Shenandoah GC

Part II: I See You Have Your Fancy GC

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.

Usual Disclaimers

This talk...

1. ...does not explain the GC basics, but rather covers the **runtime** parts needed for collector to work. See «Part I» for basics!

2. ...covers the runtime interface itself, and sometimes discusses GC and runtime tricks to mitigate problems. Shenandoah, ZGC, and other collectors need them!

3. ...is specific to **current** state of OpenJDK and Hotspot. Future work may render many of these issues fixed!

Overall

Overall: When Everything Is Perfect

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
```

Overall: When Everything Is Perfect

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
```

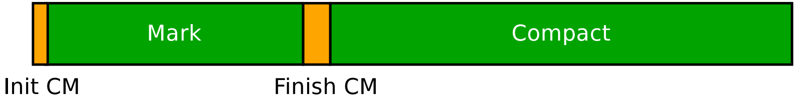

redhat

Overall: When Something Is Not So Good

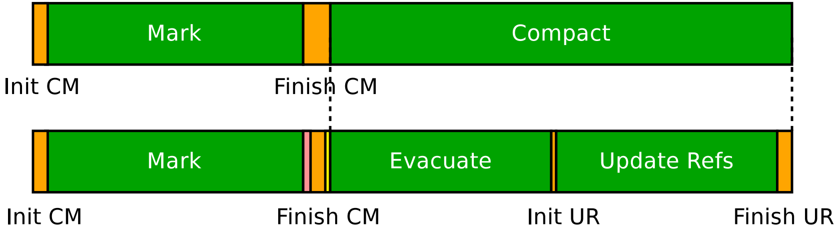
Worst-case cycle in one of the workloads:

```
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
```

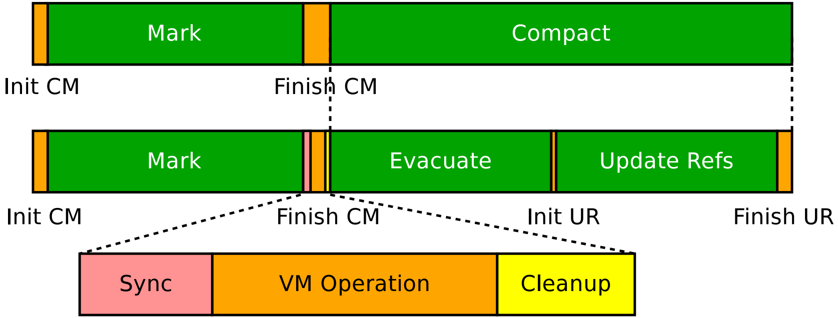
Overall: Pause Taxonomy



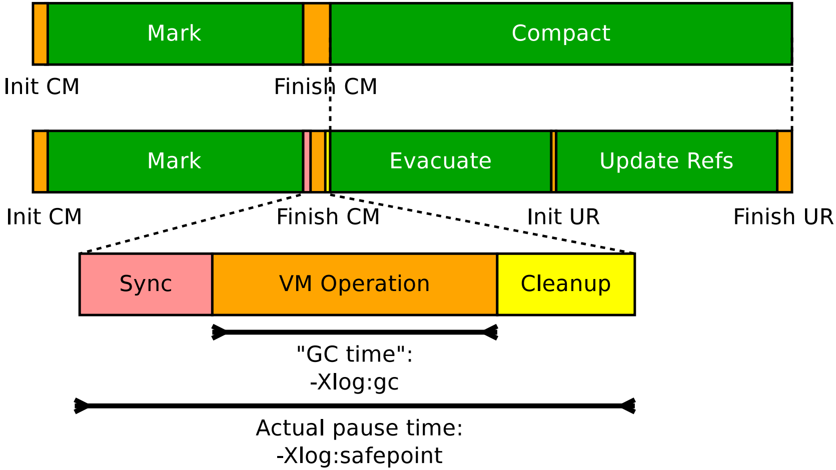
Overall: Pause Taxonomy



Overall: Pause Taxonomy



Overall: Pause Taxonomy



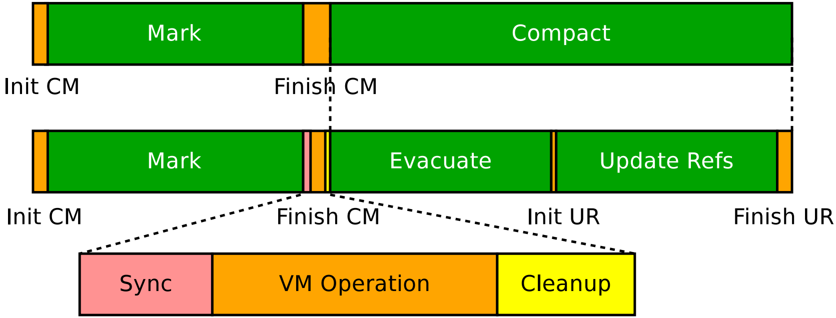
Safepoint Prolog

Safepoint Prolog: Ideas

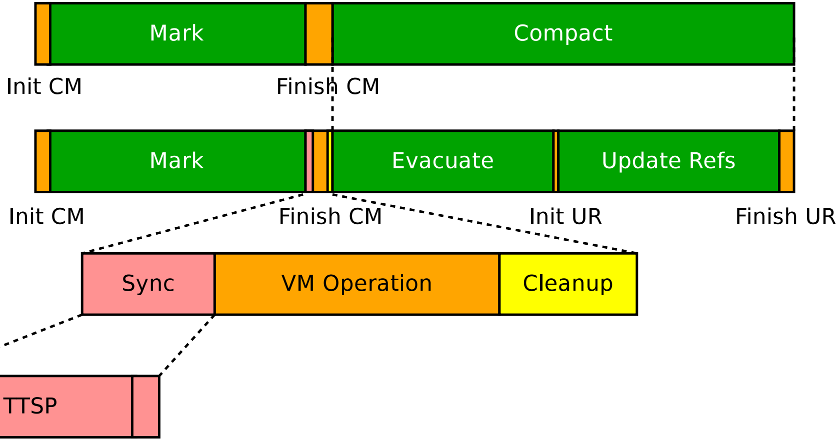
1. Make sure changing the heap is **safe**
2. Enable **cooperative** thread suspension
3. Have the known state points: e.g. where are the **pointers**

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

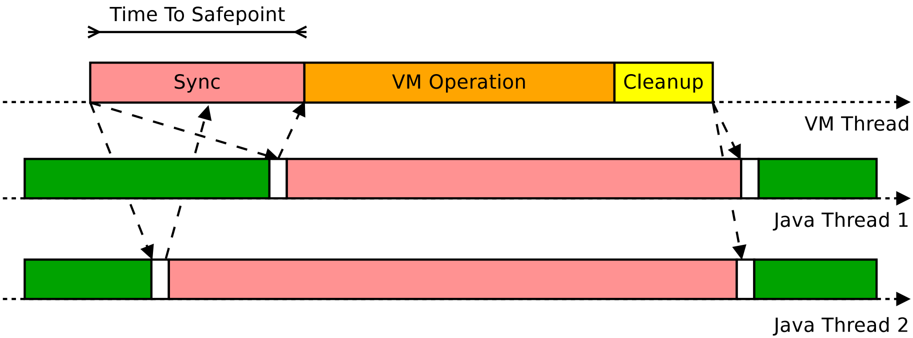
TTSP: Pause Taxonomy



TTSP: Pause Taxonomy

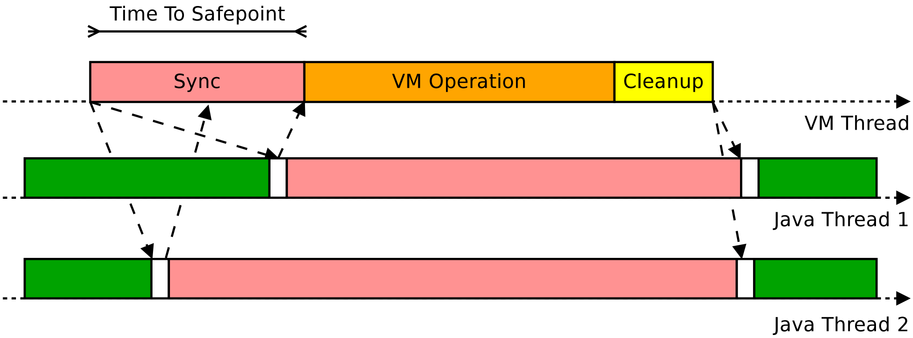


TTSP: Definition



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

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
```

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

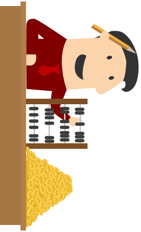
In tight loops, safepoint poll costs are very visible!
Solution: eliminate safepoint polls in short cycles

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

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

redhat

TTSP: Long Loops



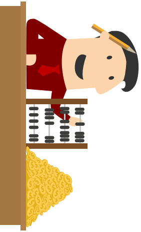
```
int[] arr;

@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)
```


redhat

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)
```


redhat

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!

