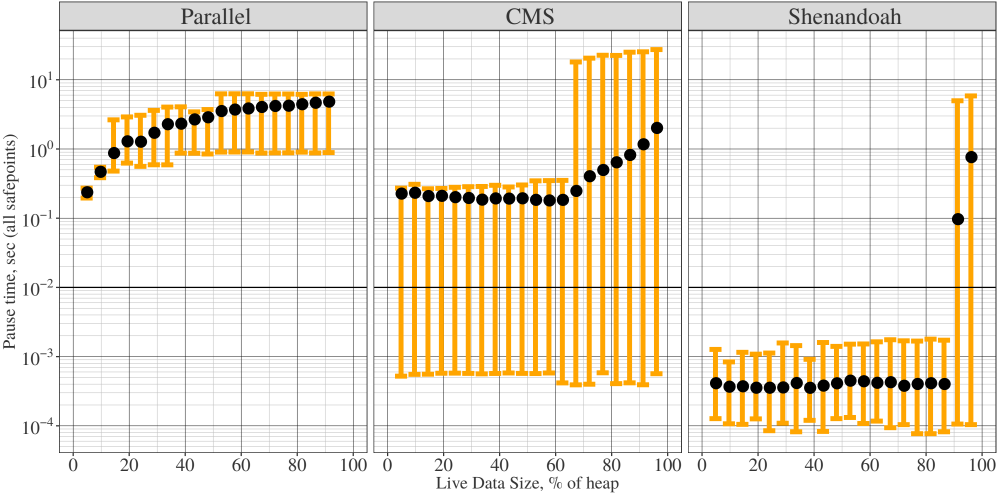
Boring config:

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`

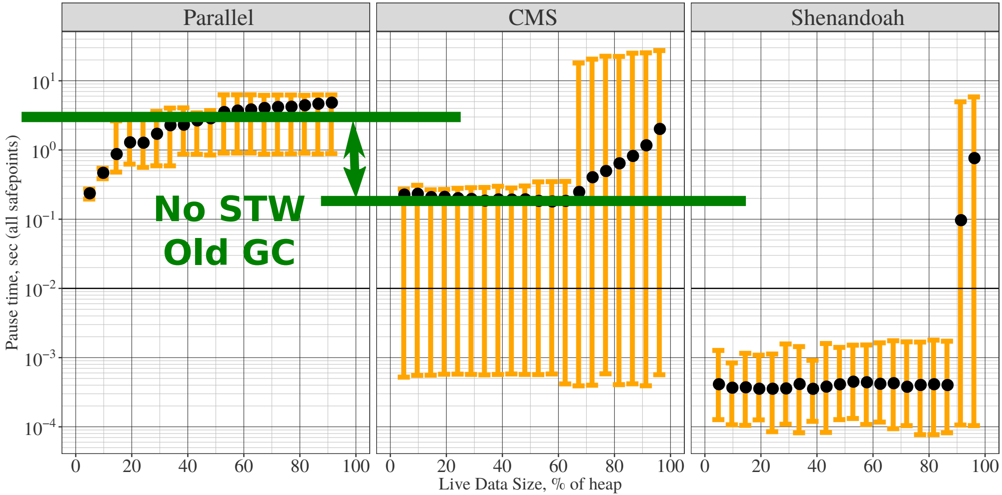
Varying cache size $\Rightarrow$ varying LDS $\Rightarrow$ make GC uncomfortable

redhat.

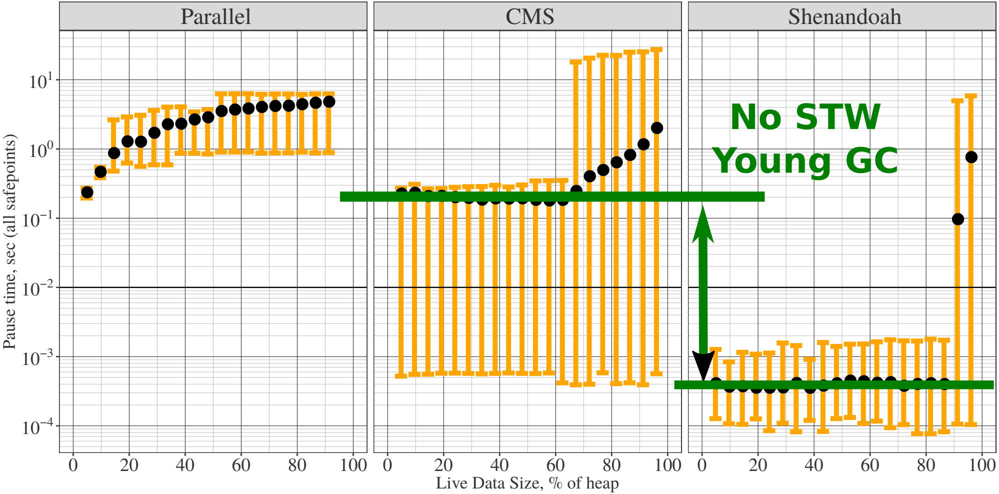
Example: Pauses vs. LDS



Example: Pauses vs. LDS

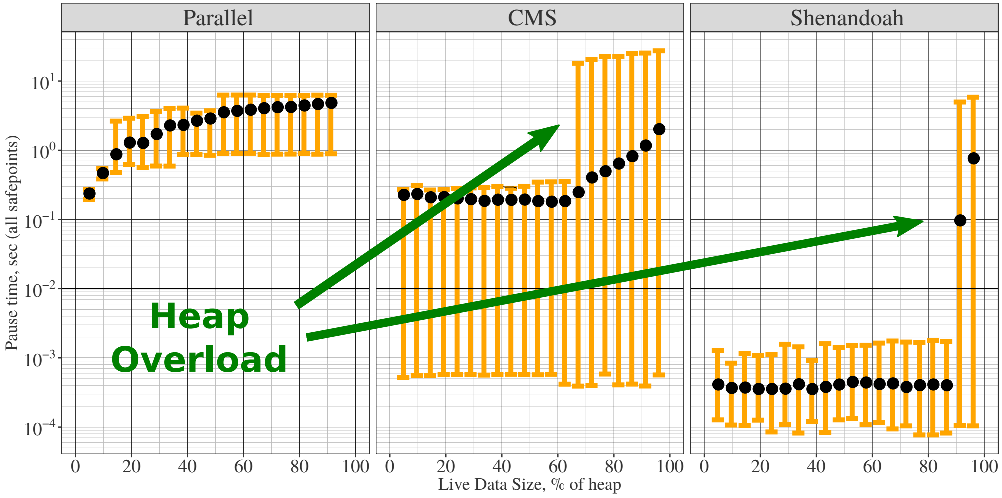


Example: Pauses vs. LDS

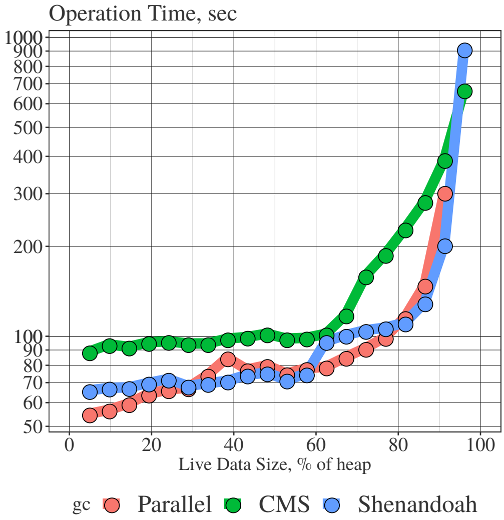


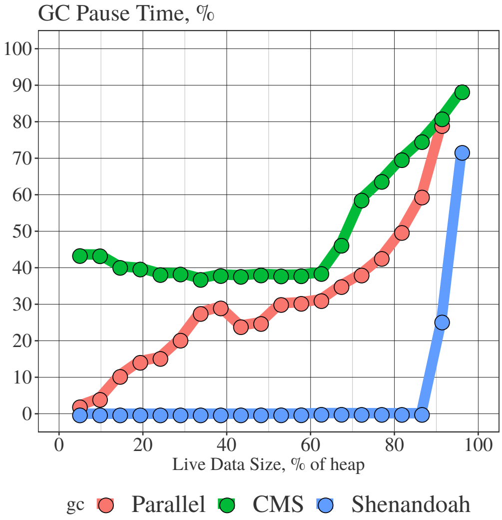
redhat.

Example: Pauses vs. LDS

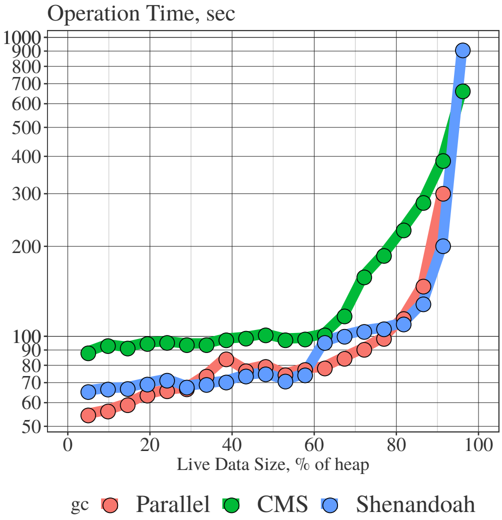


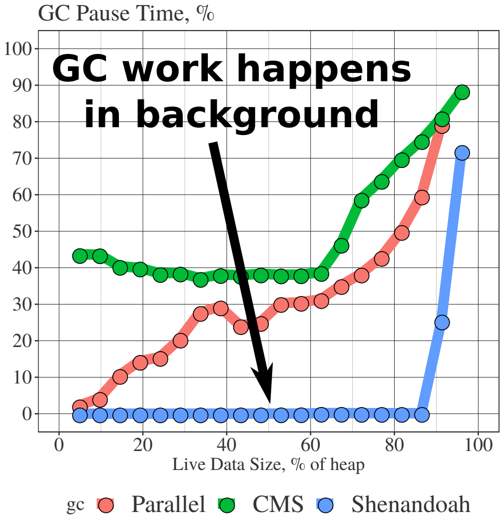
Example: Perf vs. LDS



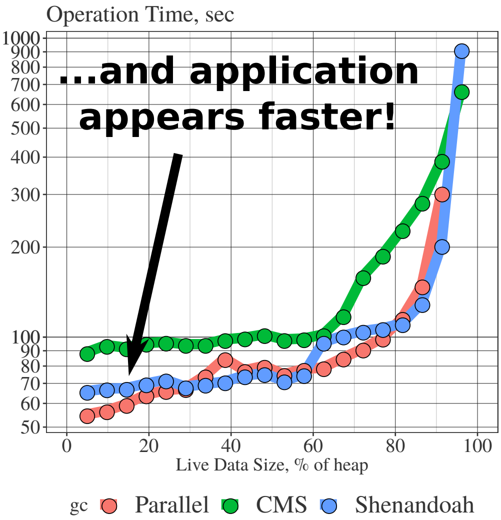


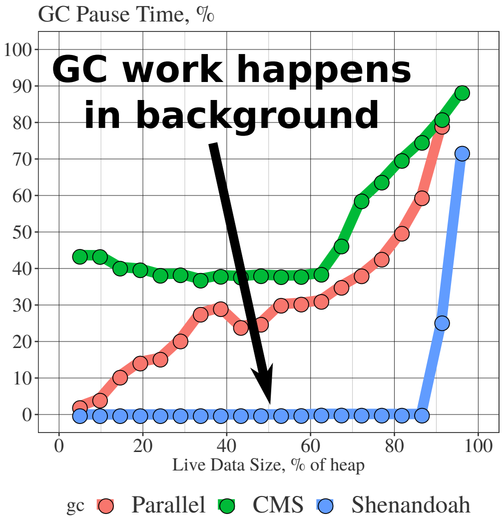
Example: Perf vs. LDS





Example: Perf vs. LDS





redhat.

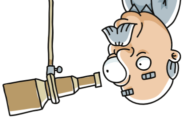
Command and Control

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

Command and Control: Off To The Races

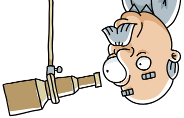


```
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is
above the time for allocation rate (3254.90 MB/s) to
deplete free headroom (13071M)
```

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

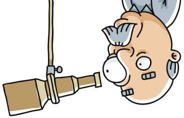


```
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is
above the time for allocation rate (3254.90 MB/s) to
deplete free headroom (13071M)
```

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

Command and Control: Off To The Races

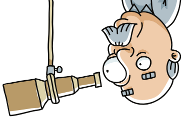


```
[1003.2s][gc] Trigger: Average GC time (4018.8 ms) is
above the time for allocation rate (3254.90 MB/s) to
deplete free headroom (13071M)
```

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

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

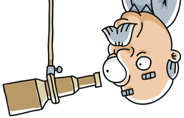
Problem:

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

Things that help:

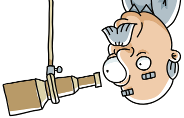
- Immediate garbage shortcuts: free memory early
- Aggressive heap expansion: prefer taking more memory
- Mutator pacing: stall allocators before they hit the wall
- Handling failures: gracefully degrade

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

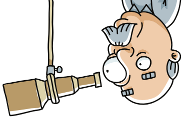


```
GC(7) Pause Init Mark 0.614ms
GC(7) Concurrent marking 76812M->76864M(102400M) 1.650ms
GC(7)   Total Garbage: 76798M
GC(7)   Immediate Garbage: 75072M, 2346 regions (97% of total)
GC(7) Pause Final Mark 0.758ms
GC(7) Concurrent cleanup 76864M->1844M(102400M) 3.346ms
```

