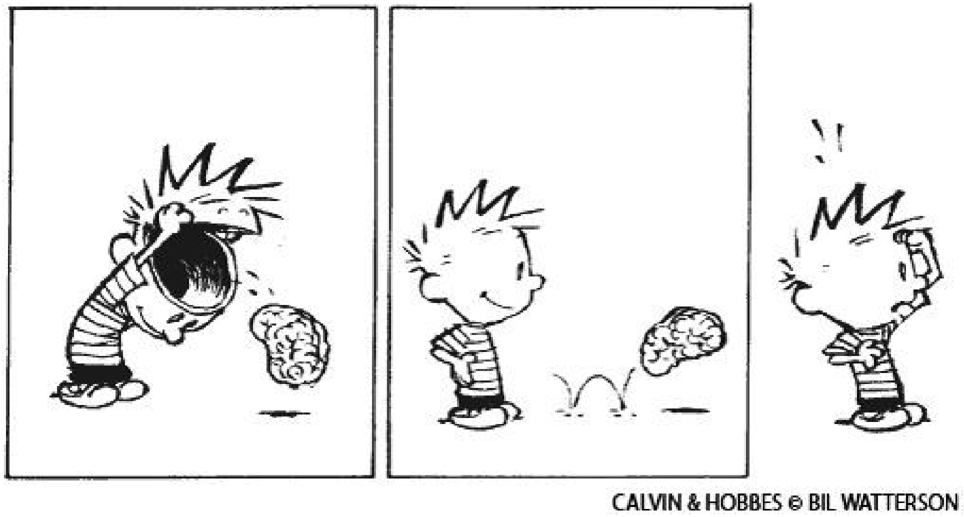
Java Garbage Collectors

Rémi Forax – June 2026



Don’t believe what I’m saying !

Vocabulary

Parallel

Uses several threads to do the GC work

Concurrent

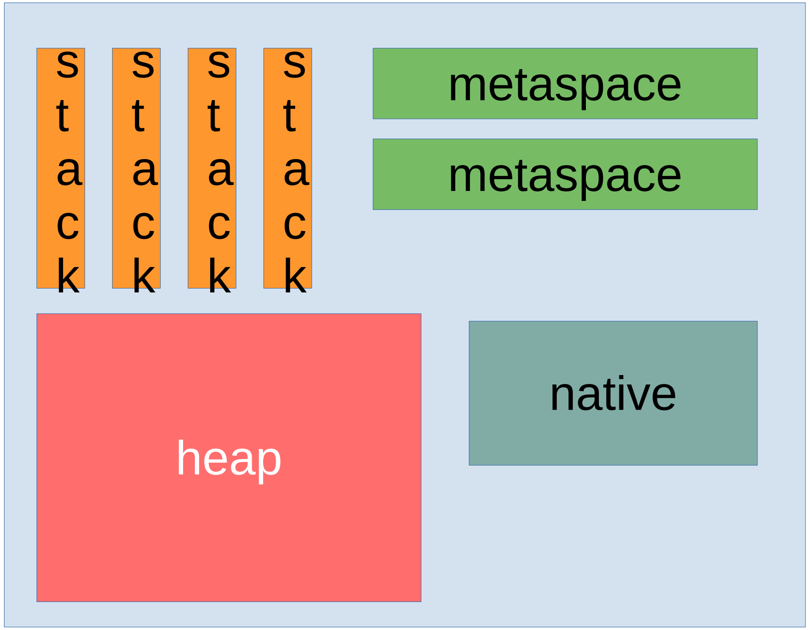
GC work is done at the same time as the application run

Incremental

Current GC cycle reclaim only parts of the heap,
but keep information for later cycles

Kind of Memory inside the JVM

- Each thread has its own stack
- GC Heap (-Xmx, -Xms)
- Native C memory (system calls)
- Metaspace
  - metadata per ClassLoader
    - Klass, bytecode, constant pool, oopMap, profiles for IC, …
- Code Cache



Java GCs

5 Garbage Collectors:

- Serial (Sun then Oracle)
- Parallel (Sun then Oracle)
- G1 (Sun then Oracle)
- Shenandoah (RedHat then Amazon)
- ZGC (Oracle)