TTSP: Loop Strip Mining

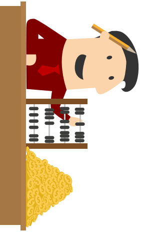
```
# -XX:+UseShenandoahGC -XX:-UseCLS
Performance: 35.832 +- 1.024  ms/op
Total Pauses (G) =  0.69 s (a = 26531 us)
Total Pauses (N) =  0.02 s (a =   734 us)
```


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)
```

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;
}
```

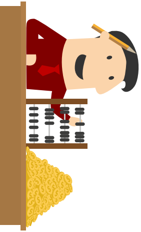
Each thread needs scheduling to roll to safepoint:

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


redhat

TTSP: Runnable Threads Test

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

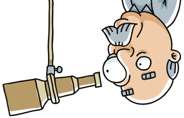


Each thread needs scheduling to roll to safepoint:

```
# 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)
```

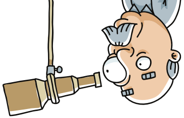

redhat

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

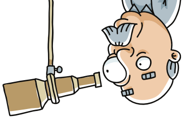
TTSP: Latency Tips



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

redhat.

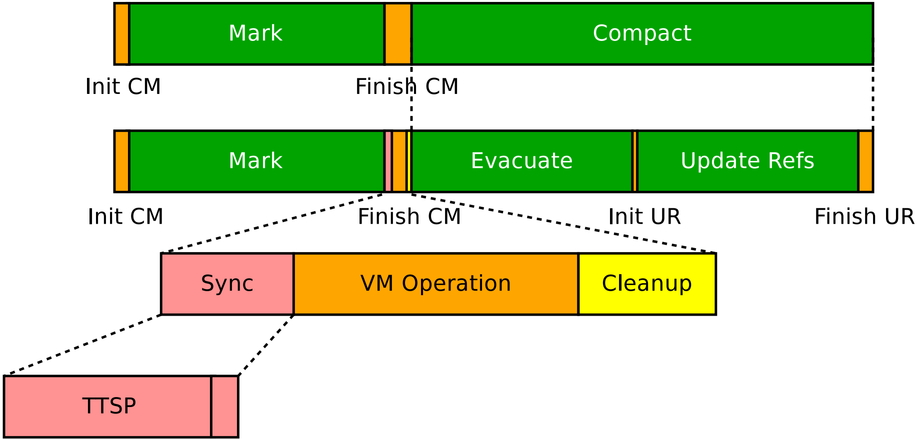
TTSP: Latency Tips



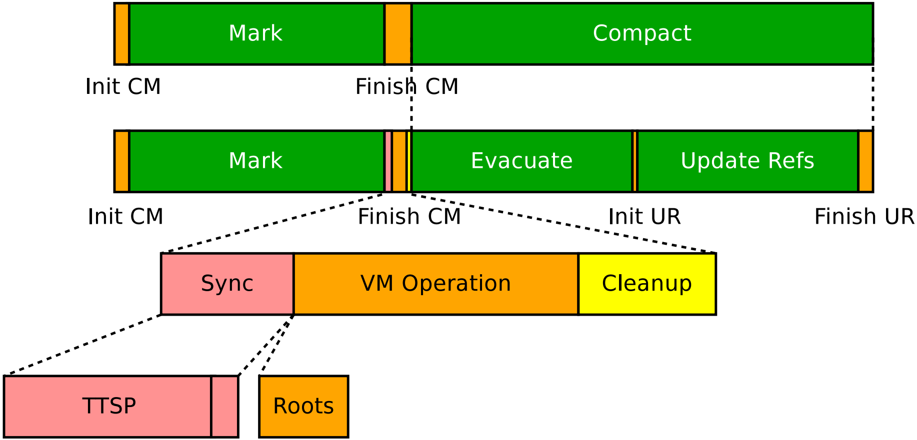
1. Safepoint monitoring is your friend
   - Enable `-XX:+PrintSafepointStatistics` along with GC logs
   - Use GC that tells you gross pause times that include safepoints
2. Trim down the number of runnable threads
   - Overwhelming the system is never good
   - Use shared thread pools, and then share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

GC Roots: Pause Taxonomy



GC Roots: Pause Taxonomy



GC Roots: What Are They, Dude

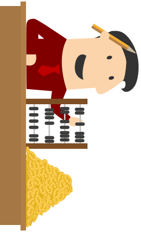
Def: «GC Root», slot with implicitly reachable object

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)
```


redhat

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

redhat

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

```
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

[1] https://shipilev.net/jvm-anatomy-park/8-local-var-reachability/

redhat

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

redhat.

Thread Roots: GC Handling

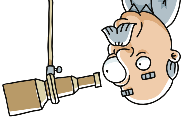
GC threads scan Java threads in parallel:
N GC threads scan K Java threads

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

Corollaries:

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

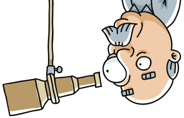
Thread Roots: Latency Tips



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

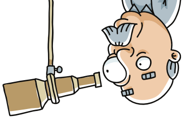
redhat.

Thread Roots: Latency Tips



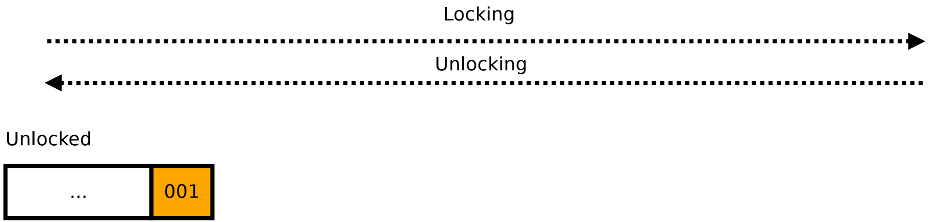
1. Make sure only a few threads are active
   - Ideally, N_CPU threads, sharing the app load
   - Natural with thread-pools: most threads are parked at shallow stack depths
2. Trim down the thread stack depths
   - Calling into thousands of methods exposes lots of locals
   - Tune up inlining: less frames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

Thread Roots: Latency Tips



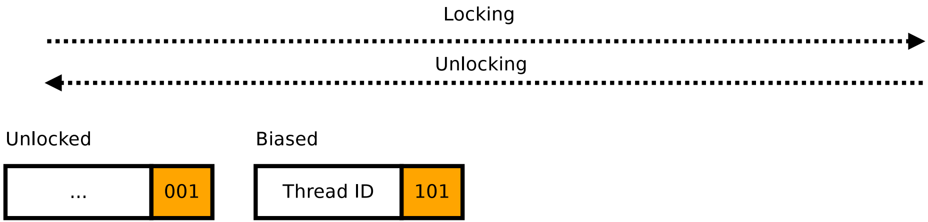
1. Make sure only a few threads are active
   - Ideally, N_CPU threads, sharing the app load
   - Natural with thread-pools: most threads are parked at shallow stack depths
2. Trim down the thread stack depths
   - Calling into thousands of methods exposes lots of locals
   - Tune up inlining: less frames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

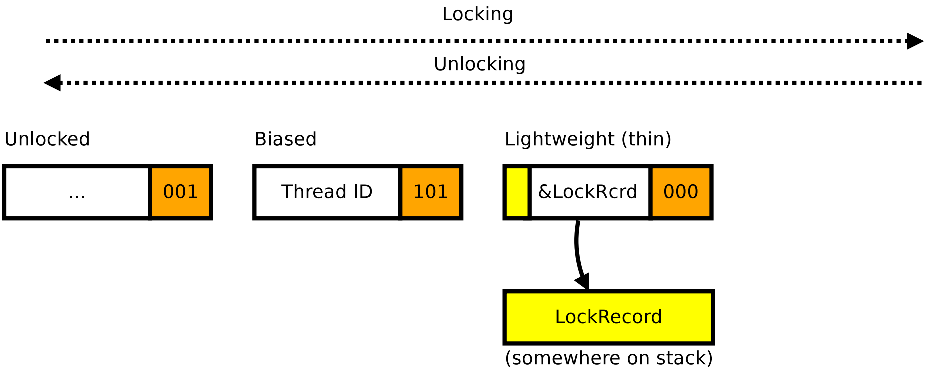
Sync Roots: Why



Progressively heavier lock metadata:
unlocked, biased

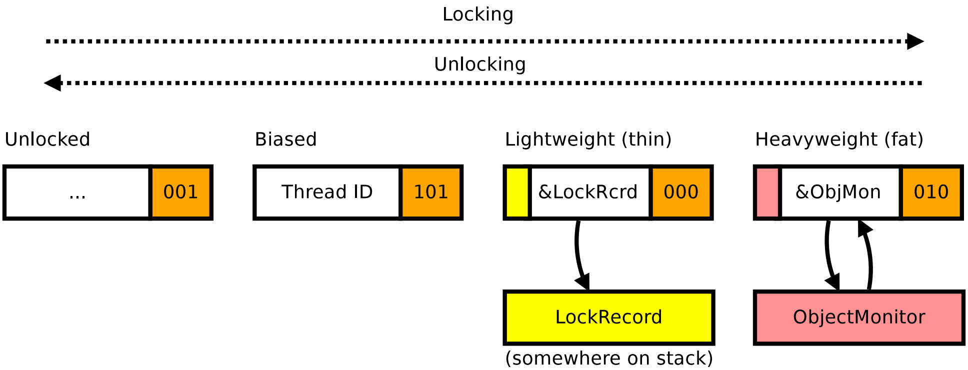
redhat.