1. Mark is fast, because most things are dead

redhat

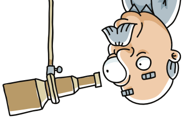
Immediates: Obvious Shortcut



```
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

Footprint: Shenandoah Overheads

Current Shenandoah **does not** require a lot of additional memory![3]

- Java heap: **no overheads at idle**
  «+»: and, compressed references are still working
  «−»: requires space for evacs when GC cycle is running
- Native structures: marking bitmap, 1/64 of heap
  «−»: `-Xmx` is still not close to RSS
  «+»: overhead is bounded: `-Xmx100g` means $\approx$102 GB RSS max

[3] Older one required a separate per-object fwdptr, yielding 1.05..1.5x overhead.

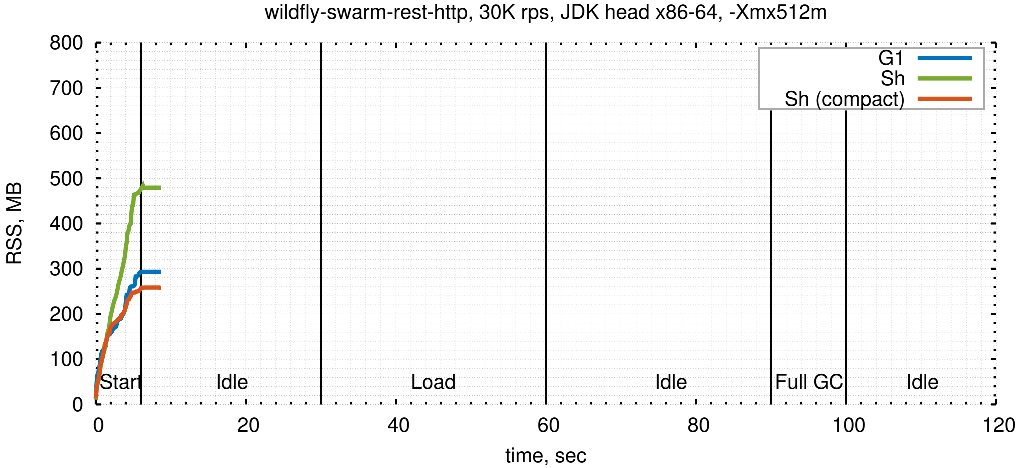
Footprint: Shenandoah Overheads

Current Shenandoah **does not** require a lot of additional memory![3]

- Java heap: **no overheads at idle**
  «+»: and, compressed references are still working
  «−»: requires space for evacs when GC cycle is running
- Native structures: marking bitmap, 1/64 of heap
  «−»: `-Xmx` is still not close to RSS
  «+»: overhead is bounded: `-Xmx100g` means ≈102 GB RSS max
- Surprise: a significant part of footprint story is heap sizing, not per-object or per-heap overheads

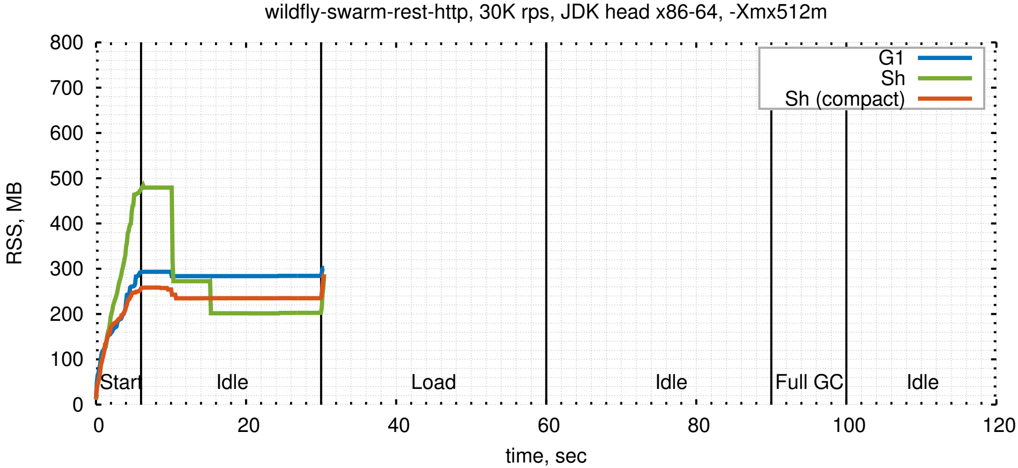
[3] Older one required a separate per-object fwdptr, yielding 1.05..1.5x overhead.

Footprint: Heap Sizing

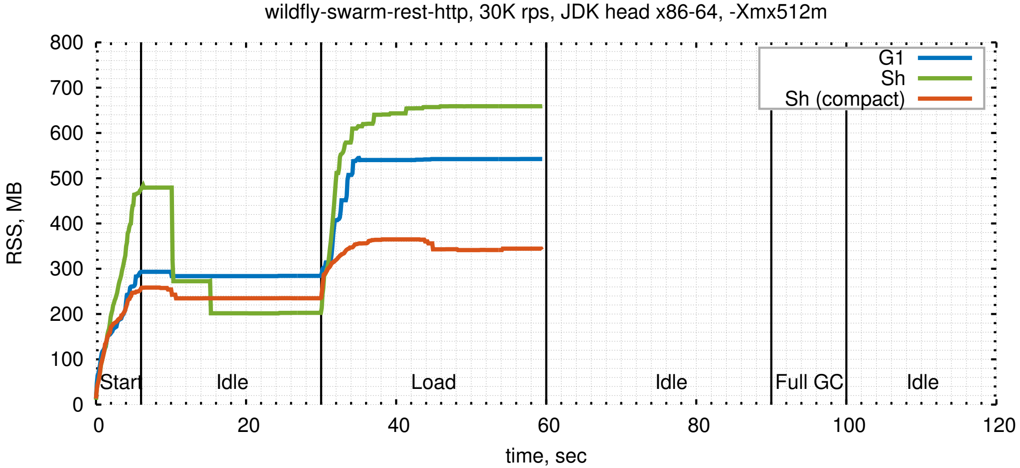


redhat.

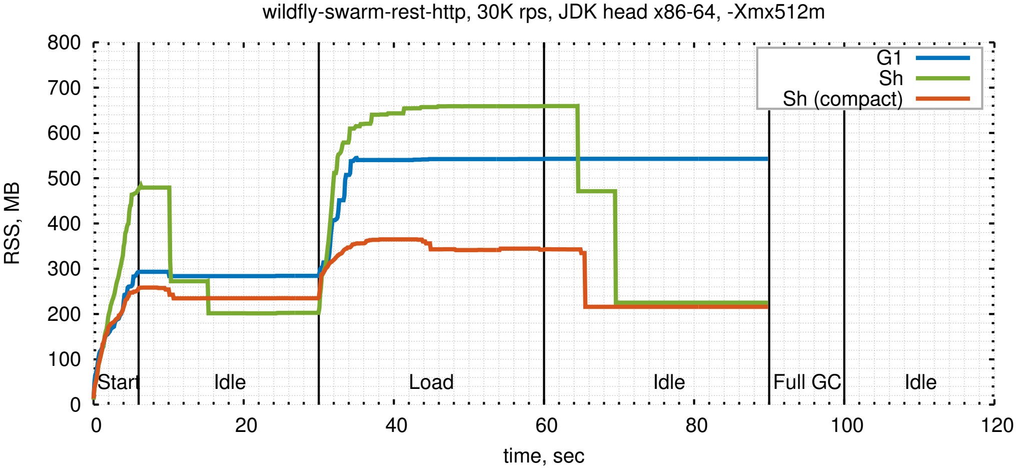
Footprint: Heap Sizing



Footprint: Heap Sizing

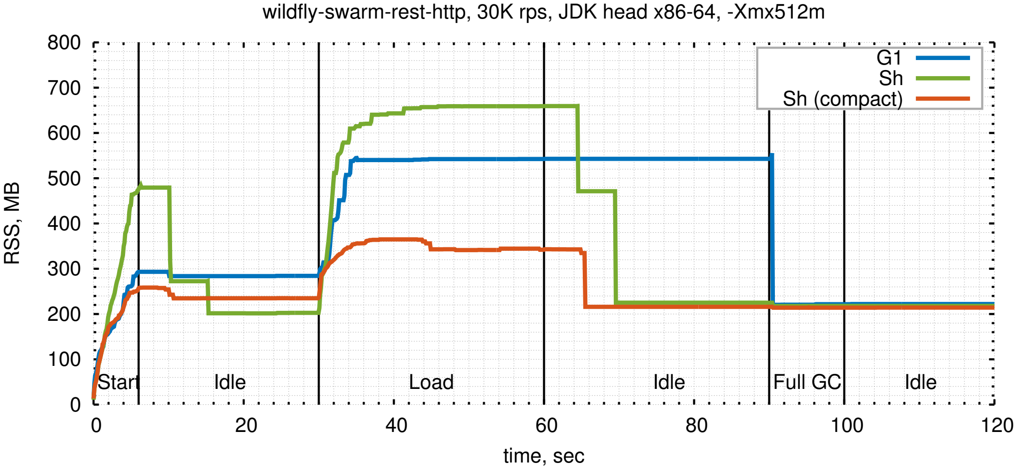


Footprint: Heap Sizing

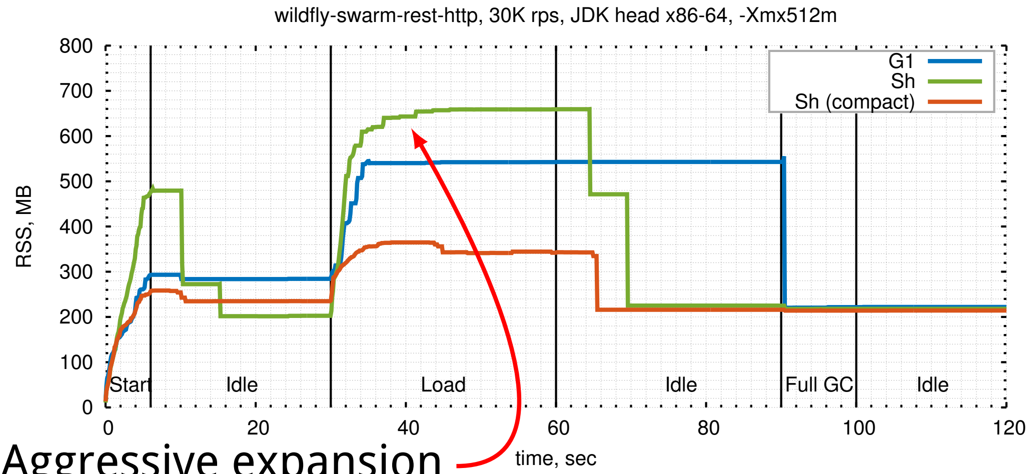


redhat.

Footprint: Heap Sizing

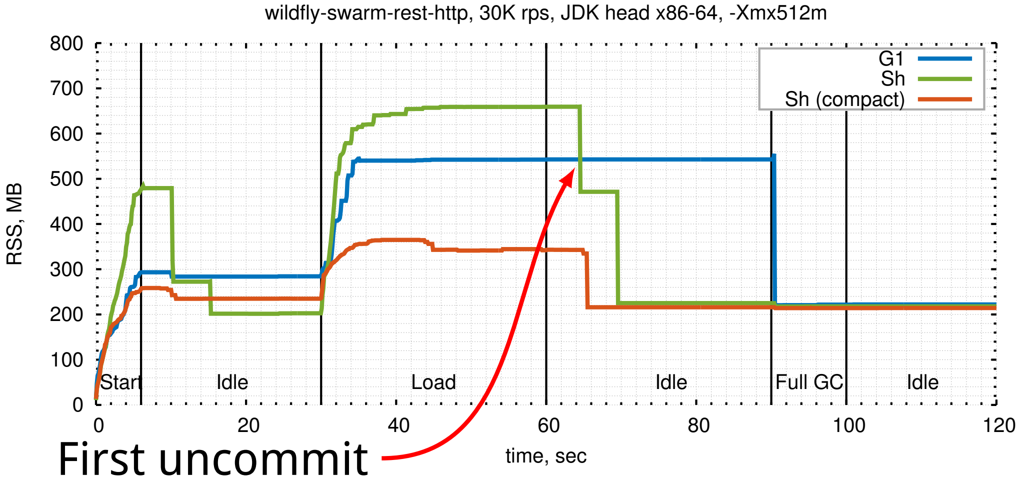


Footprint: Heap Sizing



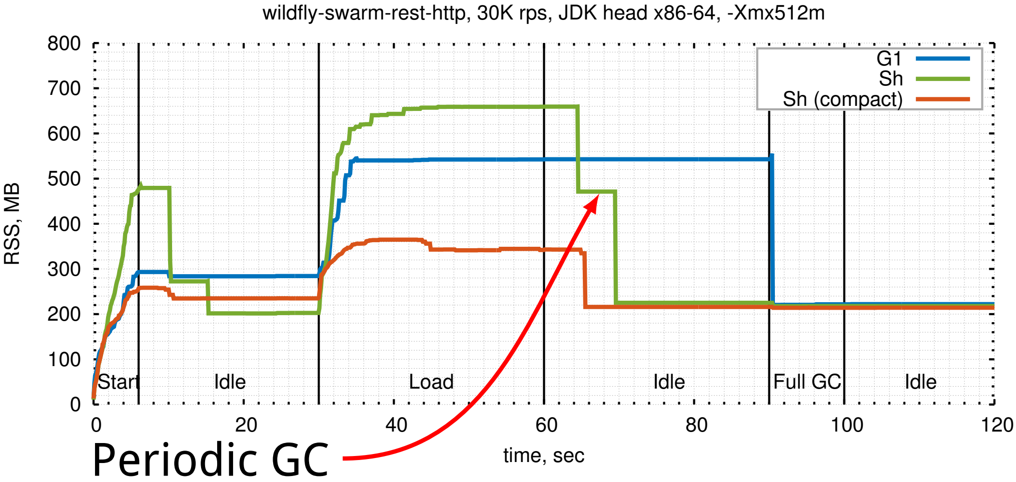
Aggressive expansion

Footprint: Heap Sizing



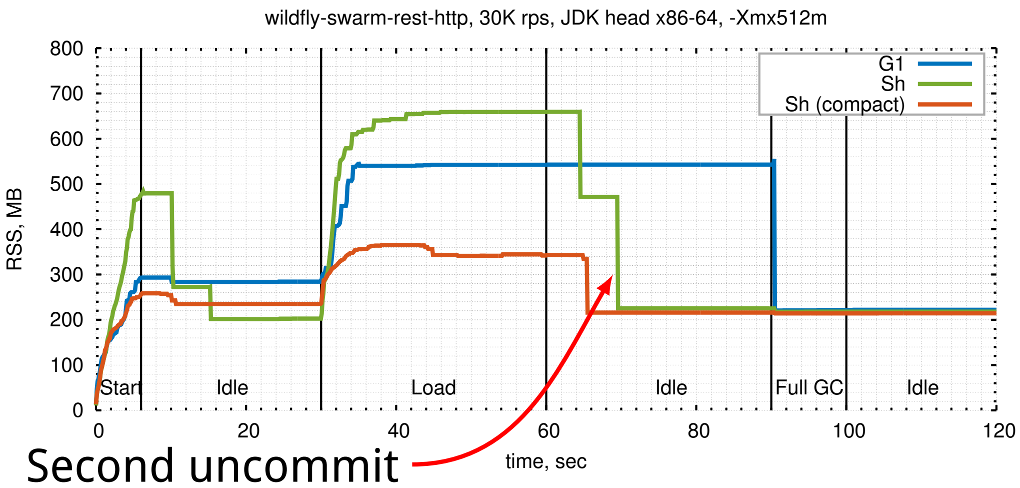
First uncommit

Footprint: Heap Sizing

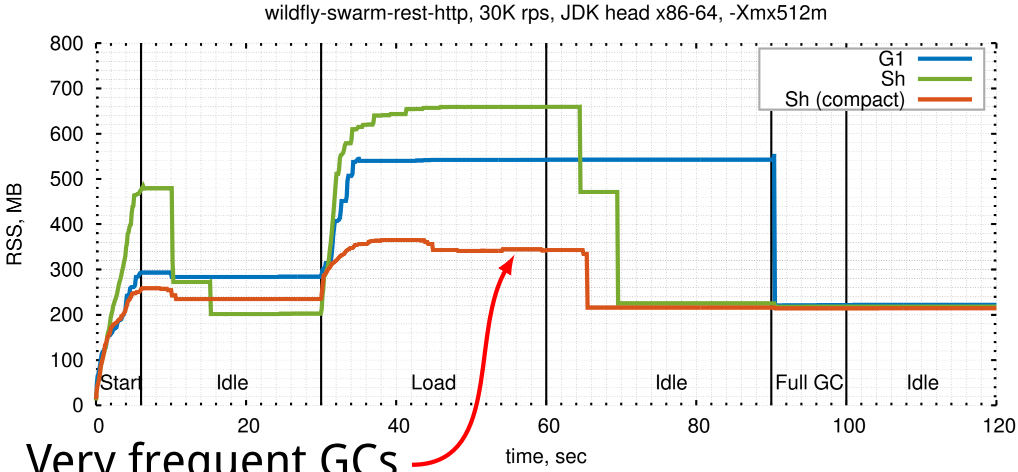


redhat.

Footprint: Heap Sizing

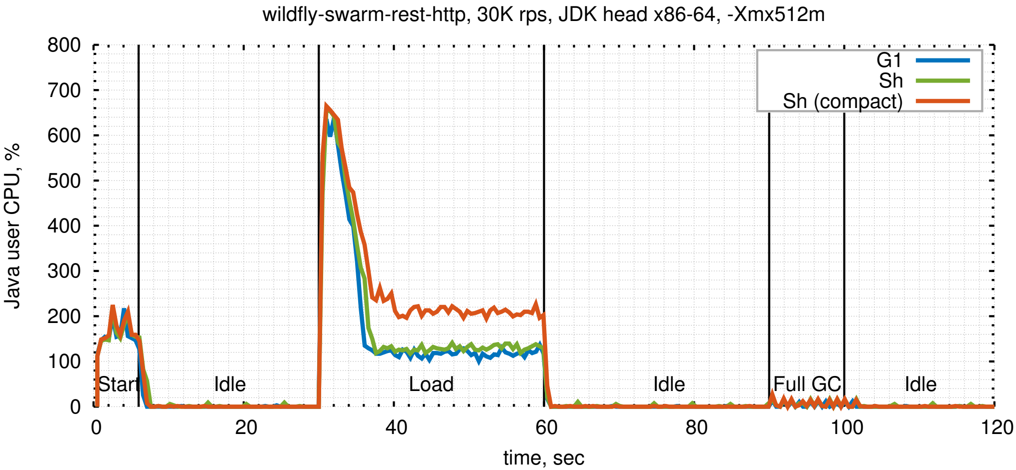


Footprint: Heap Sizing



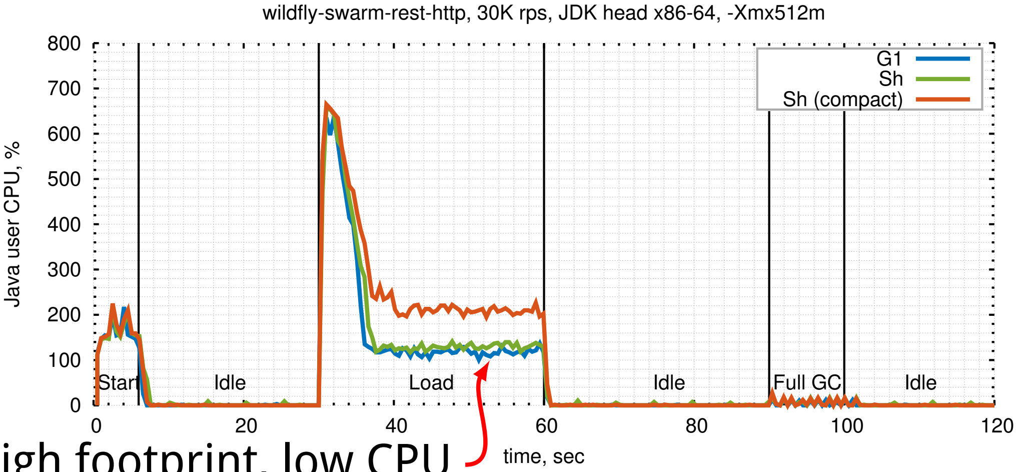
Very frequent GCs

Footprint: CPU Time Tradeoffs



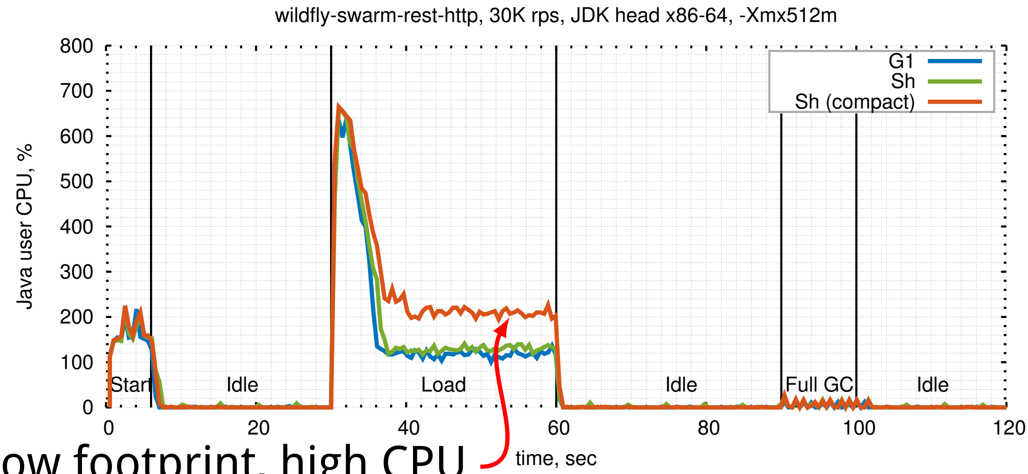
Footprint: CPU Time Tradeoffs

Footprint: CPU Time Tradeoffs



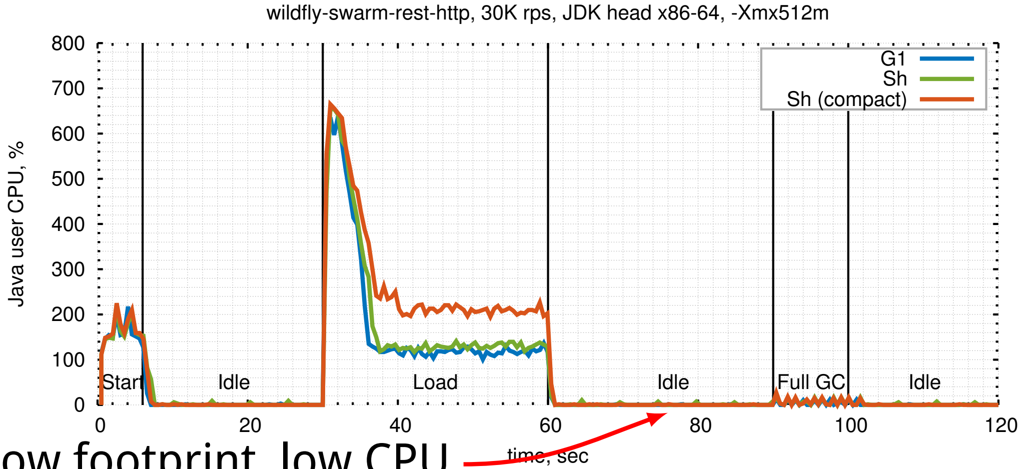
High footprint, low CPU

Footprint: CPU Time Tradeoffs



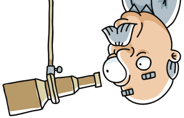
Low footprint, high CPU

Footprint: CPU Time Tradeoffs



Low footprint, low CPU

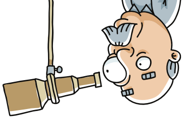
Footprint: Footprint Tips



1. Use GCs that can predictably size the heap
   - All current OpenJDK GCs have adaptive sizing
   - Most of them give back memory reluctantly

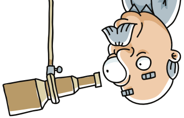
redhat.

Footprint: Footprint Tips



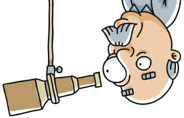
1. Use GCs that can predictably size the heap
   - All current OpenJDK GCs have adaptive sizing
   - Most of them give back memory reluctantly
2. Tune GC for lower footprint
   - Configure smaller `-Xms` and `-Xmx`
   - Tune uncommit delays, periodic GCs
     Ex.: `-XX:ShenandoahGCHeuristics=compact`

Footprint: Footprint Tips



1. Use GCs that can predictably size the heap
   - All current OpenJDK GCs have adaptive sizing
   - Most of them give back memory reluctantly
2. Tune GC for lower footprint