GC Performance Trilemma

Throughput

- The % of the time spent in the application instead of in the GC

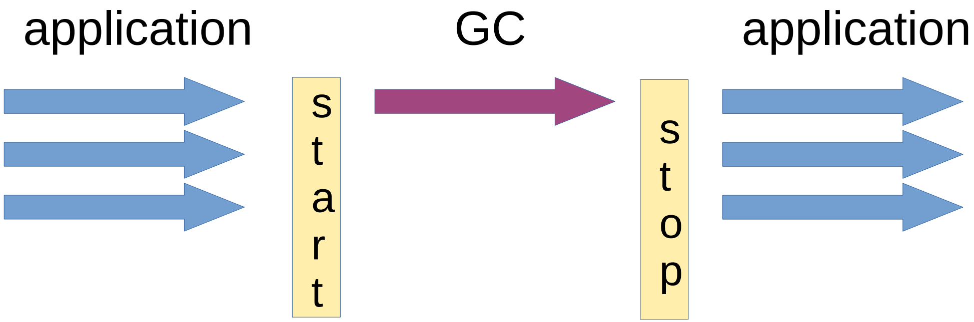
Latency

- Duration of the GC pause + GC work done by the application

Footprint

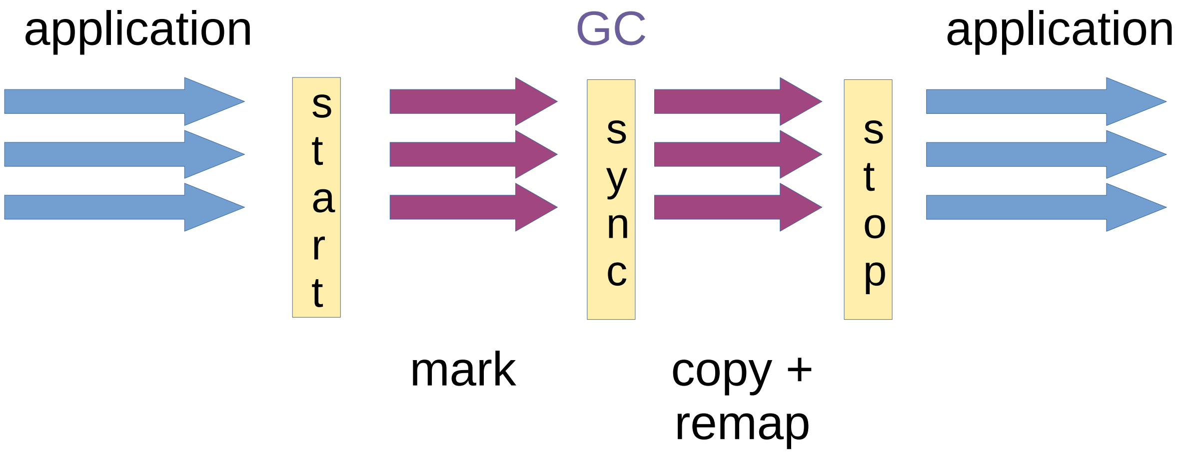
- Amount of native memory used by the GC itself