Sync Roots: Why



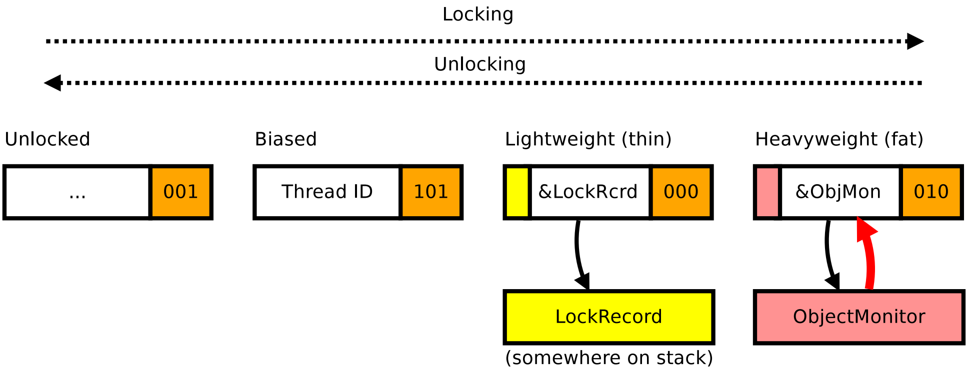
Progressively heavier lock metadata: unlocked, biased, thin locks

Sync Roots: Why



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

Sync Roots: Why



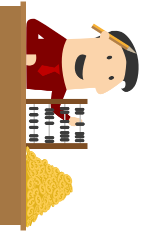
Ultimately, `ObjectMonitor` that associates object with its fat native synchronizer, in both directions

redhat.

Sync Roots: Syncie-Syncie Test

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

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

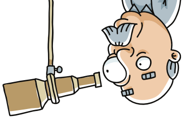
Sync Roots: Depletion Test



```
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)
```

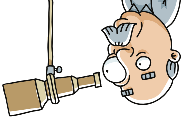
Sync Roots: Latency Tips



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

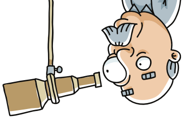
redhat

Sync Roots: Latency Tips



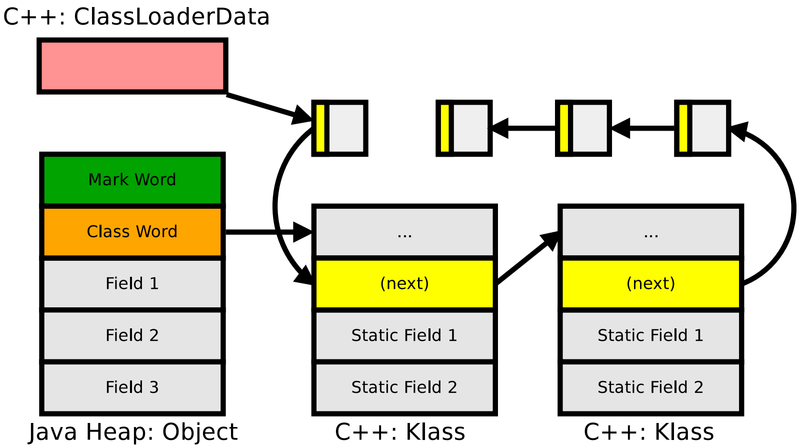
1. Avoid **contended** locking on **lots** of `synchronized`-s
   - Most applications do seldom contention on few monitors
   - Replace with `j.u.c.Lock`, Atomics, `VarHandle`, etc. otherwise
2. Have more frequent safepoints
   - Counter-intuitive, but may keep inflated monitors count at bay
   - (More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

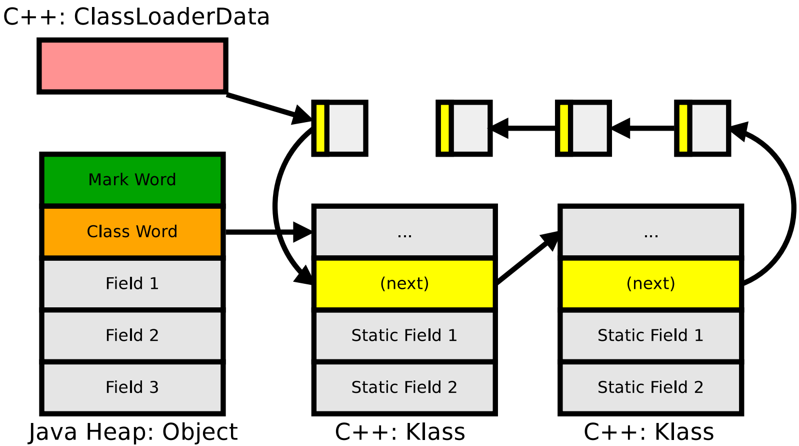
Class Roots: Why



Static fields are stored in class mirrors outside the objects

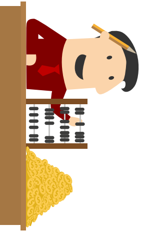
redhat

Class Roots: Why



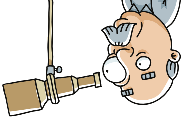
Even without instances, we need to visit static fields

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)
```

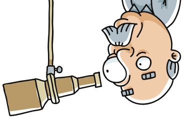


Class Roots: Latency Tips

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

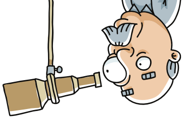
redhat.

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
   - Avoiding oops in native structures (JDK 9+ onwards)
   - Parallel classloader data scans (Shenandoah)
   - Concurrent class scans?

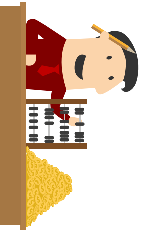
redhat.

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, ...);
  ...
}
```

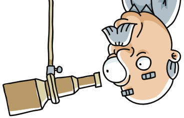
String Table Roots: Intern Test



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

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

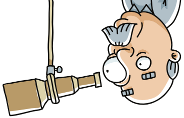
```
# jdk10/bin/java -XX:+UseShenandoahGC -Dsize=1'000'000
Pause Init Mark (G)         =  0.30 s (a = 10698 us)
Pause Init Mark (N)         =  0.29 s (a = 10315 us)
  Scan Roots                =  0.28 s (a = 10046 us)
    S: String Table Roots =  0.25 s (a =  8991 us)
```

String Table Roots: Latency Tips



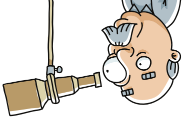
1. Do not use `String.intern()`
   - It is almost never worth it
   - Roll on your own deduplicator/interner

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

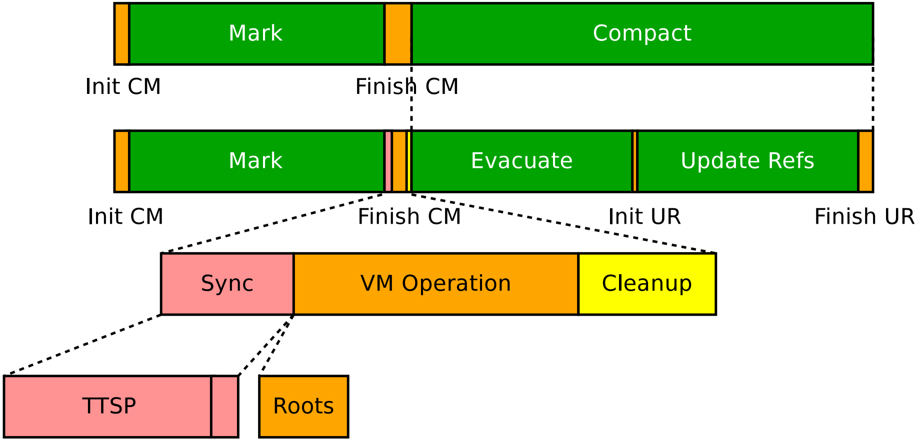
String Table Roots: Latency Tips



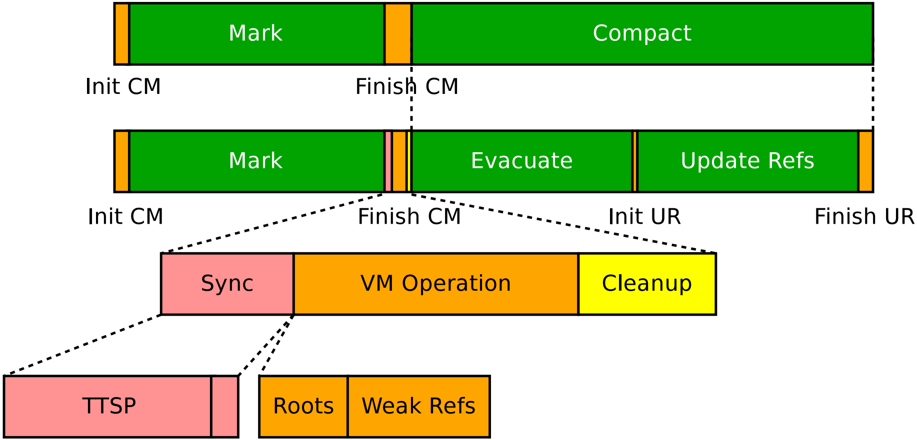
1. Do not use `String.intern()`
   - It is almost never worth it
   - Roll on your own deduplicator/interner
2. Watch out for StringTable rehashing and cleanups
   - `-XX:StringTableSize=#` is your friend here
   - Surprise: `-XX:-ClassUnloading` disables StringTable cleanup
   - Surprise: StringTable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

Weak References

Weak References: Pause Taxonomy



Weak References: Pause Taxonomy



redhat

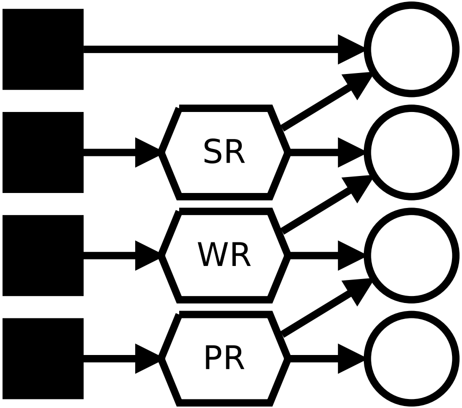
Weak References: What, How, When

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

- Usually named «weak references», in contrast to «strong references»: soft, weak, finalizable, phantom are the subtypes
- Finalizable objects are yet another synthetic weak reachability level: modeled with `j.l.ref.Finalizer`

redhat.