   - Configure smaller `-Xms` and `-Xmx`
   - Tune uncommit delays, periodic GCs
     Ex.: `-XX:ShenandoahGCHeuristics=compact`
3. Exploit GC and infra improvements
   - Java Agents that bash GC with Full GCs on idle
   - Modern GCs that recycle memory better
     Ex.: Shenandoah (JDK 8+), G1 (JDK 12+), ZGC (JDK 13+)

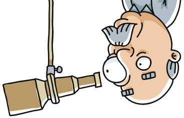
Footprint: Observations



1. Footprint story is nuanced
   - First-order effect: heap sizing policies
   - Second-order effects: per-object and per-reference overheads

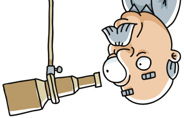
redhat.

Footprint: Observations



1. Footprint story is nuanced
   - First-order effect: heap sizing policies
   - Second-order effects: per-object and per-reference overheads
2. Beware of footprint surprises
   - Universal surprise: GCs need free memory to breathe
   - G1 surprise: native overhead (much improved in later versions)
   - ZGC surprise: no compressed oops (design disadvantage)

Footprint: Observations



1. Footprint story is nuanced
   - First-order effect: heap sizing policies
   - Second-order effects: per-object and per-reference overheads
2. Beware of footprint surprises
   - Universal surprise: GCs need free memory to breathe
   - G1 surprise: native overhead (much improved in later versions)
   - ZGC surprise: no compressed oops (design disadvantage)
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

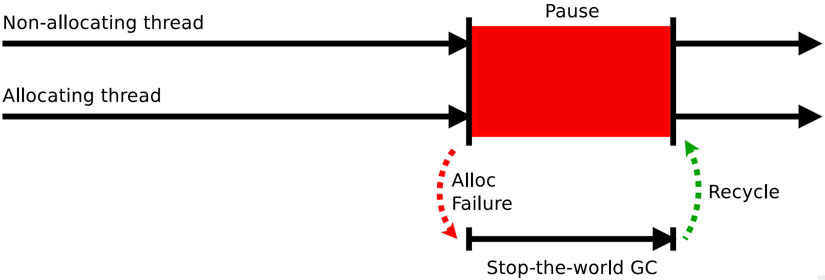
Problem:
Concurrent GC needs breathing room to succeed,
while applications allocate like madmen

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

redhat.

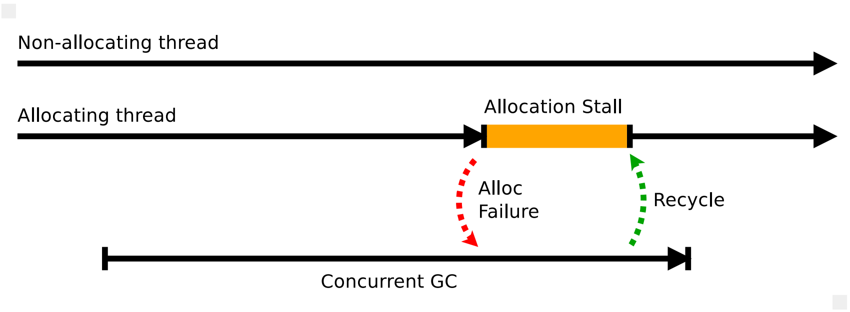
Pacing: STW GC Control Loop



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

redhat.

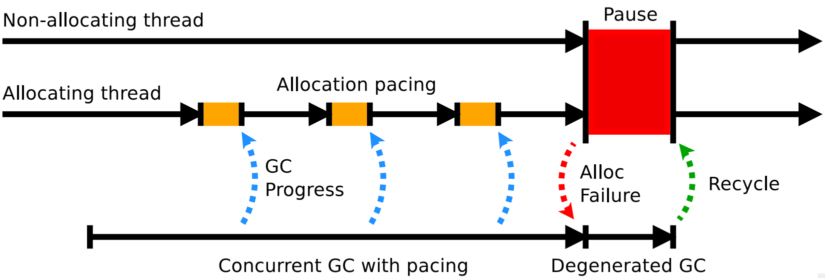
Pacing: Simple Conc GC Control Loop



- Memory is exhausted $\Rightarrow$ 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*

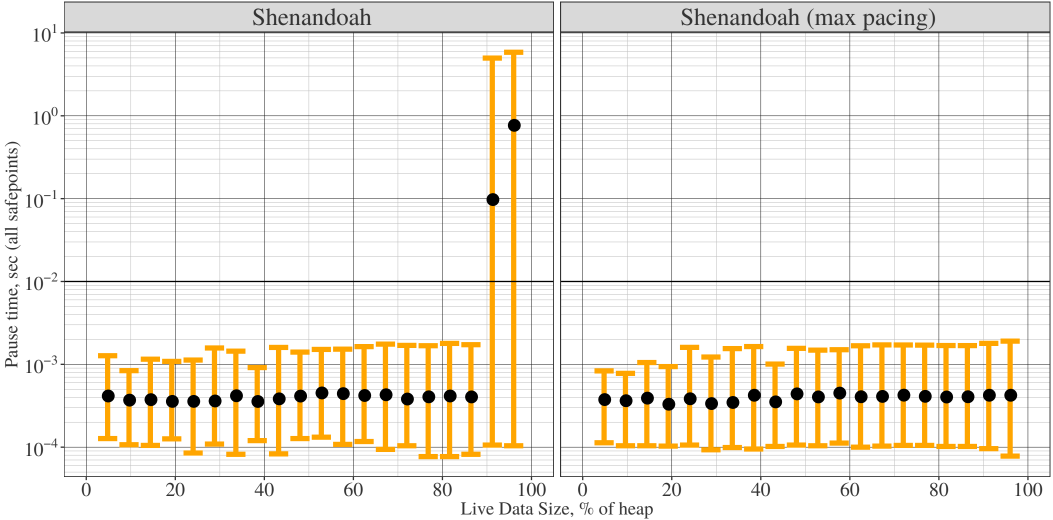
redhat.

Pacing: Shenandoah Control Loop

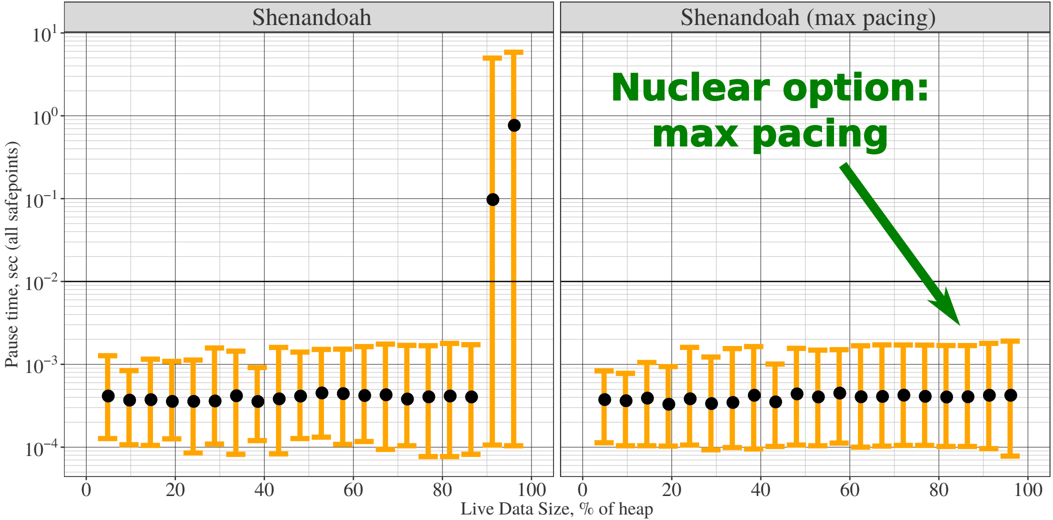


- Incremental pacing stalls allocations a bit at a time
- If AF happens, «degenerates»: completes under STW
- Pacing introduces latency, but the capped one

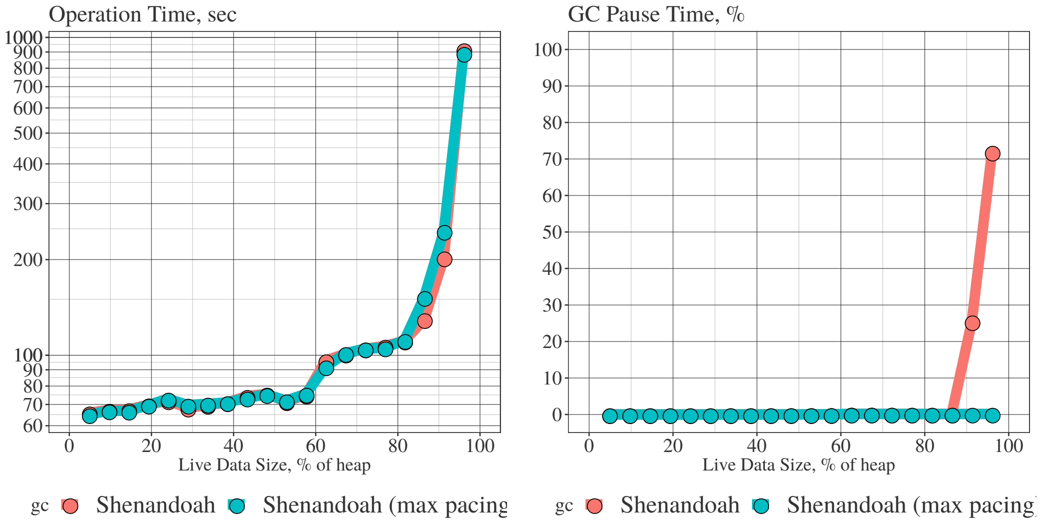
Pacing: Nuclear Option, Pauses



Pacing: Nuclear Option, Pauses

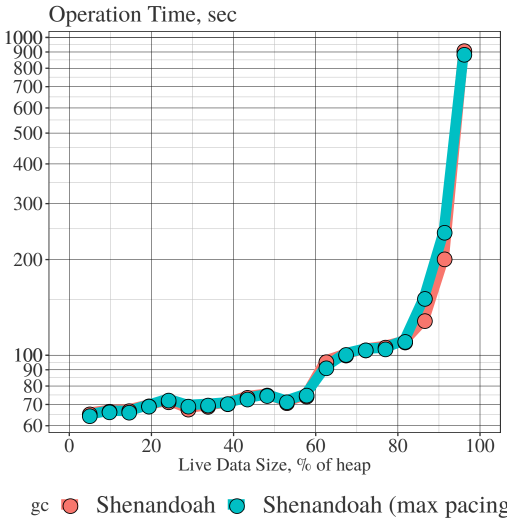


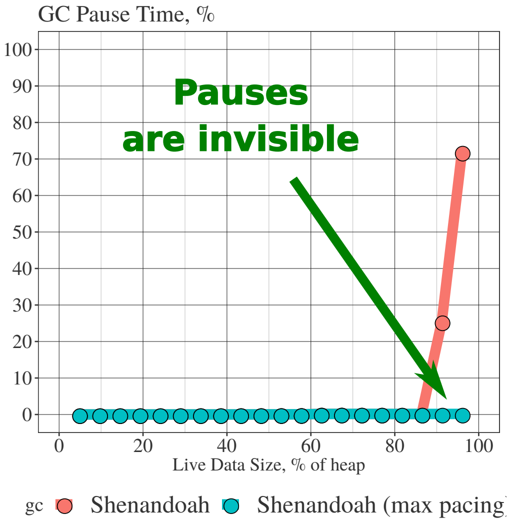
Pacing: Nuclear Option, Times



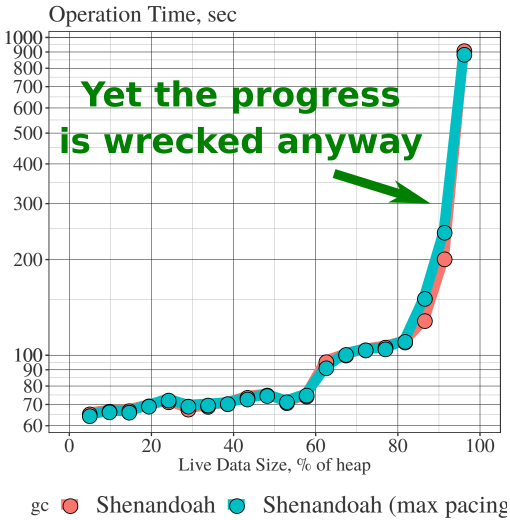
redhat

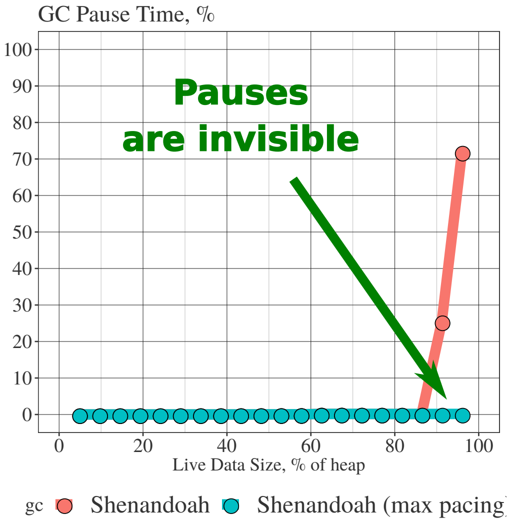
Pacing: Nuclear Option, Times



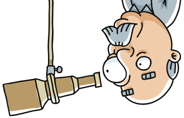


Pacing: Nuclear Option, Times



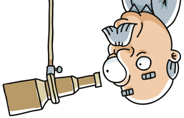


Pacing: Observations



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

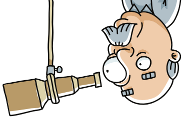
Pacing: Observations



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:

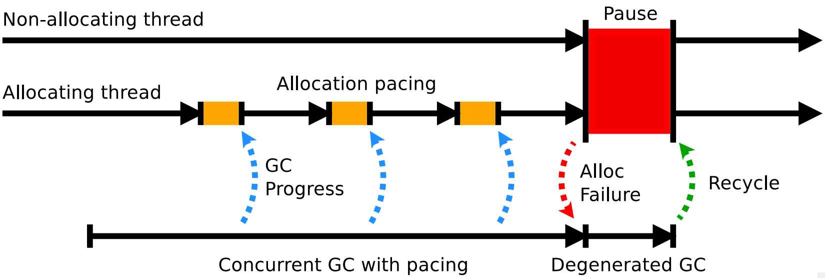
Concurrent GC needs breathing room to succeed,
while applications allocate like madmen

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

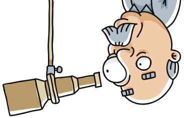
redhat.

Handling Failures: Shenandoah Control Loop



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

Handling Failures: Degenerated GC

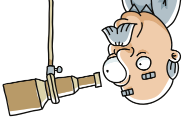