Serial GC



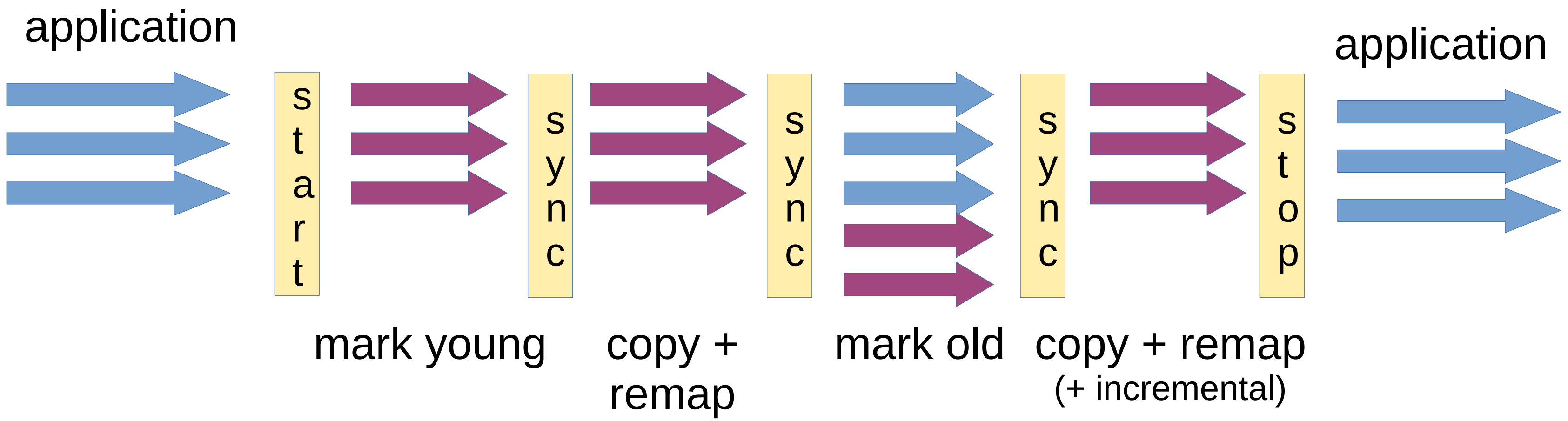
GC for CLIs and small cloud containers

Parallel GC



GC for batch programs (good throughput)

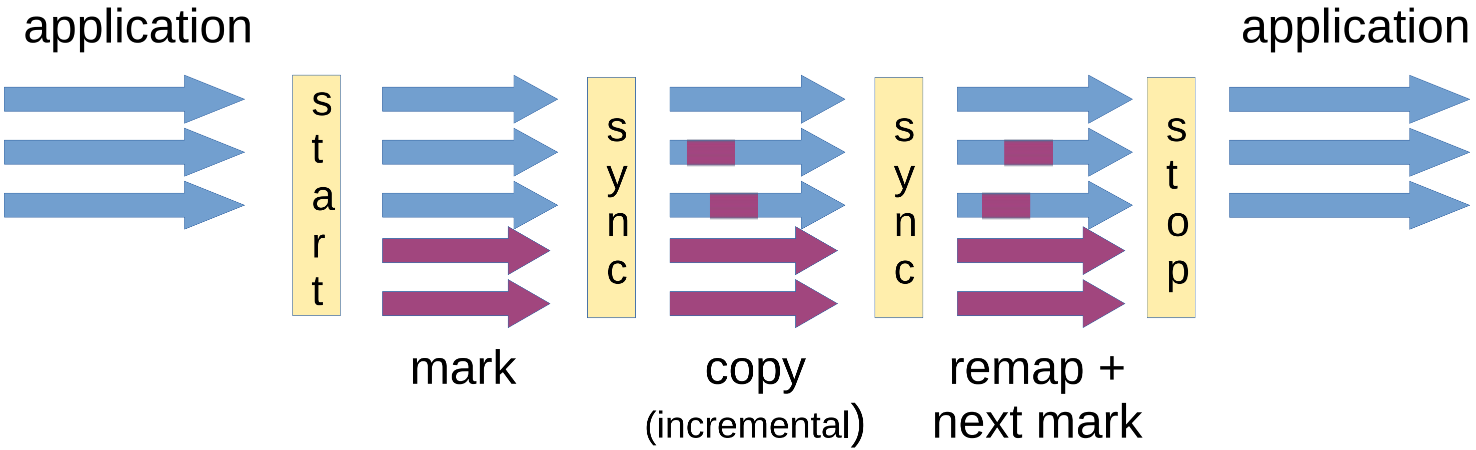
G1 GC (default)



latency goal <= 200 ms (-XX:MaxGCPauseMillis)