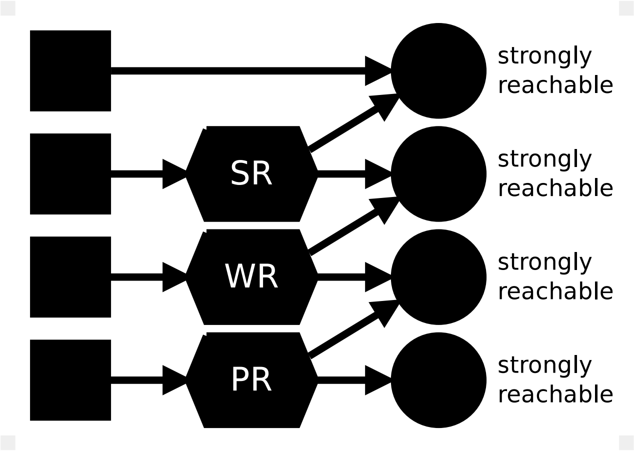
Weak References: How Do They Work?



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

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

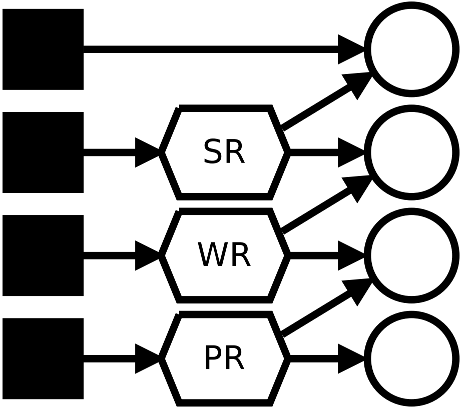
Weak References: How Do They Work?



Scanning **through**[2] the weak references yields strongly reachable heap: normal GC cycle

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

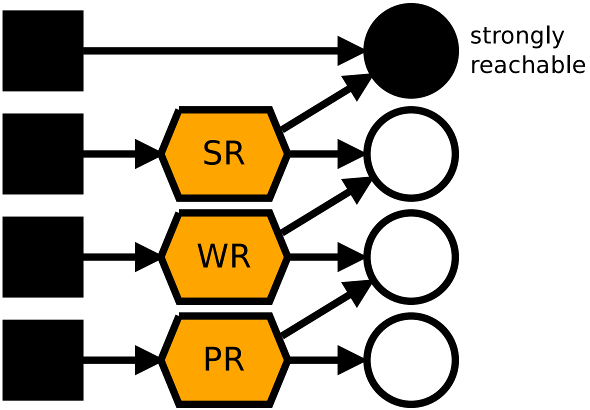
Weak References: How Do They Work?



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

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

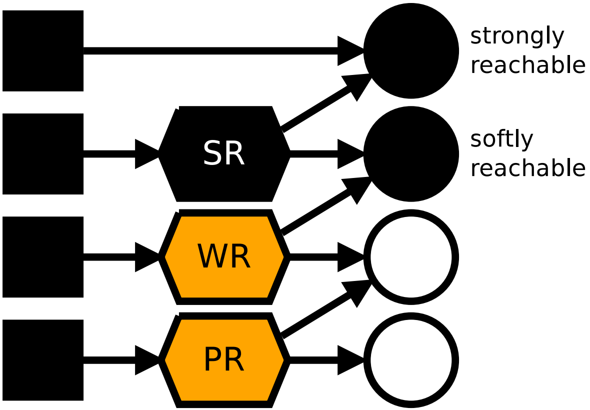
Weak References: How Do They Work?



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

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

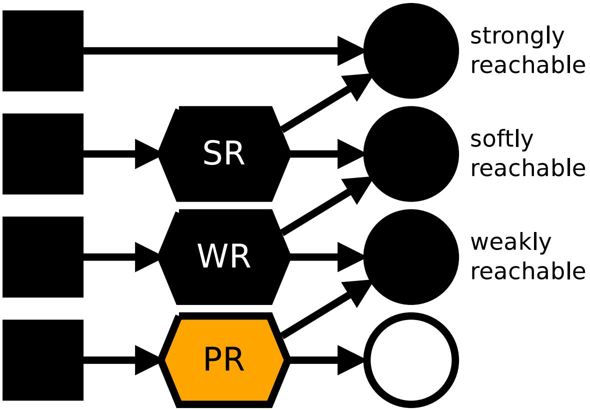
Weak References: How Do They Work?



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

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

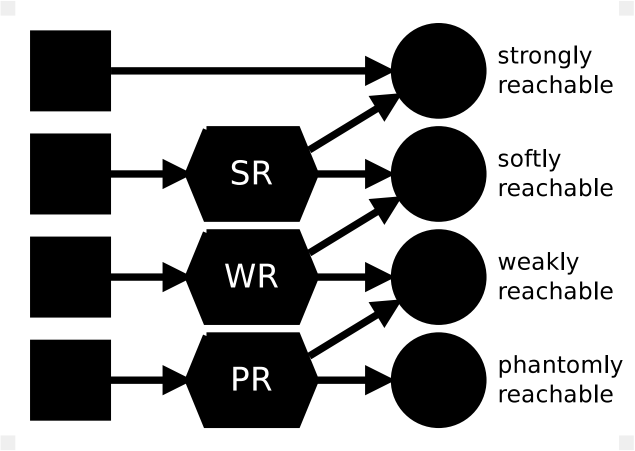
Weak References: How Do They Work?



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

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

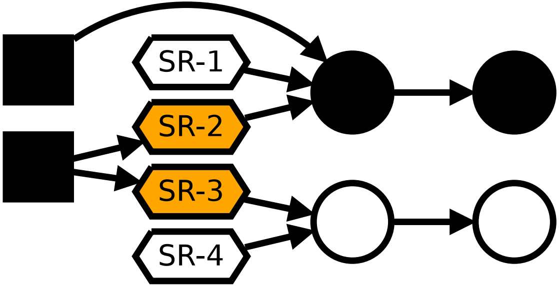
Weak References: How Do They Work?



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

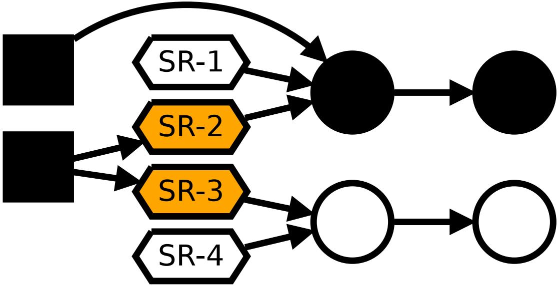
[1] e.g. treating `Reference.referent` as normal field