```
Pause Init Update Refs 0.034ms
Cancelling GC: Allocation Failure
Concurrent update references 7265M->8126M(8192M) 248.467ms
Pause Degenerated GC (Update Refs) 8126M->2716M(8192M) 29.787ms
```

- 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

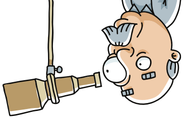
Handling Failures: Full GC

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.

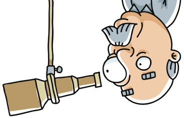
Handling Failures: Observations



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

redhat.

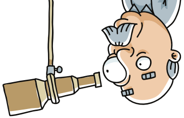
Handling Failures: Observations



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

redhat

Handling Failures: Observations

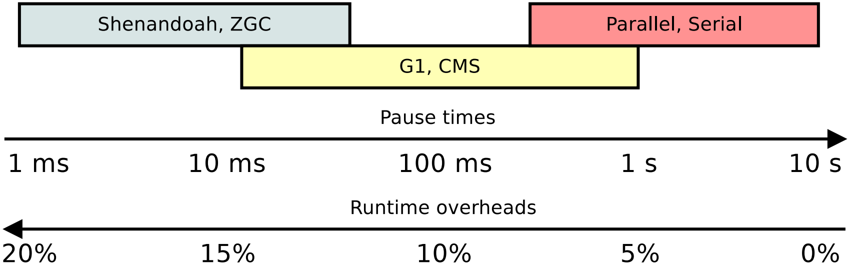


1. Being fully concurrent is nice, but own the failures
   - The failures will happen, accept it
   - «Our perfect GC melted down, because you forgot this magic VM option(, stupid)» flies only that far
2. Graceful and observable degradation is key
   - Getting worse incrementally is better than falling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

Conclusion (I)

Conclusion (I): In Single Picture

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



Choose this for your workload!

Conclusion (I): In Single Paragraph

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

Conclusion (I): 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 (I): 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 (I): 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 (I): Releases

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

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

- Dev follows latest JDK, backports to 13, **11u, 8u**
- 8u backport ships in RHEL 7.4+, Fedora 24+
- 11u backport ships in Fedora 27+
- Nightly development builds (tarballs, Docker images)

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

End of Part I

Part II

Pauses

Pauses: When Everything Is Good

LRUFragger, 100 GB heap, $\approx$ 80 GB LDS:

```
Pause Init Mark 0.437ms
Concurrent marking 76780M->77260M(102400M) 700.185ms
Pause Final Mark 0.698ms
Concurrent cleanup 77288M->77296M(102400M) 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 0.351ms
Concurrent cleanup 85928M->56620M(102400M) 14.316ms
```


redhat.

Pauses: When Everything Is Good

LRUFragger, 100 GB heap, $\approx$ 80 GB LDS:

```
Pause Init Mark 0.437ms
Concurrent marking 76780M->77260M(102400M) 700.185ms
Pause Final Mark 0.698ms
Concurrent cleanup 77288M->77296M(102400M) 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 0.351ms
Concurrent cleanup 85928M->56620M(102400M) 14.316ms
```

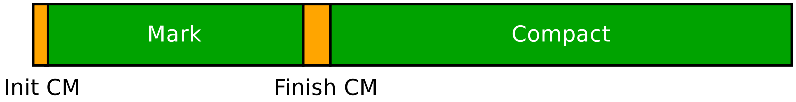
Pauses: When Something Is Not So Good

Worst-case cycle in one of the workloads:

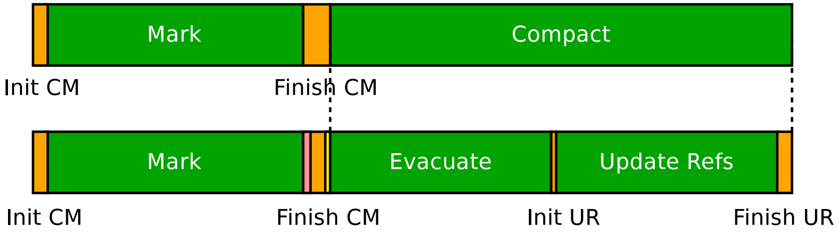
```
Pause Init Mark 4.915ms
Concurrent marking 794M->794M(4096M) 95.853ms
Pause Final Mark 30.876ms
Concurrent cleanup 795M->795M(4096M) 0.170ms
Concurrent evacuation 795M->796M(4096M) 0.197ms
Pause Init Update Refs 0.029ms
Concurrent update references 796M->796M(4096M) 28.707ms
Pause Final Update Refs 2.764ms
Concurrent cleanup 796M->792M(4096M) 0.372ms
```


redhat.

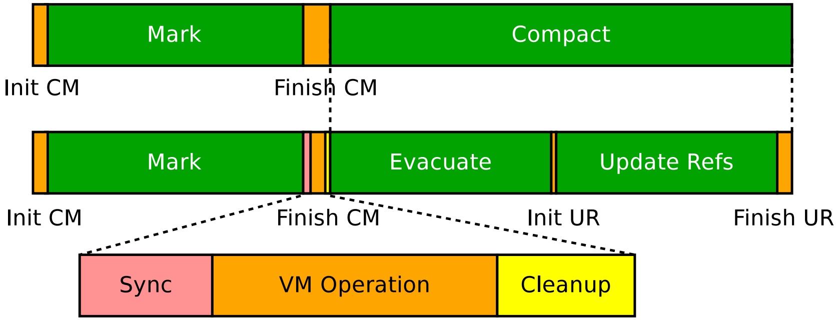
Pauses: Pause Taxonomy



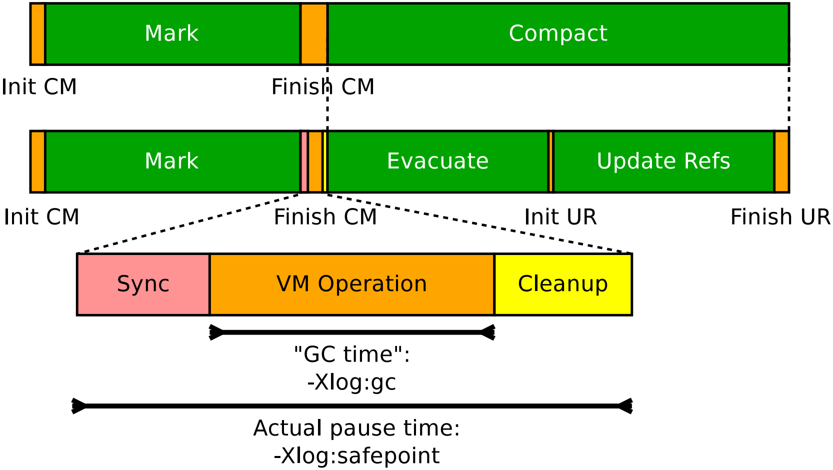
Pauses: Pause Taxonomy



Pauses: Pause Taxonomy



Pauses: Pause Taxonomy

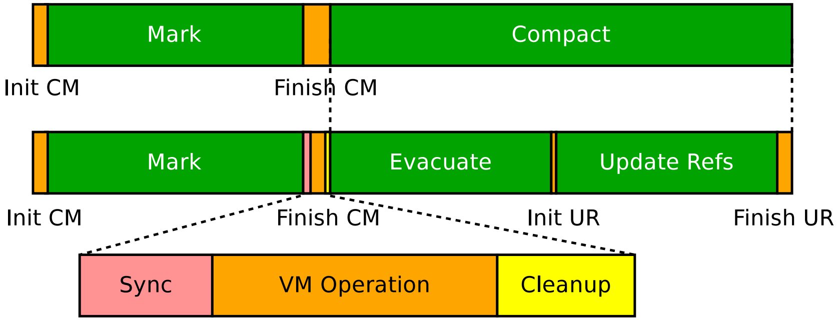


Safepoint Prolog: Ideas

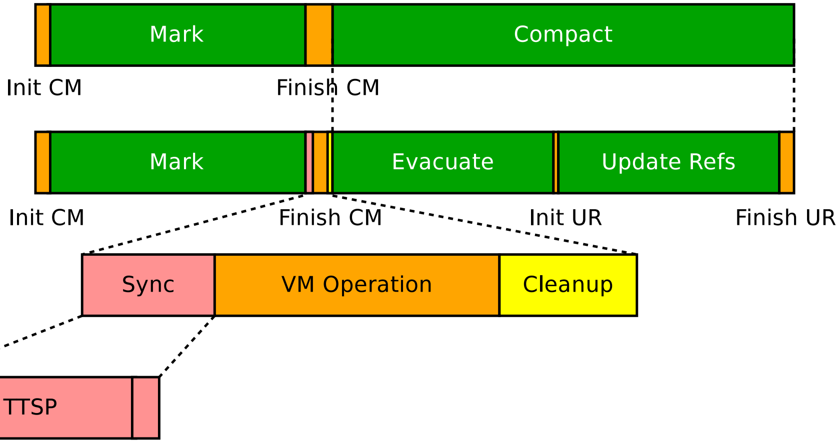
1. Make sure changing the VM state is **safe**
2. Enable **cooperative** thread suspension
3. Have the known state points: e.g. where are the **pointers**

```
push %rbx
LOOP:
 inc %rax
 test (%rip, 0x488313) # safepoint poll
                       # %rbx is ptr, (%rsp) is ptr
 cmp %rax, (%rbx, 8)
 jl  LOOP
```

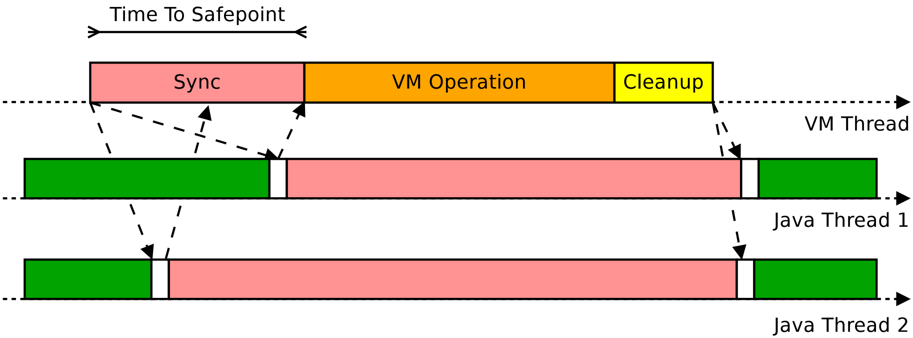
TTSP: Pause Taxonomy



TTSP: Pause Taxonomy



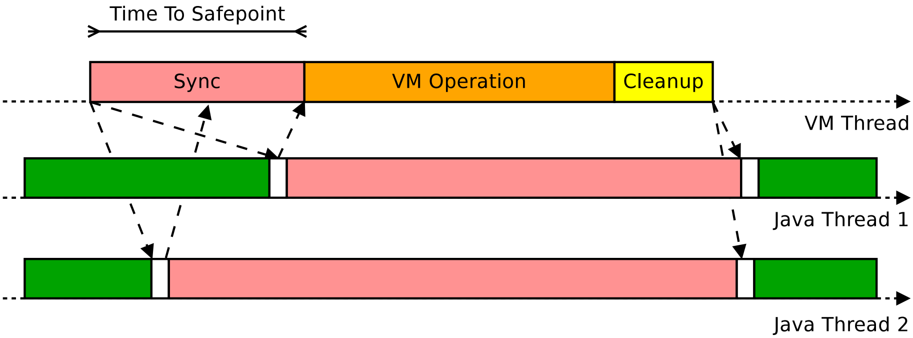
TTSP: Definition



TTSP: Time between VM Thread decision to make a safepoint, until all Java threads have reacted

redhat.

TTSP: Definition



Some threads are still happily executing after safepoint request, having not observed it yet

TTSP: Long Loops

In tight loops, safepoint poll costs are very visible!
Solution: eliminate safepoint polls in short cycles