Garbage-first garbage collection – ISMM 2004
Printezis, Flood, and al

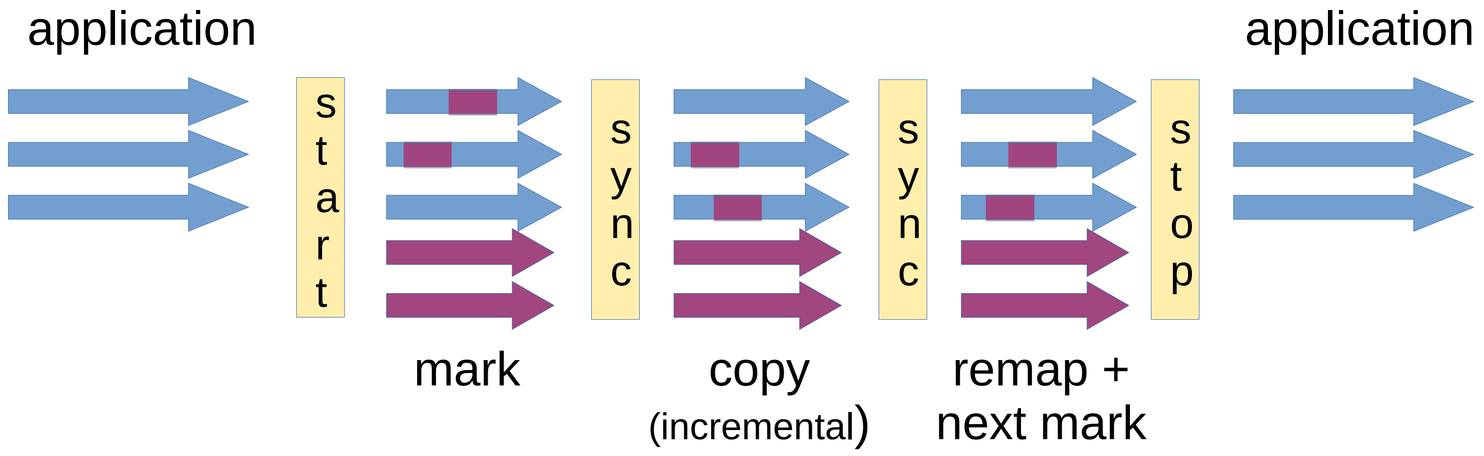
Gen? Shenandoah



latency goal <= 1 ms

Shenandoah - Clark et al. – 2021, Flood al. – 2016

Gen ZGC



latency goal <= 1 ms

Garbage Collectors

Kind of GCs

Determine liveness
- Reference counting (find dead objects, problem with cycles)
- Tracing (find live objects)
  - **Precise GC** ? (stack and heap are parseable)
    - vs conservative GC : BoehmGC

Moving GCs ?
- Non moving (interop with C), sweep → free()
- **Moving GC** (CPU vs RAM using -Xmx, pointers are opaque)

Allocation in Java

- Thread Local Allocation Buffer (TLAB)
  - Bump pointer
  - Concurrency (CAS) only at the end of the pages
- Allocators are NUMA aware

Marking Phase

Tracing

From roots, mark recursively objects by following pointers

- Roots in Java
  - references on stack
  - static fields
  - + JNI roots, VM roots

Real implementations are not recursive

=> stack overflow !

Precise GC => Safepoints

The stack frame layout has to be known for marking

- GC operations can only run at safepoints
  - At allocation site (**new** ...)
  - At the exit of a method
  - At the end of a body of a non counted loops
    (or every 1000 iterations if using a concurrent GC)

- Interpreter uses bytecode type info (abstract interpretation),
  JITs know the registers/spilled slots containing pointers
  - Stored in oopMap (metadata)
- At runtime, at a safepoint, poll to know if GC needs to run

Safepoint polls with x86_64

Loop poll (r15 is the current thread)

- Try to access a read protected? page

  movq $poll_page_offset(%r15), %rtmp
  testl %rax, (%rtmp)

Method exit poll (+ watermark comparison)

- Test %rsp > value (watermark or armed)

  cmpq %rsp, $poll_word_offset(%r15)
  ja slow_path

Tri-colors Marking

Algo: Breadth First Search

- White : not yet marked
- Grey : working set
- Black : marked and children are marked

  At the end, black and white are swapped

In Java

- grey set : 1 queue per GC threads + Work stealing
- white/black are 2 bits in a livemap (white, weak black/strong black)

Concurrent marking

Mark when application run (not STW)

- Snapshot At The Beginning (SATB)

During marking phase

- New objects are marked black
- If an object field mutated by the application,
  the referenced object is added to grey set (if ref not null)

Generational GCs

Generational Hypothesis

young objects are more likely to die than old objects

“most objects die young”

A Real-Time Garbage Collector Based on the Lifetimes of Objects Henry Lieberman and Carl Hewitt – MIT - 1981

Generation Scavenging: A non-disruptive high performance storage reclamation algorithm
David Ungar – ACM - 1984