Weak References: Reachability Tricks



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

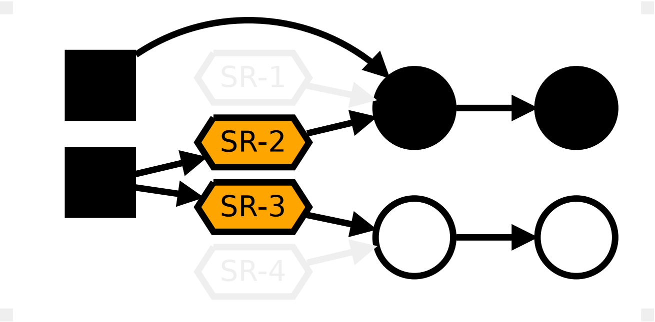
Weak References: Reachability Tricks



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

redhat

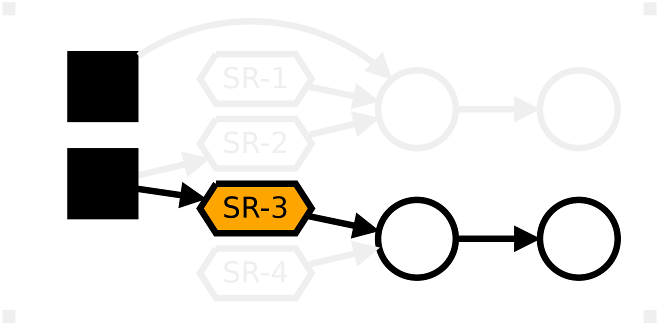
Weak References: Reachability Tricks



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

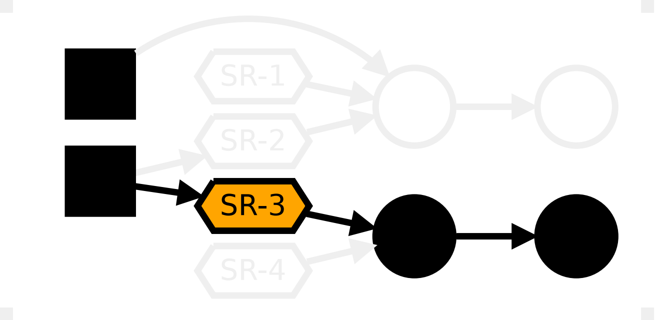
redhat

Weak References: Reachability Tricks



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

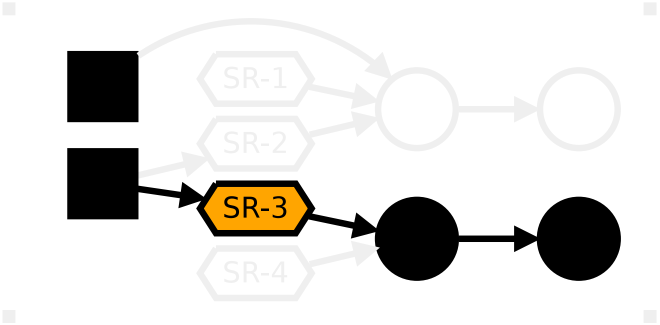
Weak References: Reachability Tricks



Trick «Soft»: SR-3 is reachable, but referent is not. We decide to keep referent alive. This means we have to mark through

redhat.

Weak References: Reachability Tricks



SR-3 is reachable, but referent is not. We decide to keep referent alive. For phantom refs it means **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

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

Weak References: Recap, Keep Alive

When referent is unreachable, should we make it reachable?

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

Weak References: Recap, Keep Alive

When referent is unreachable, should we make it reachable?

- Finalizable objects are required to be walked!

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	← НЕНАВИСТЬ

Weak References: Recap, Keep Alive

When referent is unreachable, should we make it reachable?

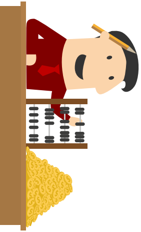
- Finalizable objects are required to be walked!
- Phantom references may have to walk the object graph!

Type	Keep Alive		Comment
	JDK 8-	JDK 9+	
Soft	no	no	Cleared on enqueue
Weak	no	no	Cleared on enqueue
Final	YES	YES	← НЕНАВИСТЬ
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]);
}
```

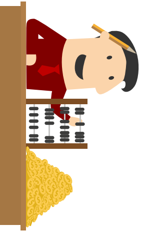


```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Final Mark (G) = 14.90 s (a = 338708 us)
Pause Final Mark (N) = 14.90 s (a = 338596 us)
  Finish Queues      =  8.36 s (a = 189976 us)
  Weak References    =  6.50 s (a = 147657 us)
    Process          =  6.04 s (a = 137335 us)
    Enqueue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


redhat

Weak References: Retain Test

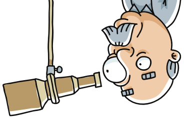
```
@Benchmark
public Object test() {
  if (rq.poll() != null) {
    ref = new PhantomReference<>(createTreeMap(), rq);
  }
  return new byte[1000];
}
```



```
# 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)
```

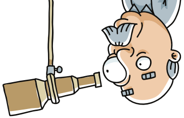

redhat

Weak References: Latency Tips



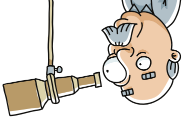
1. Avoid reference churn!
   - Make sure referents normally stay reachable
   - Do more explicit lifecycle mgmt if they get unreachable often
   - Avoid finalizable objects! Use `java.lang.ref.Cleaner`!

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 minimal
   - Depending on JDK, phantom references need care: use `clear()`
   - Or, make sure references die along with referents

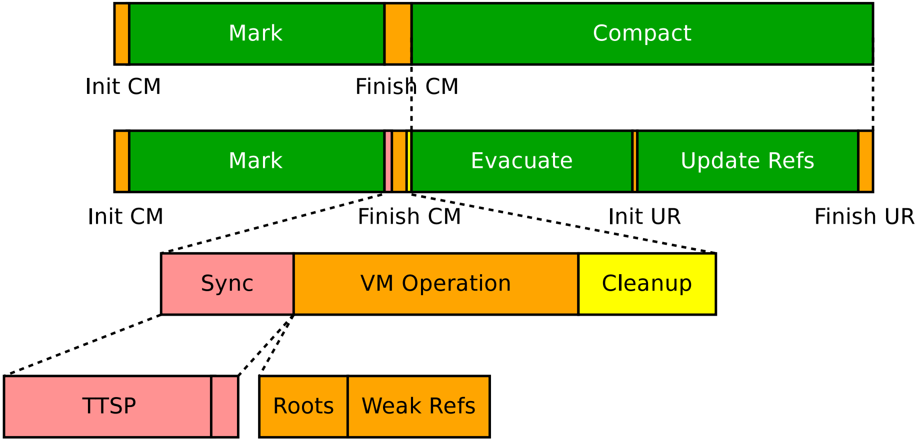
Weak References: Latency Tips



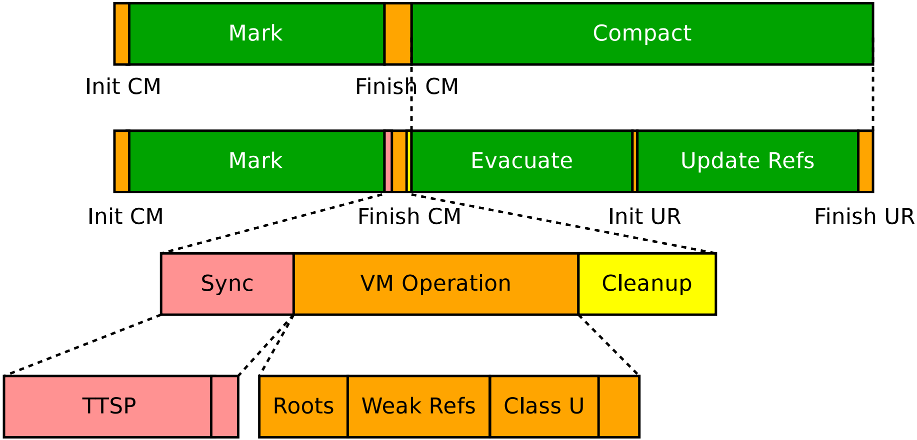
1. Avoid reference churn!
   - Make sure referents normally stay reachable
   - Do more explicit lifecycle mgmt if they get unreachable often
   - Avoid finalizable objects! Use `java.lang.ref.Cleaner`!
2. Keep graphs reachable via special references minimal
   - Depending on JDK, phantom references 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

Class Unload: Pause Taxonomy



Class Unload: Pause Taxonomy



Class Unload: Why, When, How

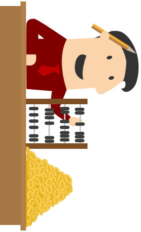
«A class or interface may be unloaded if and only if its defining class loader may be reclaimed by the GC»[2]

- Matters the most when classloaders come and go: enterprisey apps and other twisted magic
- Class unloading is enabled by default in Hotspot (`-XX:+ClassUnloading`)
- Current implementation requires stop-the-world

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

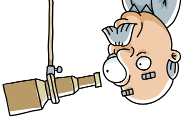
Class Unload: Test

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



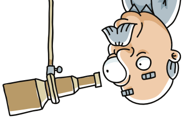
```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Final Mark (G) = 0.66 s (a = 328942 us)
Pause Final Mark (N) = 0.66 s (a = 328860 us)
  System Purge       = 0.66 s (a = 328668 us)
    Unload Classes   = 0.09 s (a =  43444 us)
    CLDG             = 0.57 s (a = 284217 us)
```

Class Unload: Latency Tips



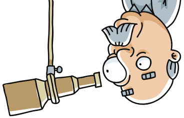
1. Do not expect class unload? $\rightarrow$ Disable the feature
   - `-XX:-ClassUnloading` is the ultimate killswitch
   - ...but may have ill performance effects when classes to go away