```
LOOP:
 inc %rax
 cmp %rax, $100
 jl  LOOP
```


redhat

TTSP: Long Loops

In tight loops, safepoint poll costs are very visible!
Solution: eliminate safepoint polls in short cycles

```
LOOP:
 inc %rax
 cmp %rax, $100
 jl  LOOP
```

How short is short, though?

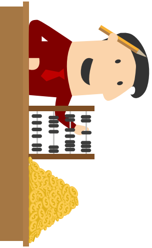
TTSP: Long Loops

In tight loops, safepoint poll costs are very visible!
Solution: eliminate safepoint polls in short cycles

```
LOOP:
 inc %rax
 cmp %rax, $100
 jl  LOOP
```

How short is short, though?
Hotspot's answer: Counted loops are short!

TTSP: Long Loops

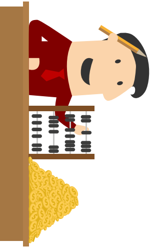


```
int[] arr;

@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)
```

TTSP: `-XX:+UseCountedLoopSafepoints`

The magic VM option to keep the safepoints in counted loops!
...with quite some throughput overhead :(



```
# -XX:+UseShenandoahGC -XX:-UseCountedLoopSafepoints
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:+UseCountedLoopSafepoints
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)
```

TTSP: Loop Strip Mining

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!

redhat

TTSP: 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)
```

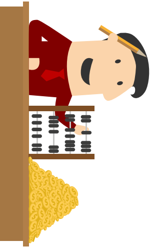

redhat

TTSP: 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)
```


redhat

TTSP: 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)
```

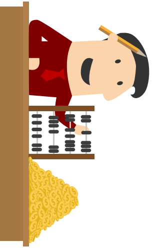
TTSP: Runnable Threads

The suspension is cooperative:
every *runnable* thread has to react to a safepoint request

- Non-runnable threads are already considered at safepoint: all those idle threads that are WAITING, TIMED_WAITING, BLOCKED, etc are safe already
- Lots of runnable threads: each thread should get scheduled to roll to safepoint

TTSP: Runnable Threads Test

```
for (int i : arr) {
  r = (i * 1664525 + 1013904223 + r) % 1000;
}
```

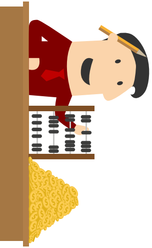


Each thread needs scheduling to roll to safepoint:

```
# java -XX:+UseShenandoahGC -Dthreads=16
Total Pauses (G) = 0.30 s (a =  1529 us)
Total Pauses (N) = 0.23 s (a =  1166 us)
```

TTSP: Runnable Threads Test

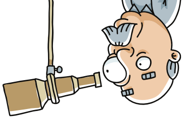
```
for (int i : arr) {
  r = (i * 1664525 + 1013904223 + r) % 1000;
}
```



Each thread needs scheduling to roll to safepoint:

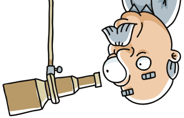
```
# java -XX:+UseShenandoahGC -Dthreads=16
Total Pauses (G) = 0.30 s (a =  1529 us)
Total Pauses (N) = 0.23 s (a =  1166 us)

# java -XX:+UseShenandoahGC -Dthreads=1024
Total Pauses (G) = 5.14 s (a = 36689 us)
Total Pauses (N) = 0.22 s (a =  1564 us)
```

TTSP: Latency Tips



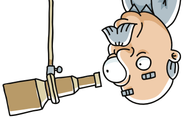
1. Safepoint monitoring is your friend
   - Enable `-XX:+PrintSafepointStatistics` along with GC logs
   - Use GC that tells you gross pause times that include safepoints

TTSP: Latency Tips



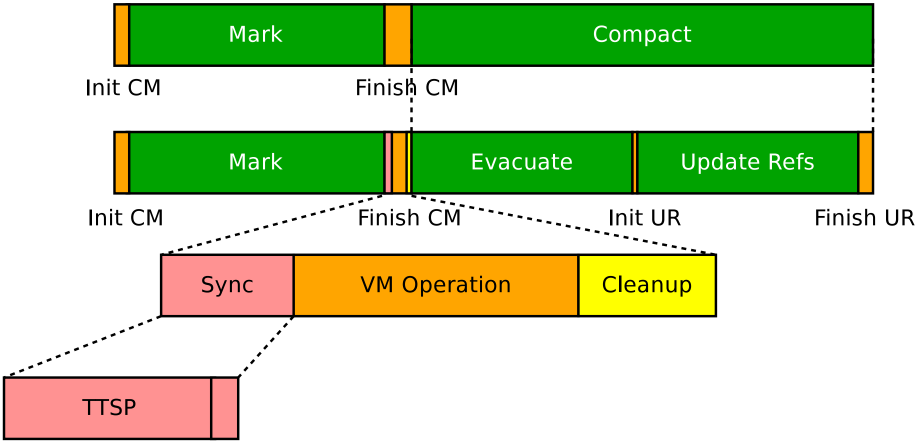
1. Safepoint monitoring is your friend
   - Enable `-XX:+PrintSafepointStatistics` along with GC logs
   - Use GC that tells you gross pause times that include safepoints
2. Trim down the number of runnable threads
   - Overwhelming the system is never good
   - Use shared thread pools, and then share the thread pools

TTSP: Latency Tips

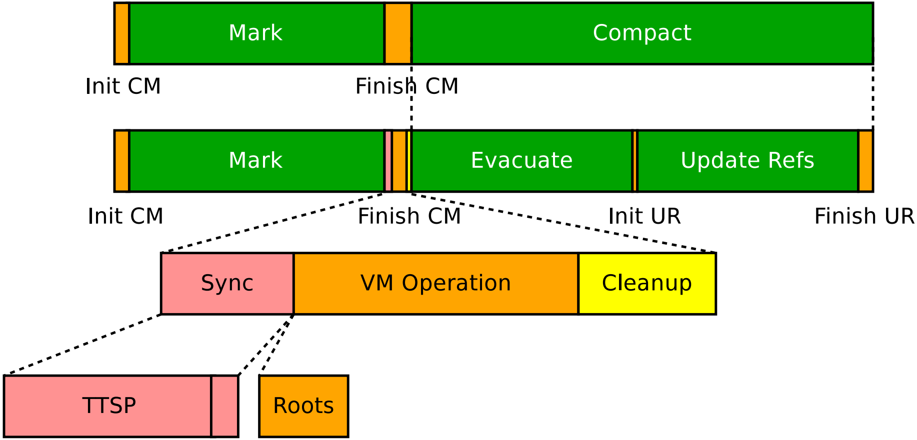


1. Safepoint monitoring is your friend
   - Enable `-XX:+PrintSafepointStatistics` along with GC logs
   - Use GC that tells you gross pause times that include safepoints
2. Trim down the number of runnable threads
   - Overwhelming the system is never good
   - Use shared thread pools, and then share the thread pools
3. Watch TTSP due to code patterns, and then enable:
   - `-XX:+UseCountedLoopSafepoints` for JDK 9-
   - `-XX:LoopStripMiningIters=#` for JDK 10+

GC Roots: Pause Taxonomy



GC Roots: Pause Taxonomy



GC Roots: What Are They, Dude

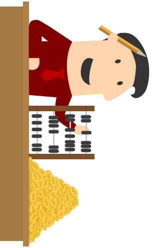
Def: «GC Root», slot with implicitly reachable object

Def: «Root set», the complete set of GC roots

«Implicitly reachable» = reachable without Java objects

- Popular: static fields, «thread stacks», «local variables»
- Less known: anything that holds Java refs in native code

GC Roots: There Are Lots of Them



```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Init Mark (G)           =  0.07 s (a = 7011 us)
Pause Init Mark (N)           =  0.06 s (a = 6052 us)
  Scan Roots                  =  0.06 s (a = 5887 us)
    S: Thread Roots           =  0.01 s (a = 1031 us)
    S: String Table Roots     =  0.02 s (a = 1647 us)
    S: Universe Roots         =  0.00 s (a =    2 us)
    S: JNI Roots              =  0.00 s (a =    8 us)
    S: JNI Weak Roots         =  0.00 s (a =  275 us)
    S: Synchronizer Roots     =  0.00 s (a =    4 us)
    S: Management Roots       =  0.00 s (a =    2 us)
    S: System Dict Roots      =  0.00 s (a =  329 us)
    S: CLDG Roots             =  0.02 s (a = 1583 us)
    S: JVMTI Roots            =  0.00 s (a =    1 us)
```

Thread Roots: Why

```
void k() {
  Object o1 = get();
  m();
  workWith(o1);
}

void m() {
  Object o2 = get();
  // <gc safepoint here>
  workWith(o2);
}
```

Once we hit the safepoint, we have to figure that both `o1` and `o2` are reachable

Need to scan all activation records up the stack looking for references

Thread Roots: Trick 1, Local Var Reachability[4]

```
void m() {
  Object o2 = get();
  // <gc safepoint here>
  doSomething();
}
```

Trick: computing the oop maps does account the variable liveness!

Here, `o2` would not be exposed at safepoint, making the object reclaimable

[4] https://shipilev.net/jvm/anatomy-quarks/8-local-var-reachability/

Thread Roots: Trick 2, Saving Grace

```
"thread-100500" #100500 daemon prio=5 os_prio=0 tid=0x13371337
nid=0x11902 waiting on condition TIMED_WAITING
at sun.misc.Unsafe.park(Native Method)
- parking to wait for  <0x0000000081e39398>
at java.util.concurrent.locks.LockSupport.parkNanos
at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObj
at java.util.concurrent.LinkedBlockingQueue.poll
at java.util.concurrent.ThreadPoolExecutor.getTask
at java.util.concurrent.ThreadPoolExecutor.runWorker
at java.util.concurrent.ThreadPoolExecutor$Worker.run
at java.lang.Thread.run
```

Most threads are stopped at shallow stacks

Thread Roots: GC Handling

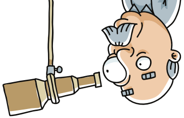
GC threads scan Java threads in parallel:
N GC threads scan K Java threads

$$\texttt{Thread Roots Count} \approx$$
$$\approx \texttt{Average Stack Depth} \times \texttt{Java Thread Count}$$

Corollaries:

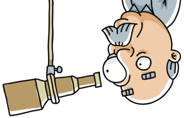
- $\texttt{Java Thread Count} \leqslant \texttt{Count(CPU)}$ – excellent
- Small $\texttt{Average Stack Depth}$ – excellent

Thread Roots: Latency Tips



1. Make sure only a few threads are active
   - Ideally, N_CPU threads, sharing the app load
   - Natural with thread-pools: most threads are parked at shallow stack depths

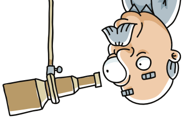
Thread Roots: Latency Tips



1. Make sure only a few threads are active
   - Ideally, N_CPU threads, sharing the app load
   - Natural with thread-pools: most threads are parked at shallow stack depths
2. Trim down the thread stack depths
   - Calling into thousands of methods exposes lots of locals
   - Tune up inlining: less frames to scan

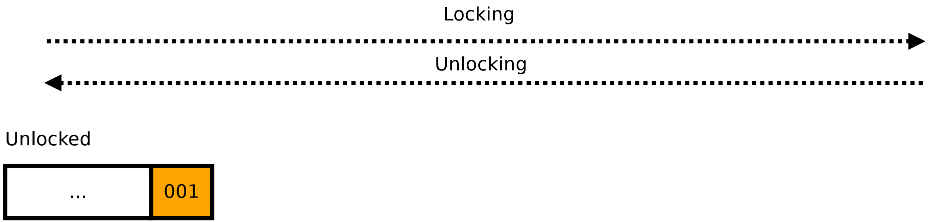
redhat.

Thread Roots: Latency Tips



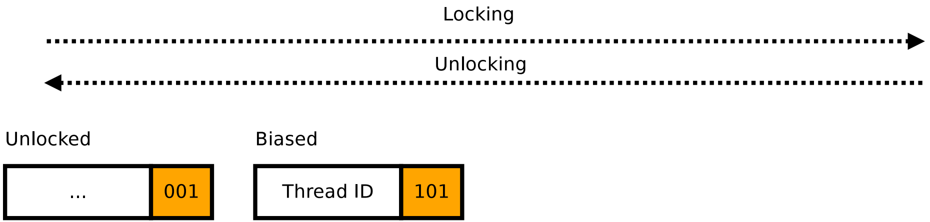
1. Make sure only a few threads are active
   - Ideally, N_CPU threads, sharing the app load
   - Natural with thread-pools: most threads are parked at shallow stack depths
2. Trim down the thread stack depths
   - Calling into thousands of methods exposes lots of locals
   - Tune up inlining: less frames to scan
3. Wait for and exploit runtime improvements
   - Grey thread roots and concurrent root scans?
   - Per-thread scans with handshakes?

Sync Roots: Why



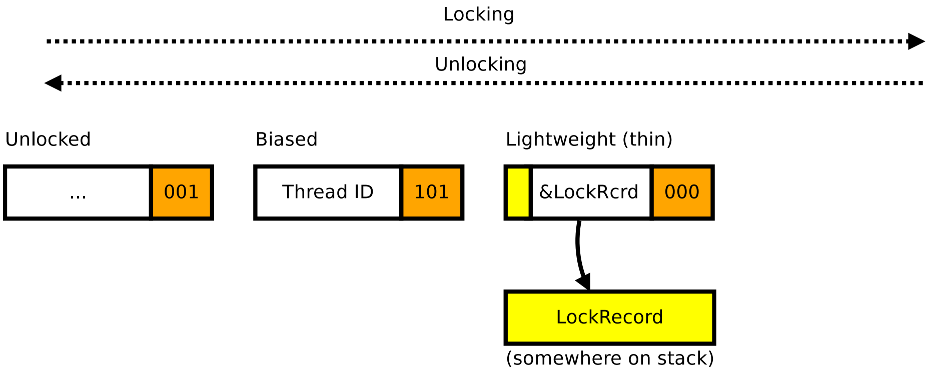
Progressively heavier lock metadata: unlocked

Sync Roots: Why



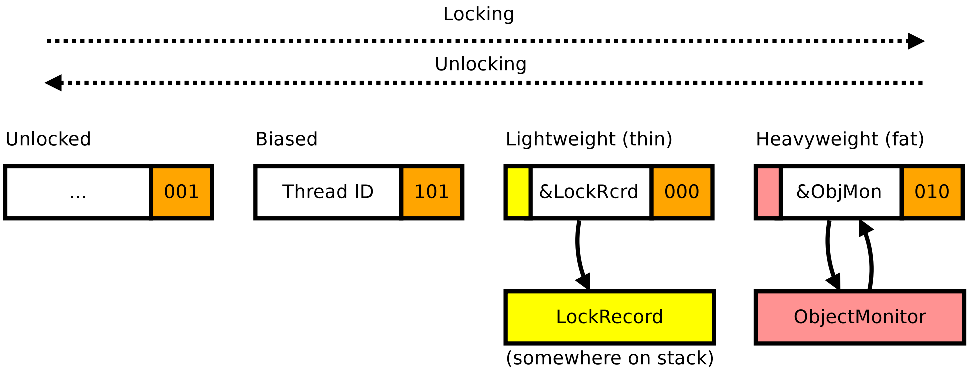
Progressively heavier lock metadata: unlocked, biased

Sync Roots: Why



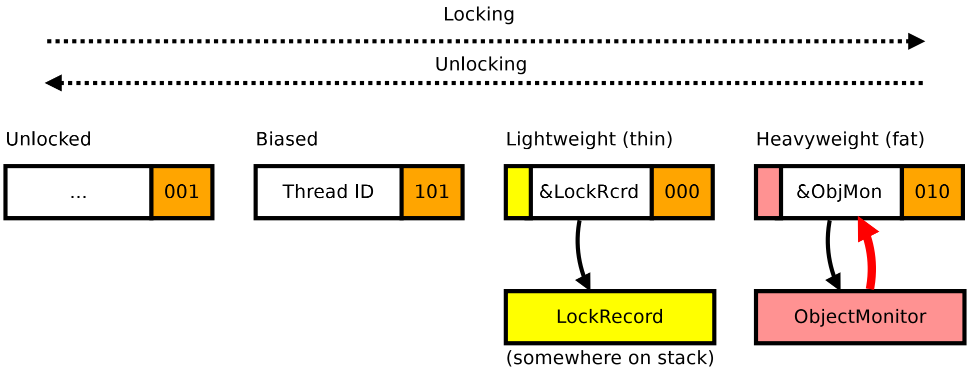
Progressively heavier lock metadata:
unlocked, biased, thin locks

Sync Roots: Why



Ultimately, `ObjectMonitor` that associates object with its fat native synchronizer, in both directions

Sync Roots: Why



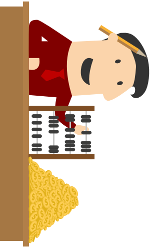
Ultimately, `ObjectMonitor` that associates object with its fat native synchronizer, in both directions

Sync Roots: Syncie-Syncie Test

```java
@Benchmark
public void test() throws InterruptedException {
  for (SyncPair pair : pairs) {
    pair.move();
  }
}

static class SyncPair {
  int x, y;
  public synchronized void move() {
    x++; y--;
  }
}
```


redhat.

Sync Roots: Depletion Test

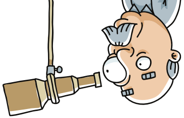


```
static class SyncPair {
  int x, y;
  public synchronized void move() {
    x++; y--;
  }
}
```

```
# java -XX:+UseShenandoahGC -Dcount=1'000'000
Pause Init Mark (N)          =  0.00 s (a =  2446 us)
  Scan Roots                 =  0.00 s (a =  2223 us)
    S: Synchronizer Roots    =  0.00 s (a =   896 us)
```

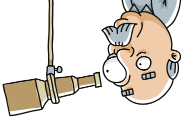

redhat.

Sync Roots: Latency Tips



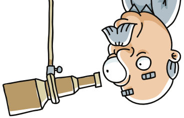
1. Avoid **contended** locking on **lots** of `synchronized`-s
   - Most applications do seldom contention on few monitors
   - Replace with `j.u.c.Lock`, Atomics, `VarHandle`, etc. otherwise

Sync Roots: Latency Tips



1. Avoid **contended** locking on **lots** of `synchronized`-s
   - Most applications do seldom contention on few monitors
   - Replace with `j.u.c.Lock`, Atomics, `VarHandle`, etc. otherwise
2. Have more frequent safepoints
   - Counter-intuitive, but may keep inflated monitors count at bay
   - (More on that later)

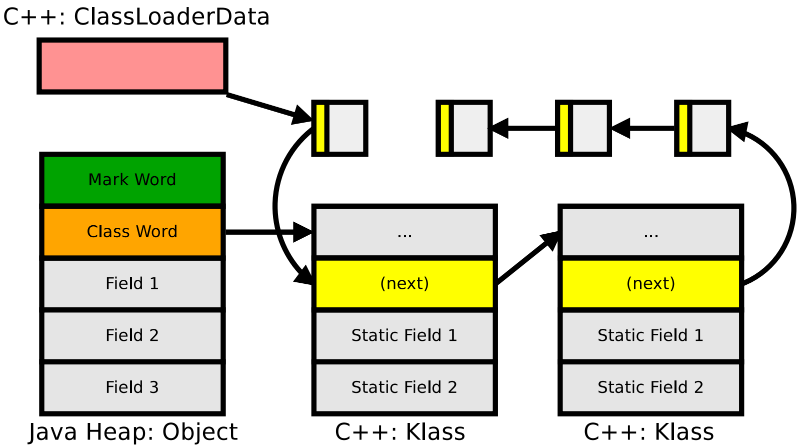
Sync Roots: Latency Tips



1. Avoid **contended** locking on **lots** of `synchronized`-s
   - Most applications do seldom contention on few monitors
   - Replace with `j.u.c.Lock`, Atomics, `VarHandle`, etc. otherwise
2. Have more frequent safepoints
   - Counter-intuitive, but may keep inflated monitors count at bay
   - (More on that later)
3. Exploit runtime improvements
   - `-XX:+MonitorInUseLists`, enabled by default since JDK 9
   - Piggybacking on thread scans (Shenandoah)

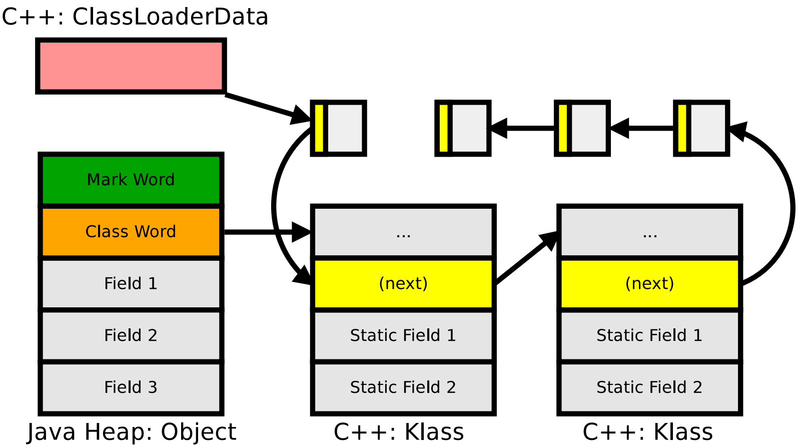
redhat.

Class Roots: Why



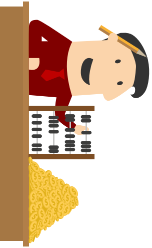
Static fields are stored in class mirrors outside the objects

Class Roots: Why



Even without instances, we need to visit static fields

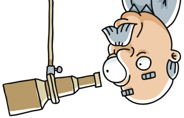
Class Roots: Enterprise Hello World Test



```
@Setup
public void setup() throws Exception {
  classes = new Class[count];
  for (int c = 0; c < count; c++) {
    classes[c] = ClassGenerator.generate();
  }
}
```

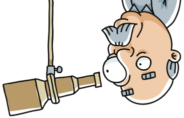
```
# java -XX:+UseShenandoahGC -Dcount=100'000
Pause Init Mark (G) = 0.17 s (a = 6068 us)
Pause Init Mark (N) = 0.15 s (a = 5484 us)
  Scan Roots        = 0.15 s (a = 5233 us)
    S: CLDG Roots   = 0.01 s (a =  432 us)
```

Class Roots: Latency Tips



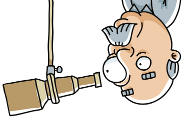
1. Avoid too many classes
   - Merge related classes together, especially autogenerated
   - If not avoidable, make sure classes are unloaded

Class Roots: Latency Tips



1. Avoid too many classes
   - Merge related classes together, especially autogenerated
   - If not avoidable, make sure classes are unloaded
2. Avoid too many classloaders
   - Roots are walked by CLData, more CLs, more CLData to walk
   - If not avoidable, make sure CLs are garbage-collected

Class Roots: Latency Tips



1. Avoid too many classes
   - Merge related classes together, especially autogenerated
   - If not avoidable, make sure classes are unloaded
2. Avoid too many classloaders
   - Roots are walked by CLData, more CLs, more CLData to walk
   - If not avoidable, make sure CLs are garbage-collected
3. Exploit runtime improvements
   - Shenandoah: parallel classloader data scans
   - JDK 9+: less and less oops in native structures
   - JDK 12+: concurrent class scans

redhat.

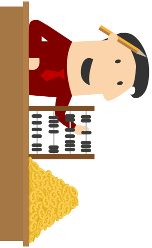
String Table Roots: Why

`StringTable` is native, and references `String` objects

```
class String {
  ...
  public native String intern();
  ...
}

class StringTable : public RehashableHashtable<oop, mtSymbol> {
  ...
  static oop intern(Handle h, jchar* chars, int length, ...);
  ...
}
```


redhat.

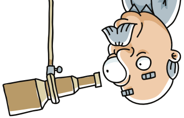
String Table Roots: Intern Test



```
@Setup
public void setup() {
  for (int c = 0; c < size; c++)
    list.add(("" + c + "root").intern());
}

@Benchmark
public Object test() { return new Object(); }
```

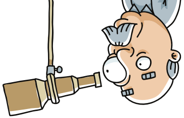
```
# jdk10/bin/java -XX:+UseShenandoahGC -Dsize=1'000'000
Pause Init Mark (G)          =  0.30 s (a = 10698 us)
Pause Init Mark (N)          =  0.29 s (a = 10315 us)
  Scan Roots                 =  0.28 s (a = 10046 us)
    S: String Table Roots =  0.25 s (a =  8991 us)
```

String Table Roots: Latency Tips



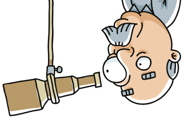
1. Do not use `String.intern()`
   - It is almost never worth it
   - Roll on your own deduplicator/interner

String Table Roots: Latency Tips



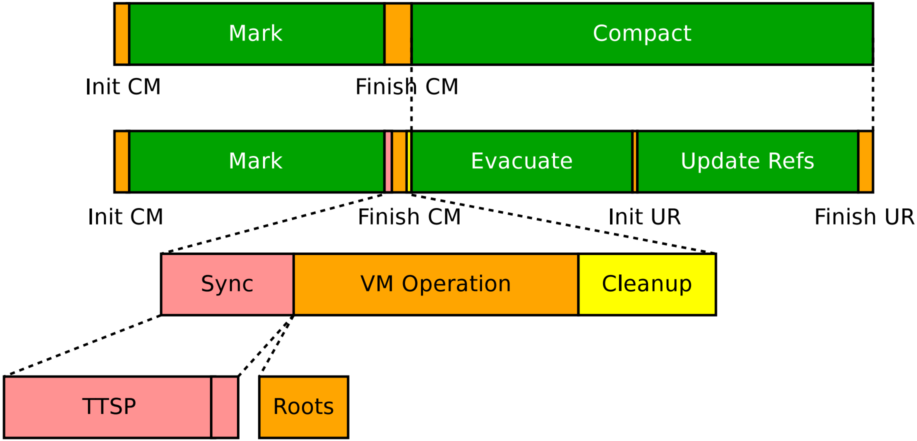
1. Do not use `String.intern()`
   - It is almost never worth it
   - Roll on your own deduplicator/interner
2. Watch out for StringTable rehashing and cleanups
   - `-XX:StringTableSize=#` is your friend here
   - Surprise: `-XX:-ClassUnloading` disables StringTable cleanup
   - Surprise: StringTable would need to rehash under STW

String Table Roots: Latency Tips

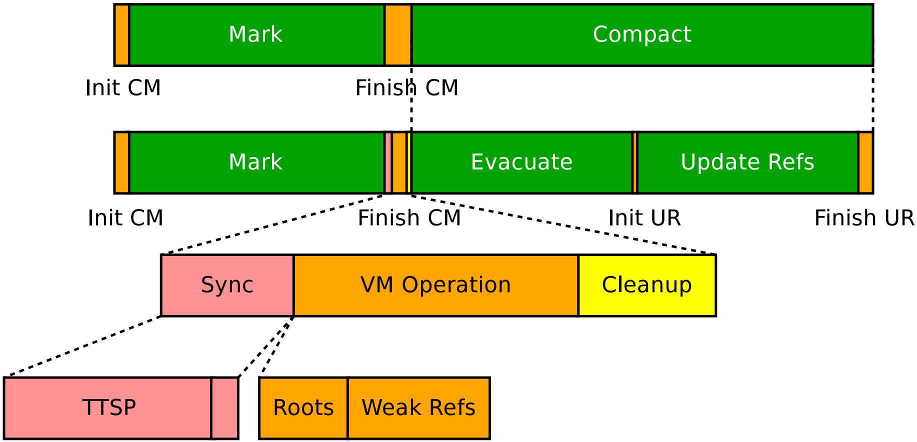


1. Do not use `String.intern()`
   - It is almost never worth it
   - Roll on your own deduplicator/interner
2. Watch out for StringTable rehashing and cleanups
   - `-XX:StringTableSize=#` is your friend here
   - Surprise: `-XX:-ClassUnloading` disables StringTable cleanup
   - Surprise: StringTable would need to rehash under STW
3. Wait for more runtime improvements
   - JDK 11+: concurrent StringTable
   - JDK 11+: resizable StringTable
   - JDK 11+: concurrent StringTable scan

Weak References: Pause Taxonomy



Weak References: Pause Taxonomy

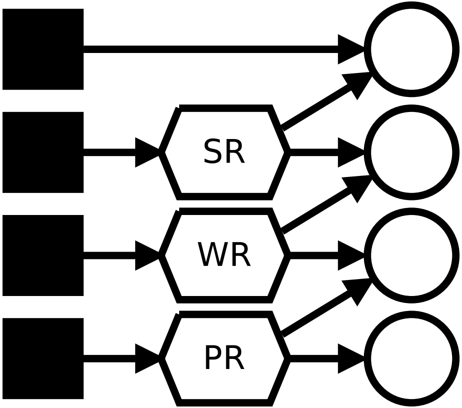


Weak References: What, How, When

The single most GC-sensitive language feature:
soft/weak/phantom references and finalizers

- Weak references have loose relation with the liveness of the target object (*referent*), can detect liveness changes
- Contrast: Strong references imply referent is always alive
- Finalizable objects are yet another synthetic weak reachability level: modeled with `j.l.ref.Finalizer`

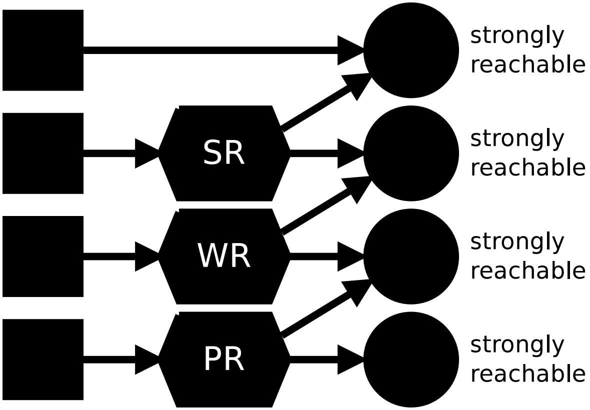
Weak References: How Do They Work?



Suppose we have the object graph where some objects are not strongly reachable

[4] e.g. treating `Reference.referent` as normal field

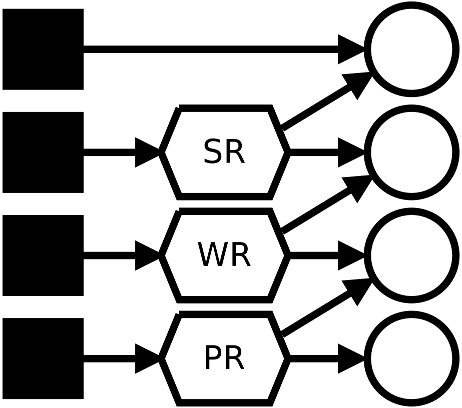
Weak References: How Do They Work?



Scanning **through**[5] the weak references yields strongly reachable heap: normal GC cycle

[5] e.g. treating `Reference.referent` as normal field

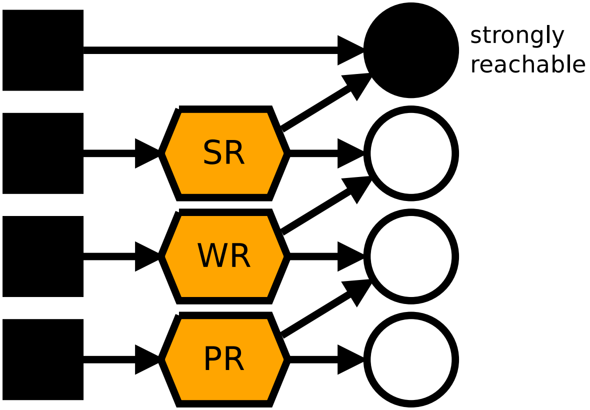
Weak References: How Do They Work?