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

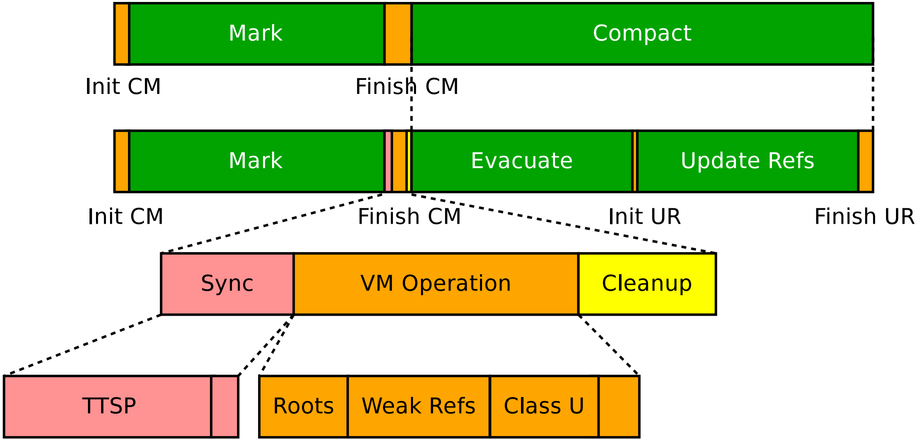


1. Do not expect class unload? → Disable the feature
   - `-XX:-ClassUnloading` is the ultimate killswitch
   - ...but may have ill performance effects when classes to go away
2. Expect rare class unload? → Tune down the frequency
   - Look for GC-specific class unloading frequency setup (Shenandoah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

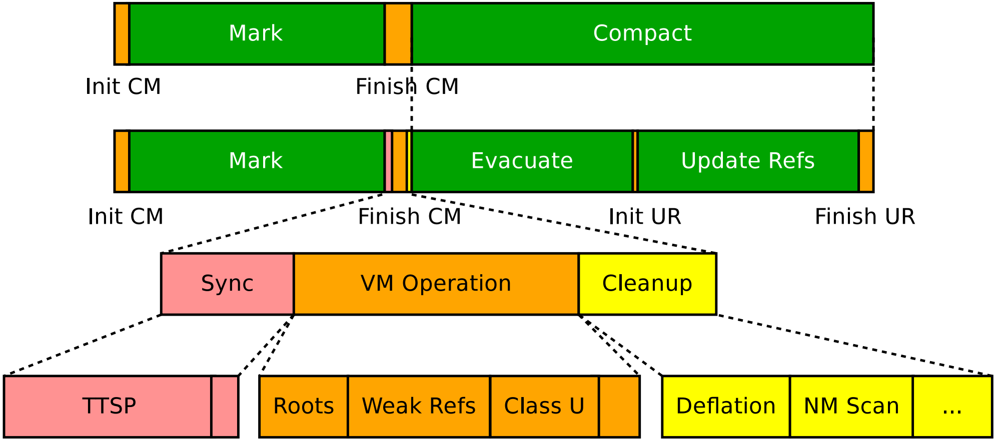
redhat

Safepoint Epilog

Safepoint Epilog: Pause Taxonomy



Safepoint Epilog: Pause Taxonomy



Safepoint Epilog: What, When, Why

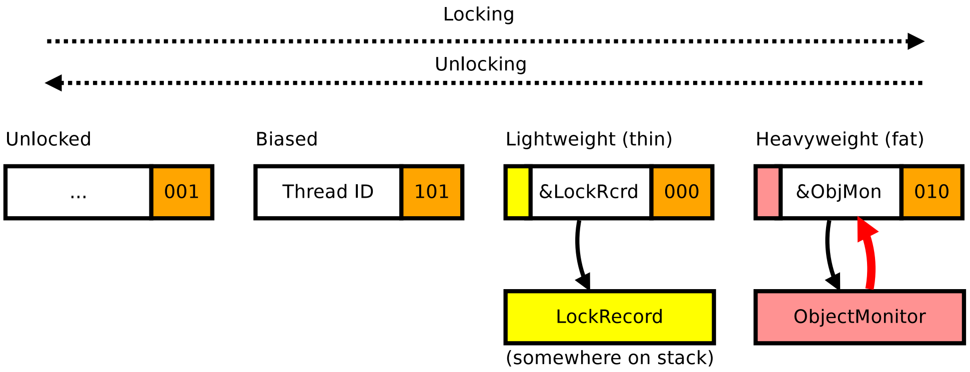
There are actions that execute at each safepoint
(because why not, if we are at STWs)

```
# 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]
[gc log file rotation, 0.0002754 secs][2]
[purging class loader data graph, 0.0000002 secs]
```

[2] Surprisingly, no such logging in default JDK

redhat.