Back to square one: start from unmarked heap...

[4] e.g. treating `Reference.referent` as normal field

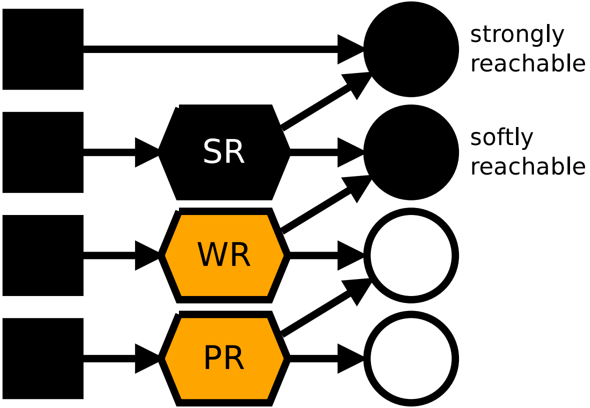
Weak References: How Do They Work?



But then, do **not** mark through the weak refs, but **discover** and record them separately

[4] e.g. treating `Reference.referent` as normal field

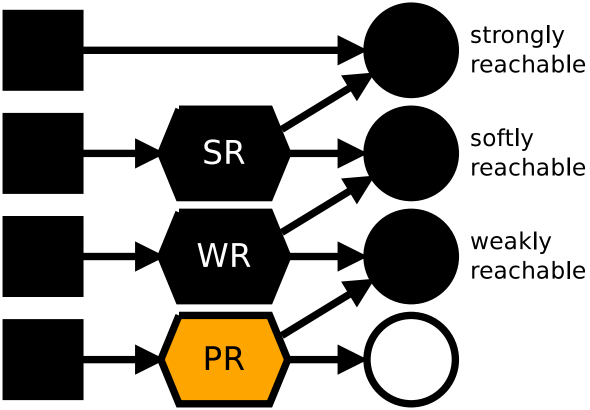
Weak References: How Do They Work?



Now, we can iterate over soft-refs, and treat all non-marked referents as softly reachable...

[4] e.g. treating `Reference.referent` as normal field

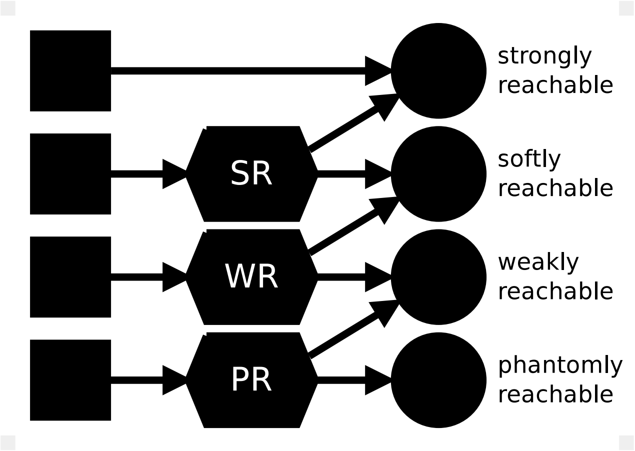
Weak References: How Do They Work?



Rinse and repeat for other subtypes, in order, and after a few iterations we have all weak refs processed

[4] e.g. treating `Reference.referent` as normal field

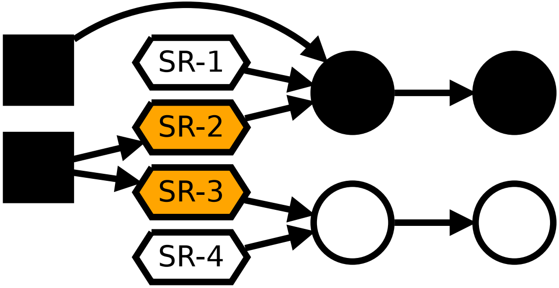
Weak References: How Do They Work?



Rinse and repeat for other subtypes, in order, and after a few iterations we have all weak refs processed

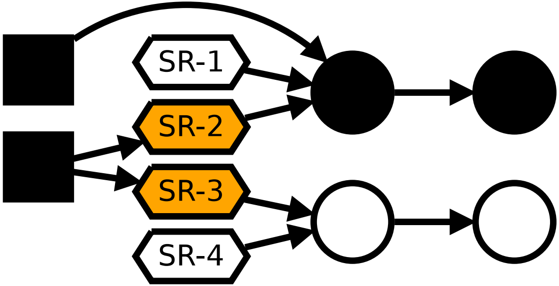
[4] e.g. treating `Reference.referent` as normal field

Weak References: Reachability Tricks



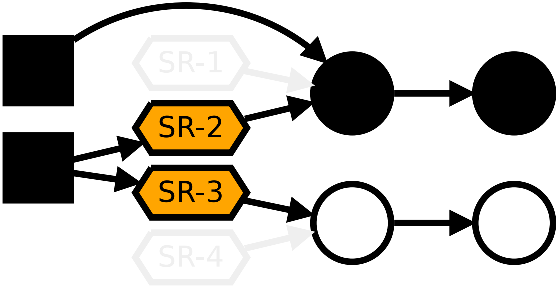
There are four cases: the reference itself can be (un)reachable, and the referent can be (un)reachable

Weak References: Reachability Tricks



SR-1 and SR-4 are unreachable.
Discovery would never visit them, stop

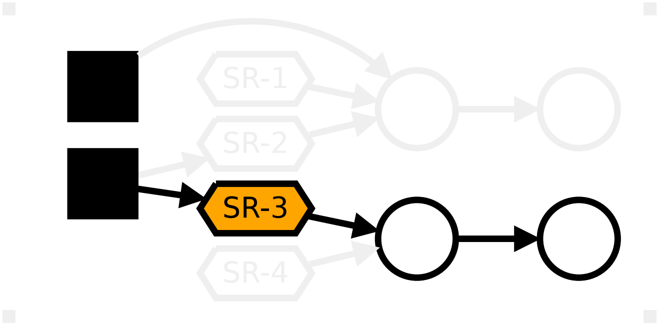
Weak References: Reachability Tricks



Trick «Precleaning»: SR-2 is reachable, and its referent is reachable. No need to handle, remove from discovered list

redhat.

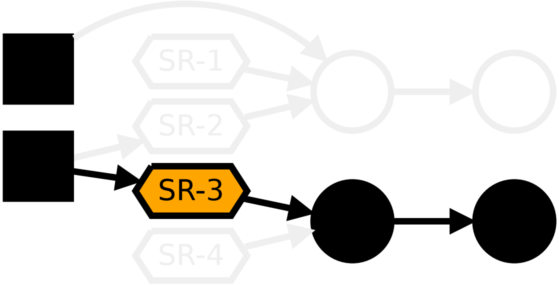
Weak References: Reachability Tricks



SR-3 is reachable, but referent is not.
We may clear the referent, and abandon the subgraph

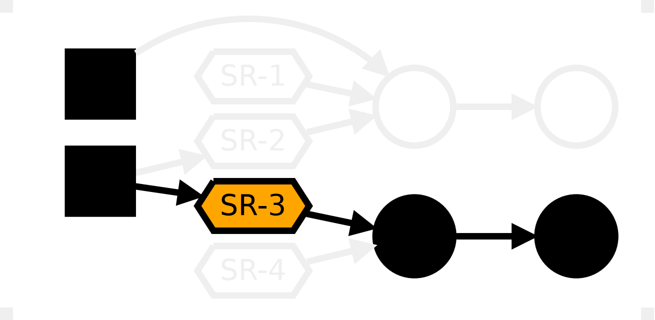
redhat.

Weak References: Reachability Tricks



Trick «Soft»: SR-3 is reachable, but referent is not. We decide to keep referent alive. Can mark through in concurrent mark!

Weak References: Reachability Tricks



SR-3 is reachable, but referent is not. We figure it at pause, decide to keep referent alive: **marking at pause!**

Weak References: Recap, Phases

- Unreachable references: excellent

Reference	Referent	Discovery (concurrent)	Process (STW)	Enqueue (STW)
Dead	Alive	no	no	no
Dead	Dead	no	no	no

redhat.

Weak References: Recap, Phases

- Unreachable references: excellent
- Reachable referents: good, little overhead

Reference	Referent	Discovery (concurrent)	Process (STW)	Enqueue (STW)
Dead	Alive	no	no	no
Dead	Dead	no	no	no
Alive	Alive	yes	maybe	no

redhat

Weak References: Recap, Phases

- Unreachable references: excellent
- Reachable referents: good, little overhead
- Unreachable referents: bad, lots of work during STW

Reference	Referent	Discovery (concurrent)	Process (STW)	Enqueue (STW)
Dead	Alive	no	no	no
Dead	Dead	no	no	no
Alive	Alive	yes	maybe	no
Alive	Dead	yes	YES	YES

redhat.

Weak References: Recap, Keep Alive

When referent is unreachable, should we make it alive again?

Type	Keep Alive		Comment
	JDK 8-	JDK 9+	
Soft	no	no	Cleared on enqueue
Weak	no	no	Cleared on enqueue

redhat.

Weak References: Recap, Keep Alive

When referent is unreachable, should we make it alive again?

- Finalizable objects require resurrection!

Type	Keep Alive JDK 8-	Keep Alive JDK 9+	Comment
Soft	no	no	Cleared on enqueue
Weak	no	no	Cleared on enqueue
Final	YES	YES	← OMG HE COMES CENTER CANNOT HOLD

Weak References: Recap, Keep Alive

When referent is unreachable, should we make it alive again?

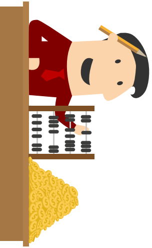
- Finalizable objects require resurrection!
- Phantom references may have to walk the object graph!

Type	Keep Alive		Comment
	JDK 8-	JDK 9+	
Soft	no	no	Cleared on enqueue
Weak	no	no	Cleared on enqueue
Final	YES	YES	← OMG HE COMES CENTER CANNOT HOLD
Phantom	yes	no	Cleared on enqueue since JDK 9

redhat.

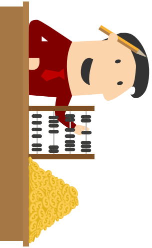
Weak References: Churn Test

```java
@Benchmark
public void churn(Blackhole bh) {
  bh.consume(new Finalizable());
  bh.consume(new byte[10000]);
}
```



```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Final Mark (G) = 14.90 s (a = 338708 us)
Pause Final Mark (N) = 14.90 s (a = 338596 us)
  Finish Queues      =  8.36 s (a = 189976 us)
  Weak References    =  6.50 s (a = 147657 us)
    Process          =  6.04 s (a = 137335 us)
    Enqueue          =  0.45 s (a =  10312 us)
```

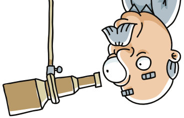
Weak References: Retain Test



```
@Benchmark
public Object test() {
  if (rq.poll() != null) {
    ref = new PhantomReference<>(createTreeMap(), rq);
  }
  return new byte[1000];
}
```

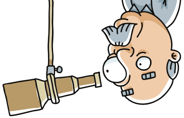
```
# jdk8/bin/java -XX:+UseShenandoahGC -verbose:gc
Pause Final Mark (G) =  0.44 s (a = 12133 us)
Pause Final Mark (N) =  0.39 s (a = 10777 us)
  Finish Queues      =  0.08 s (a =  2123 us)
  Weak References    =  0.29 s (a = 41841 us)
    Process          =  0.29 s (a = 41757 us)
    Enqueue          =  0.00 s (a =    78 us)
```

Weak References: Latency Tips



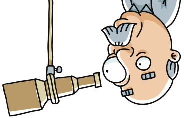
1. Avoid reference churn!
   - Make sure referents normally stay reachable
   - Do more explicit lifecycle mgmt if they get unreachable often
   - Avoid finalizable objects! Use `java.lang.ref.Cleaner`!

Weak References: Latency Tips



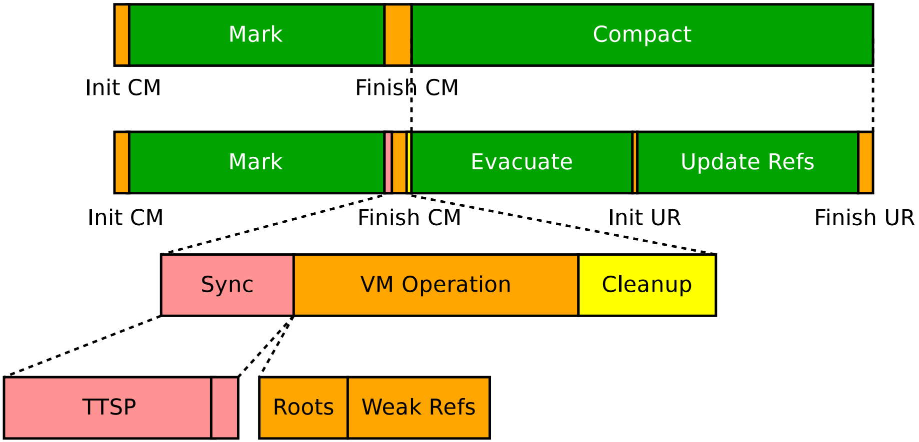
1. Avoid reference churn!
   - Make sure referents normally stay reachable
   - Do more explicit lifecycle mgmt if they get unreachable often
   - Avoid finalizable objects! Use `java.lang.ref.Cleaner`!
2. Keep graphs reachable via special references small
   - Depending on JDK, phantom references need care: use `clear()`
   - Or, make sure references die along with referents

Weak References: Latency Tips

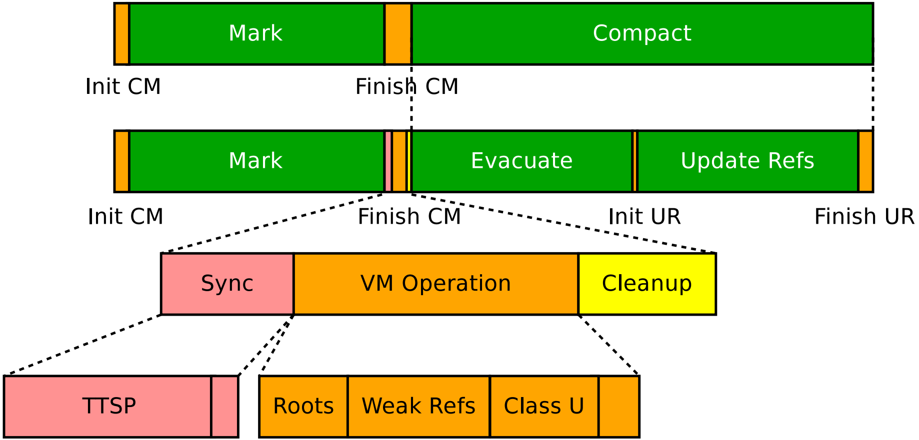


1. Avoid reference churn!
   - Make sure referents normally stay reachable
   - Do more explicit lifecycle mgmt if they get unreachable often
   - Avoid finalizable objects! Use `java.lang.ref.Cleaner`!
2. Keep graphs reachable via special references small
   - Depending on JDK, phantom references need care: use `clear()`
   - Or, make sure references die along with referents
3. Tune down the weakref processing frequency
   - Look for GC-specific setup
     (Shenandoah example: `-XX:ShenandoahRefProcFrequency=#`)

Class Unload: Pause Taxonomy



Class Unload: Pause Taxonomy



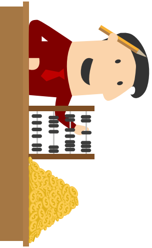
Class Unload: Why, When, How

«A class or interface may be unloaded if and only if its defining class loader may be reclaimed by the GC»[5]

- Matters the most when classloaders come and go: enterprisey apps and other twisted magic
- Class unloading is enabled by default in Hotspot (`-XX:+ClassUnloading`)
- Before JDK 12, all implementations required STW