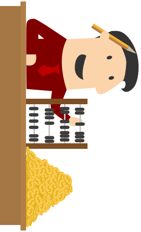
Monitor Deflation: Why



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

redhat

Monitor Deflation: Deflation Test

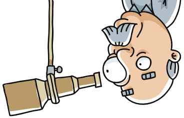


```java
static class SyncPair {
  int x, y;
  public synchronized void move() {
    x++; y--;
  }
}
```

```
# 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.

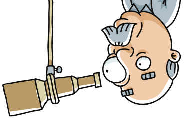
Monitor Deflation: Latency Tips[3]



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

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

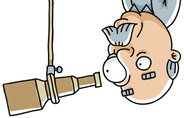
Monitor Deflation: Latency Tips[3]



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

Monitor Deflation: Latency Tips[3]



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

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

redhat.

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

redhat.

NMethod Scanning: Caveat

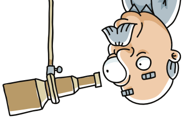
To sweep the generated method,
we need to make sure nothing uses it

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[4]

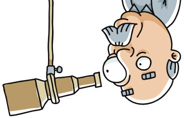


1. Turn off method flushing
   - `-XX:-MethodFlushing` is your friend here
   - There are potential ill effects: code cache overfill (compilation stops), code cache locality problems (performance problems)

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

redhat.

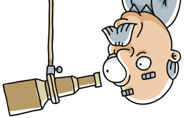
NMethod Scanning: Latency Tips[4]



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

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

NMethod Scanning: Latency Tips[4]



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
   - (Currently only shenandoah/jdk10 supports it)

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

Heap Management

Heap Management: Internals

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

1. Java heap: forwarding pointer (8 bytes/object)
2. Native: 2 marking bitmaps (1/64 bits per heap bit)
3. Native: $N_CPU workers (≈ 2 MB / GC thread)
4. Native: region data (≈ 1 KB per region)

redhat.

Heap Management: Internals

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

1. Java heap: forwarding pointer (8 bytes/object)
2. Native: 2 marking bitmaps (1/64 bits per heap bit)
3. Native: $N_CPU workers (≈ 2 MB / GC thread)
4. Native: region data (≈ 1 KB per region)

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

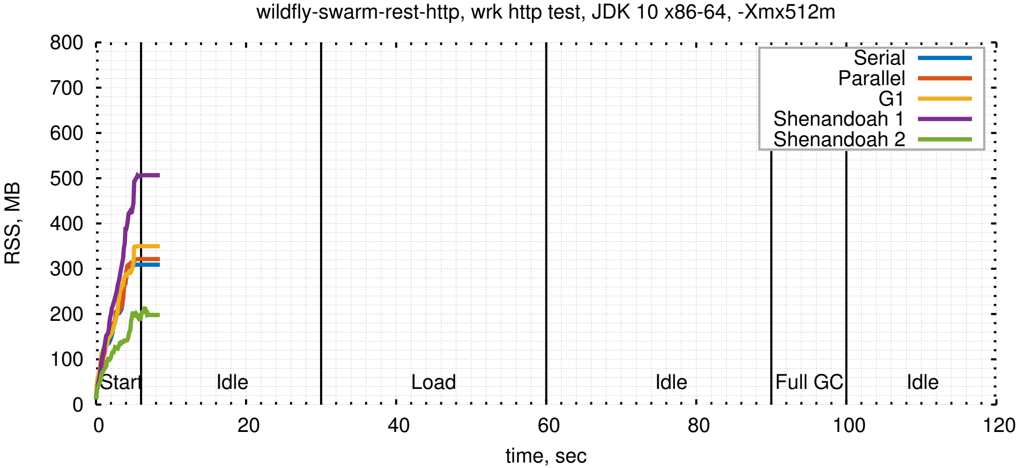
Heap Management: Internals

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

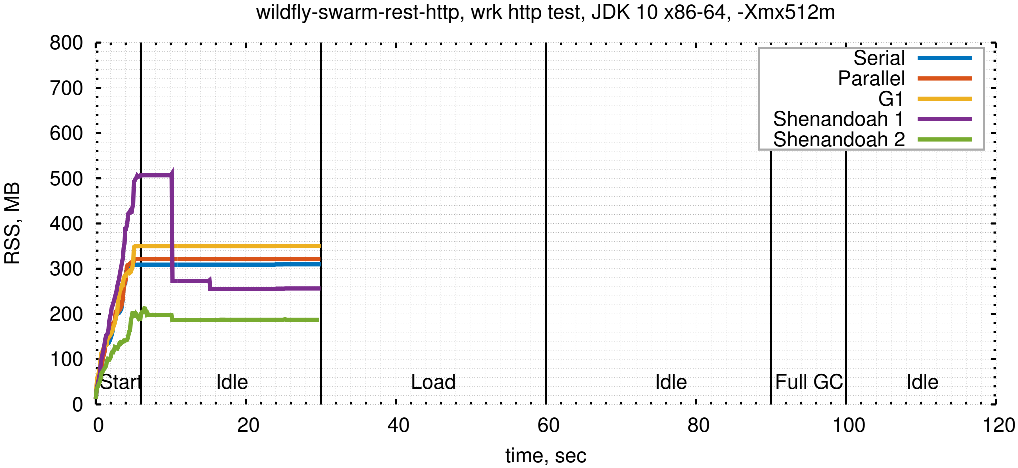
But all of that is totally dwarfed by GC heap sizing policies

Example: `-XX:+UseShenandoahGC -Xmx100G` means:
$\approx$ 90..95 GB accessible for Java objects,
$\approx$ 103 GB RSS for GC parts

Heap Management: Microservice Example

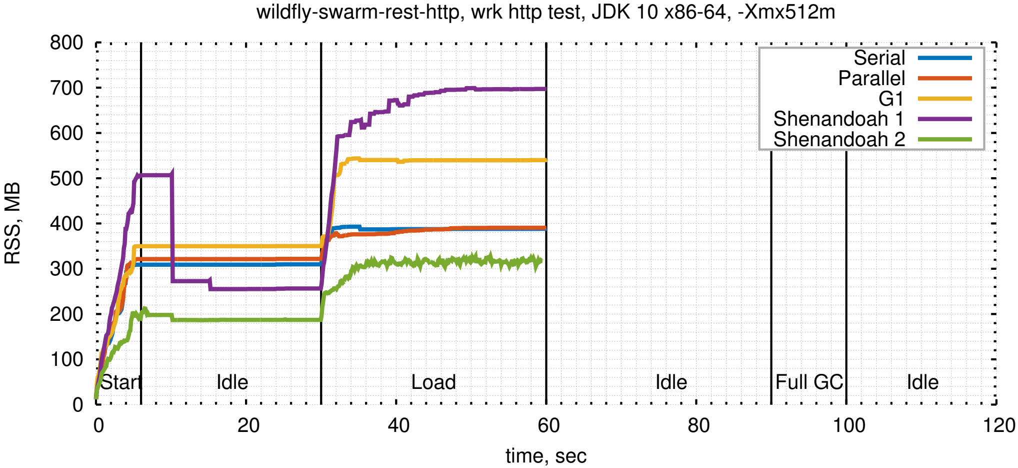


Heap Management: Microservice Example

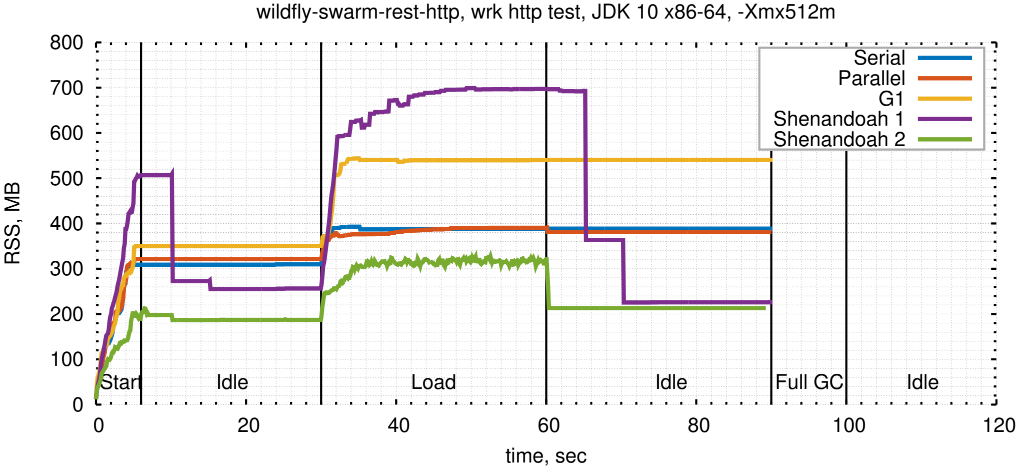


redhat

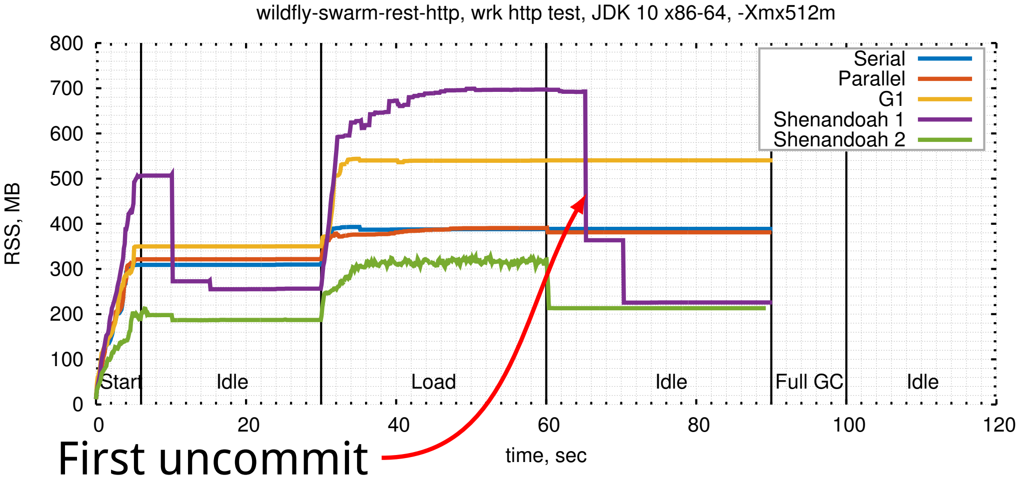
Heap Management: Microservice Example



Heap Management: Microservice Example

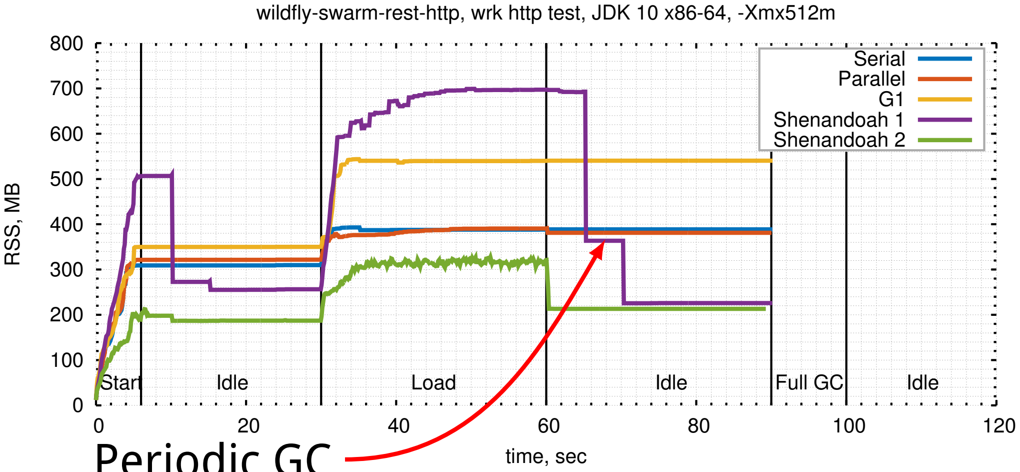


Heap Management: Microservice Example



redhat.

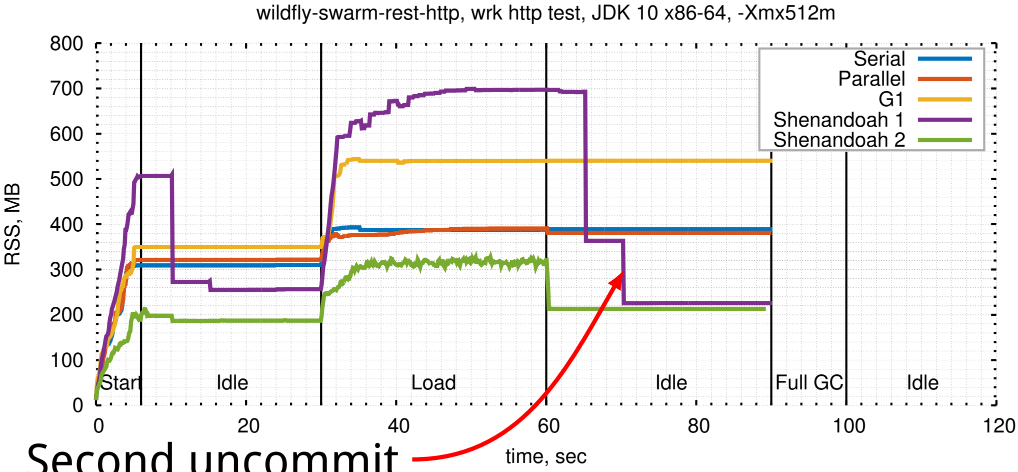
Heap Management: Microservice Example



Periodic GC

redhat

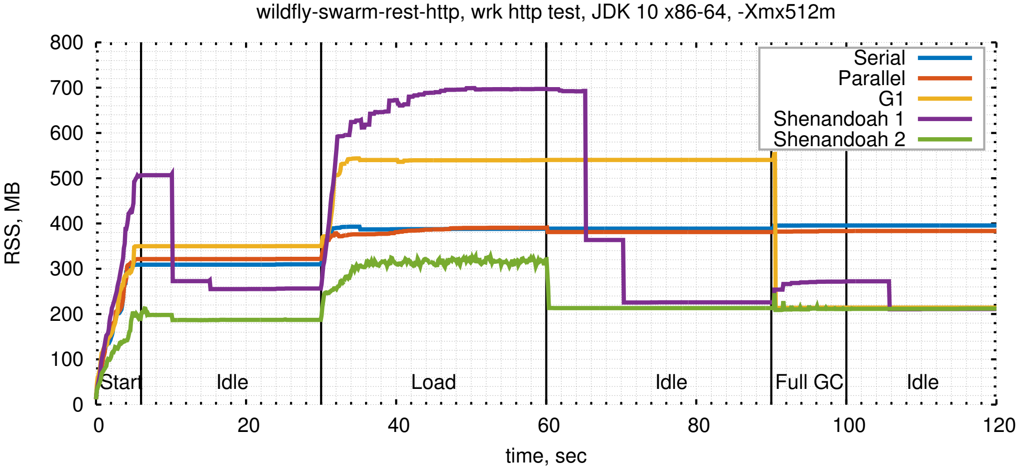
Heap Management: Microservice Example



Second uncommit

redhat.

Heap Management: Microservice Example



Heap Management: Shenandoah's M.O.

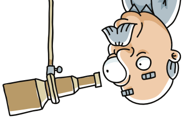
**«We shall take all the memory when we need it,
but we shall also give it back when we don't»**

1. Start with `-Xms` committed memory
2. Expand aggressively under load up to `-Xmx`
3. Stay close to `-Xmx` under load
4. Uncommit the heap and bitmaps down to zero when idle
5. Do periodic GCs to knock out floating garbage when idle

Tunables: `-Xms`, `-Xmx`, periodic GC interval, uncommit delay

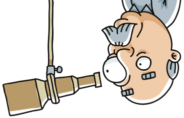
redhat

Heap Management: 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

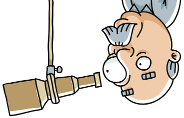
Heap Management: Footprint Tips



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
   - Smaller heaps, lower GC thread counts
   - Uncommit tuning, periodic GC. Shenandoah examples:

   ```
   -XX:ShenandoahGuaranteedGCInterval=(ms)
   -XX:ShenandoahUncommitDelay=(ms)
   ```

Heap Management: 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
   - Smaller heaps, lower GC thread counts
   - Uncommit tuning, periodic GC. Shenandoah examples:

   `-XX:ShenandoahGuaranteedGCInterval=(ms)`

   `-XX:ShenandoahUncommitDelay=(ms)`
3. Exploit GC and infra improvements
   - Java Agents that bash GC with Full GCs on idle?
   - Modern GCs that recycle memory better?

redhat

Conclusion

Conclusion: In One Paragraph

Pre-requisite: get a decent concurrent GC.

Conclusion: In One Paragraph

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: In One Paragraph

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: In One Paragraph

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

redhat.

Conclusion: Releases

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

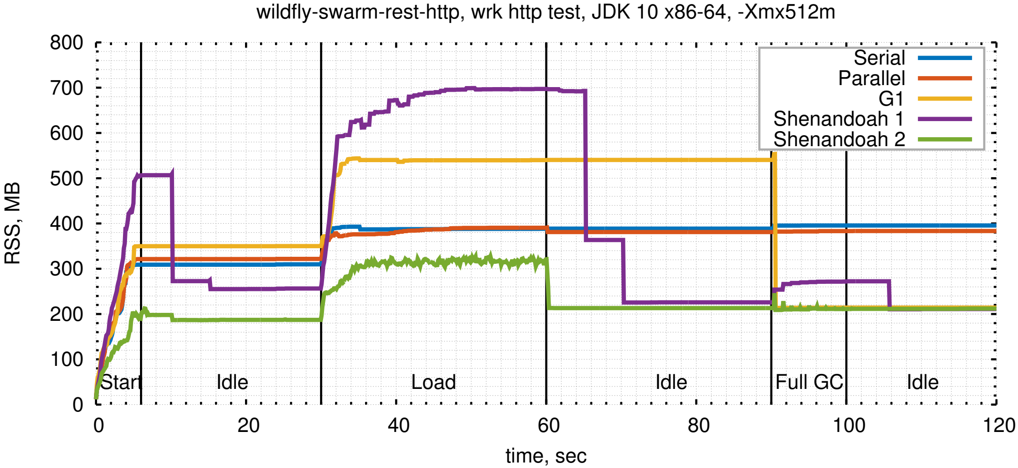
`https://wiki.openjdk.java.net/display/shenandoah/`

- Development in separate JDK 10 forest, regular backports to separate JDK 9 and 8u forests
- JDK 8u backport ships in RHEL 7.4+, Fedora 24+, and derivatives (CentOS, Oracle Linux[5], Amazon Linux, etc)
- Nightly development builds (tarballs, Docker images)

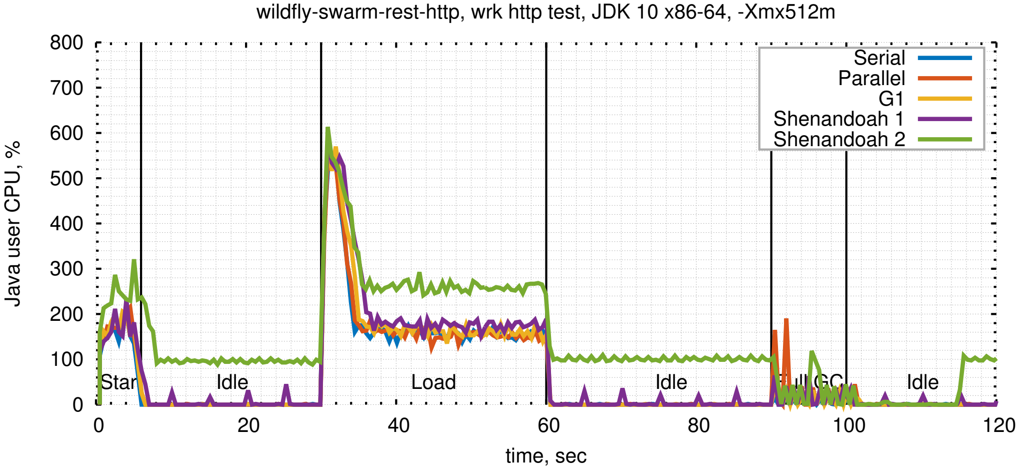
[5] One can find that quite amusing

Backup

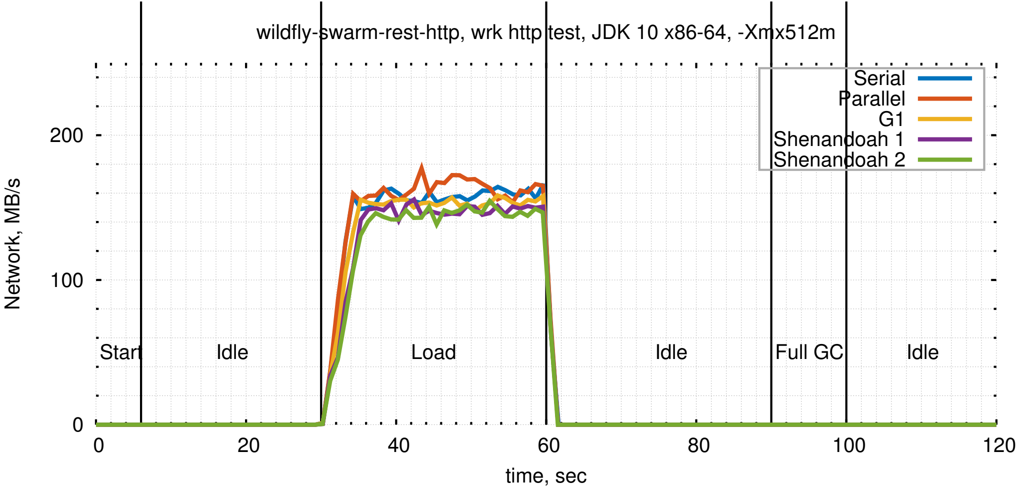
Backup: Microservice Example



Backup: Microservice Example



Backup: Microservice Example



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

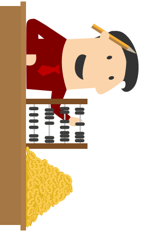
Code Roots: Fixups

```
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:

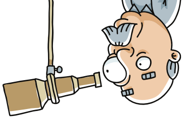


```
# jdk10/bin/java -XX:+UseShenandoahGC -Xlog:gc+stats
Pause Final Mark (G)     = 0.13 s (a =   2768 us)
Pause Final Mark (N)     = 0.10 s (a =   2623 us)
  Initial Evacuation     = 0.08 s (a =   2515 us)
    E: Code Cache Roots = 0.04 s (a =   1227 us)
```

Alternative: barriers after constants, with throughput hit

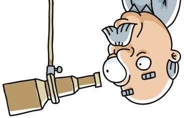
redhat.

Code Roots: Latency Tips



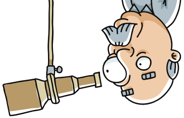
1. Have less compiled code around
   - Disable tiered compilation
   - More aggressive code cache sweeping

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

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
   - Special code cache roots recording (G1, JDK 9+)

redhat.

Cleanups

Cleanups: Problem

With 1 ms pause time budget,
processing 10K regions means 100 ns per region

- Hit a contended location ⇒ out of budget
- Want to clean aux data structures?
- Want to clean up dirty regions?
- Want to uncommit the empty regions?

redhat.

Cleanups: Cleanups

Solution: asynchronous cleanups

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

Works well, but a perception problem:
What is GC time here?