[5] https://docs.oracle.com/javase/specs/jls/se9/html/jls-12.html#jls-12.7

Class Unload: Test

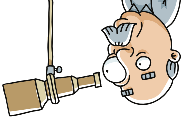


```java
@Benchmark
public Class<?> load() throws Exception {
  return Class.forName("java.util.HashMap",
             true, new URLClassLoader(new URL[0]));
}
```

```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Final Mark (G) = 0.66 s (a = 328942 us)
Pause Final Mark (N) = 0.66 s (a = 328860 us)
  System Purge       = 0.66 s (a = 328668 us)
    Unload Classes   = 0.09 s (a =  43444 us)
    CLDG             = 0.57 s (a = 284217 us)
```

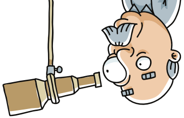

redhat

Class Unload: Latency Tips



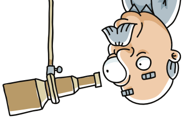
1. Do not expect class unload? → Disable the feature
   - `-XX:-ClassUnloading` is the ultimate killswitch
   - ...but may have ill performance effects when classes to go away

Class Unload: Latency Tips



1. Do not expect class unload? → Disable the feature
   - `-XX:-ClassUnloading` is the ultimate killswitch
   - ...but may have ill performance effects when classes to go away
2. Expect rare class unload? → Tune down the frequency
   - Look for GC-specific class unloading frequency setup (Shenandoah example: `-XX:ShenandoahUnloadClassesFreq=#`)

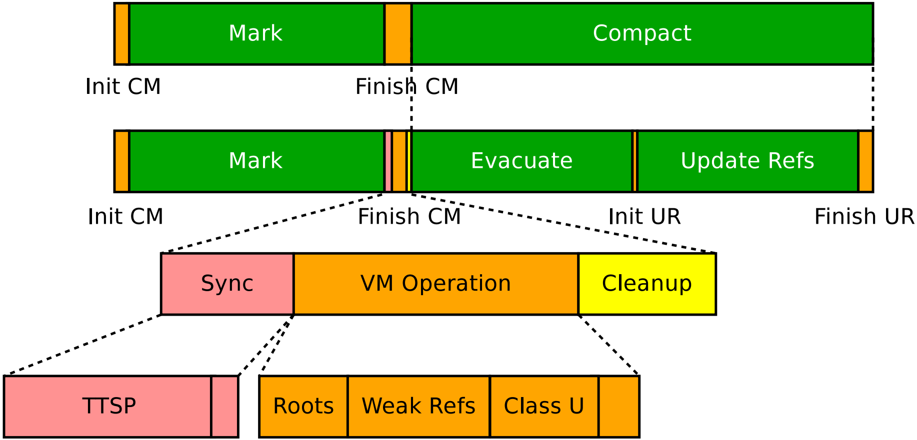
Class Unload: Latency Tips



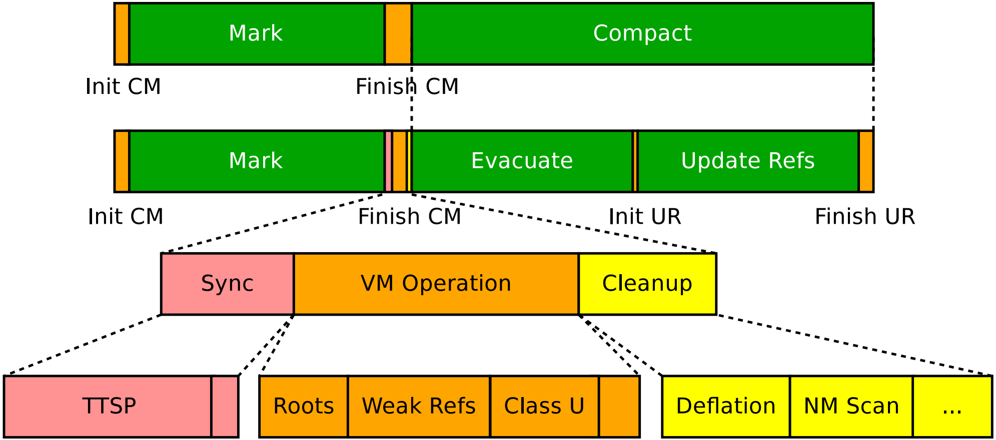
1. Do not expect class unload? $\rightarrow$ Disable the feature
   - `-XX:-ClassUnloading` is the ultimate killswitch
   - ...but may have ill performance effects when classes to go away
2. Expect rare class unload? $\rightarrow$ Tune down the frequency
   - Look for GC-specific class unloading frequency setup (Shenandoah example: `-XX:ShenandoahUnloadClassesFreq=#`)
3. Exploit runtime improvements
   - JDK 12+: concurrent class unloading
   - Shenandoah: parallel class metadata scans

redhat.

Safepoint Epilog: Pause Taxonomy



Safepoint Epilog: Pause Taxonomy



Safepoint Epilog: What, When, Why

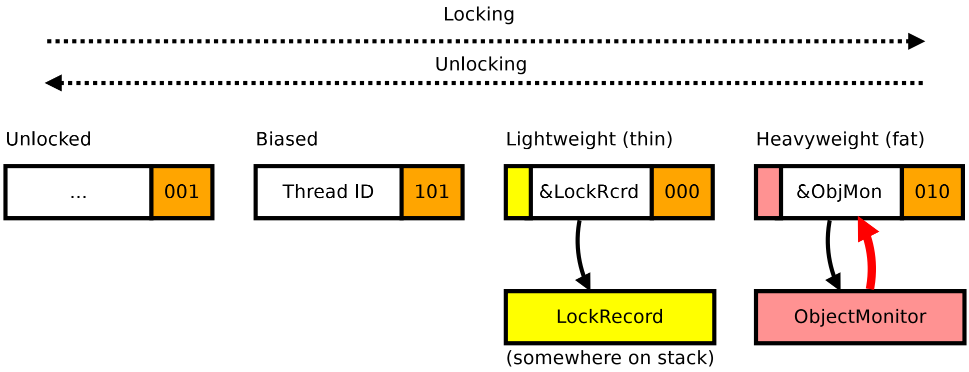
There are actions that execute at each safepoint
(because why not, if we are at STWs)

```
# jdk8/bin/java -XX:+TraceSafepointCleanupTime
[deflating idle monitors, 0.0013491 secs]
[updating inline caches, 0.0000395 secs]
[compilation policy safepoint handler, 0.0000004 secs]
[mark nmethods, 0.0005378 secs]
[rotating gc logs, 0.0002754 secs]^2
[purging class loader data graph, 0.0000002 secs]
```

[5] Specific for 8, fixed in 9; logging added to 8 recently with JDK-8231398

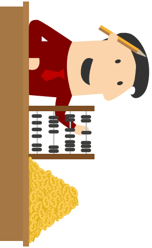
redhat

Monitor Deflation: Why



Missed me? Missed me? Missed me? Missed me?
Somebody needs to «deflate» the monitors...

Monitor Deflation: Deflation Test

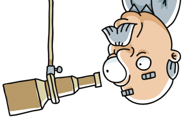


```
static class SyncPair {
  int x, y;
  public synchronized void move() {
    x++; y--;
  }
}
```

```
# java -XX:+TraceSafepointCleanup -Dcount=1'000'000
[deflating idle monitors, 0.0877930 secs]
...

Pause Init Mark (G) = 0.09 s (a = 92052 us)
Pause Init Mark (N) = 0.00 s (a =  3982 us)
```


redhat

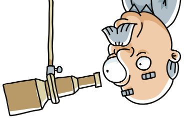
Monitor Deflation: Latency Tips[6]



1. Avoid heavily contended `synchronized` locks
   - `j.u.c.l.Lock`: footprint overheads
   - Atomic operations: performance and complexity overhead

[6]All these are for extreme cases, and need verification that nothing else gets affected

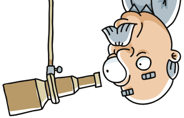
Monitor Deflation: Latency Tips[6]



1. Avoid heavily contended `synchronized` locks
   - `j.u.c.l.Lock`: footprint overheads
   - Atomic operations: performance and complexity overhead
2. Have more safepoints!
   - Keeps monitor population low by eagerly cleaning them up
   - `-XX:GuaranteedSafepointInterval=#` is your friend here

[6] All these are for extreme cases, and need verification that nothing else gets affected

Monitor Deflation: Latency Tips[6]



1. Avoid heavily contended `synchronized` locks
   - `j.u.c.l.Lock`: footprint overheads
   - Atomic operations: performance and complexity overhead
2. Have more safepoints!
   - Keeps monitor population low by eagerly cleaning them up
   - `-XX:GuaranteedSafepointInterval=#` is your friend here
3. Exploit runtime improvements
   - `-XX:+MonitorInUseLists`, enabled by default since JDK 9
   - `-XX:MonitorUsedDeflationThreshold=#`, incremental deflation
   - In progress: concurrent monitor deflation

---

[6] All these are for extreme cases, and need verification that nothing else gets affected

NMethod Scanning: Why

JIT compilers generate lots of code,
some of that code is unused after a while:

```
9680    2  o.a.c.c.StandardContext::unbind
10437   3  o.a.c.c.StandardContext::unbind
9680    2  o.a.c.c.StandardContext::unbind  made not entrant
11385   4  o.a.c.c.StandardContext::unbind
10437   3  o.a.c.c.StandardContext::unbind  made not entrant
9680    2  o.a.c.c.StandardContext::unbind  made zombie
10437   3  o.a.c.c.StandardContext::unbind  made zombie
11385   4  o.a.c.c.StandardContext::unbind  made not entrant
```

Need to clean up stale versions of the code

NMethod Scanning: Caveat

To sweep the generated method,
we need to make sure nothing uses it

1. Decide the method needs sweep
2. Mark method «not entrant»: forbid new activations
3. Check no activations are present on stacks
4. Mark the nmethod «zombie»: ready for sweep
5. Sweep the method

NMethod Scanning: Caveat

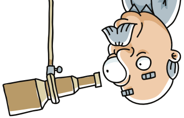
To sweep the generated method,
we need to make sure nothing uses it

1. Decide the method needs sweep
2. Mark method «not entrant»: forbid new activations
3. Check no activations are present on stacks
4. Mark the nmethod «zombie»: ready for sweep
5. Sweep the method

```
# jdk8/bin/java -XX:+TraceSafepointCleanupTime
[mark nmethods, 0.0005378 secs]
```


redhat

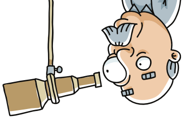
NMethod Scanning: Latency Tips[7]



1. Turn off method flushing
   - `-XX:-MethodFlushing` is your friend here
   - There are potential ill effects: code cache overfill (compilation stops), code cache locality problems (performance problems)

[7] All these are for extreme cases, and need verification that nothing else gets affected

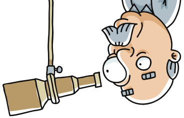
NMethod Scanning: Latency Tips[7]



1. Turn off method flushing
   - `-XX:-MethodFlushing` is your friend here
   - There are potential ill effects: code cache overfill (compilation stops), code cache locality problems (performance problems)
2. Reconsider the control flow to avoid deep stacks
   - Less stack frames to scan, gets easier on sweeper

[7] All these are for extreme cases, and need verification that nothing else gets affected

NMethod Scanning: Latency Tips[7]



1. Turn off method flushing
   - `-XX:-MethodFlushing` is your friend here
   - There are potential ill effects: code cache overfill (compilation stops), code cache locality problems (performance problems)
2. Reconsider the control flow to avoid deep stacks
   - Less stack frames to scan, gets easier on sweeper
3. Exploit runtime improvements
   - JDK 10+ provides piggybacking nmethod scans on GC safepoints
   - Shenandoah: enables nmethod scans piggybacking

[7] All these are for extreme cases, and need verification that nothing else gets affected

Code Roots: Why

```
static final MyIntHolder constant = new MyIntHolder();

@Benchmark
public int test() {
  return constant.x;
}
```

Inlining reference constants into generated code is natural for throughput performance:

```
movabs $0x7111b5108,%r10   # Constant oop
mov    0xc(%r10),%edx      # getfield x
...
callq 0x00007f73735dff80   # Blackhole.consume(int)
```

Code Roots: Fixups

```
movabs $0x7111b5108,%r10   # Constant oop
mov    0xc(%r10),%edx      # getfield x
...
callq 0x00007f73735dff80   # Blackhole.consume(int)
```

- Inlined references require code patching: only safe to do when nothing executes the code block ⇒ pragmatically, under STW

redhat.

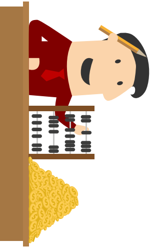
Code Roots: Fixups

```
movabs $0x7111b5108,%r10   # Constant oop
mov    0xc(%r10),%edx      # getfield x
...
callq 0x00007f73735dff80   # Blackhole.consume(int)
```

- Inlined references require code patching: only safe to do when nothing executes the code block ⇒ pragmatically, under STW
- Also need to *pre-evacuate* the code roots before anyone sees old object reference!

Code Roots: Pre-Evacuation

Need to pre-evacuate code roots before unparking from STW:

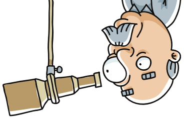


```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Final Mark (G)     = 0.13 s (a =  2768 us)
Pause Final Mark (N)     = 0.10 s (a =  2623 us)
  Initial Evacuation     = 0.08 s (a =  2515 us)
    E: Code Cache Roots  = 0.04 s (a =  1227 us)
```

Alternative: barriers after constants, with throughput hit

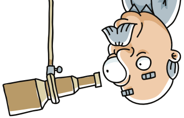
redhat

Code Roots: Latency Tips



1. Have less compiled code around
   - Disable tiered compilation
   - More aggressive code cache sweeping

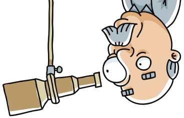
Code Roots: Latency Tips



1. Have less compiled code around
   - Disable tiered compilation
   - More aggressive code cache sweeping
2. Tell runtime to treat code roots for latency
   - `-XX:ScavengeRootsInCode=0` to remove compiler oops
   - GC-specific tuning enabling concurrent code cache evacuation

redhat

Code Roots: Latency Tips



1. Have less compiled code around
   - Disable tiered compilation
   - More aggressive code cache sweeping
2. Tell runtime to treat code roots for latency
   - `-XX:ScavengeRootsInCode=0` to remove compiler oops
   - GC-specific tuning enabling concurrent code cache evacuation
3. Exploit runtime improvements
   - Shenandoah (JDK 8+), G1 (JDK 9+): code cache roots tracking

Conclusion (II)

Conclusion (II): Ultra-Low Latency Addendum

Pre-requisite: get a decent concurrent GC.

Conclusion (II): Ultra-Low Latency Addendum

Pre-requisite: get a decent concurrent GC. After that:

1. OpenJDK is able to provide ultra-low (< 1 ms) pauses in non-extreme cases, and low pauses (< 100 ms) in extreme cases

Conclusion (II): Ultra-Low Latency Addendum

Pre-requisite: get a decent concurrent GC. After that:

1. OpenJDK is able to provide ultra-low (< 1 ms) pauses in non-extreme cases, and low pauses (< 100 ms) in extreme cases
2. OpenJDK is able to provide ultra-low pauses in extreme cases with some runtime improvements. Some of them are already available, **upgrade!**

redhat.

Conclusion (II): Ultra-Low Latency Addendum

Pre-requisite: get a decent concurrent GC. After that:

1. OpenJDK is able to provide ultra-low (< 1 ms) pauses in non-extreme cases, and low pauses (< 100 ms) in extreme cases

2. OpenJDK is able to provide ultra-low pauses in extreme cases with some runtime improvements. Some of them are already available, **upgrade!**

3. One can avoid extreme case pitfalls with careful and/or specialized code, until runtimes catch up

Conclusion (II): Releases

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

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

- Dev follows latest JDK, backports to 13, **11u, 8u**
- 8u backport ships in RHEL 7.4+, Fedora 24+
- 11u backport ships in Fedora 27+
- Nightly development builds (tarballs, Docker images)

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