Object Header from GC POV

Each Java object has a header with 4 age bits

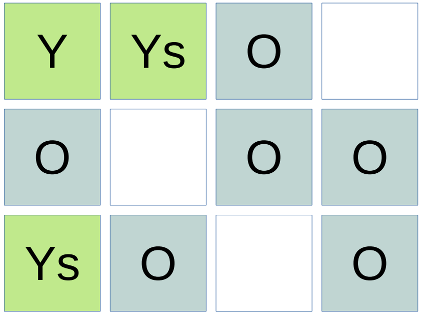
Not used by ZGC

```
Mark Word (normal):
 64                     39                              8    3  0
 [.......................HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH.AAAA.TT]

Class Word (compressed):
 32                               0
 [CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC]



Header (compact):
 64                    42                             11   7   3  0
 [CCCCCCCCCCCCCCCCCCCCCCHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHVVVVAAAASTT]
 (Compressed Class Pointer)       (Hash Code)          /(GC Age)^(Tag)
```

Regions

Contiguous (Serial, Parallel)

Partitions (G1, Shenandoah, ZGC)



Y = young
O = old

Regions have different sizes
2M < size < as big as an array

Generational GCs

Different kind of collection

- Minor
  - mark young (+ RememberSets) and collect young
- Major (10 to 100 x less)
  - mark all and collect old
- Full (`System.gc()` or allocation failure)
  - mark all and collect all

Remember Set

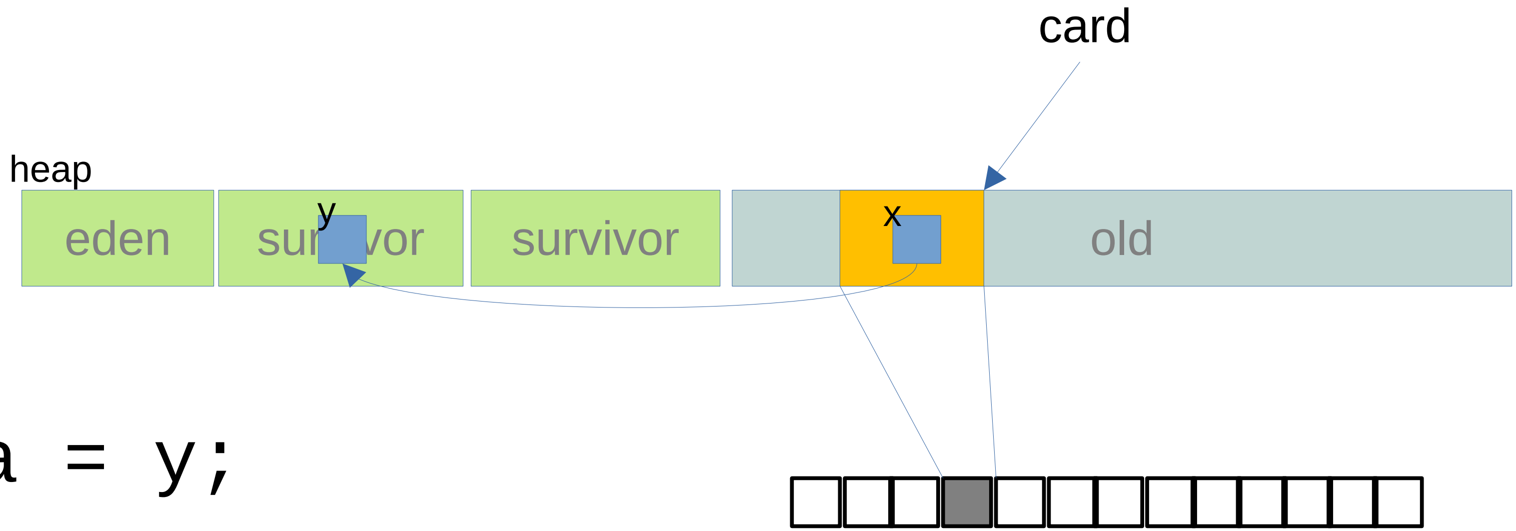
Register the addresses of an old region containing a pointer to a young region

- Updated when the application does a write (Write Barrier)

- Implementations:
  - Serial / Parallel: use a “fuzzy” bitset of card (512 bytes)
  - Concurrent GCs: specific data structure (fine/sparse/coarse)

A RememberSet is a root for minor collections

RSet using a card table



`x.a = y;`

each bit is 1 card of the old region

G1 new Write Barrier

Before Java 26, the RSet is a concurrent hash map

- too complex for a Write Barrier
  - Less throughput
- Idea: two BitSets (one read/one write) to avoid concurrent updates

Two BitSet of cards

- Applications thread (WB) write a bit
- GC threads read and update the RememberSet then clear the bit
- Swap at the stat of the marking

G1 new Write Barrier

Snippet of code added after each field write

Write Barrier for `x.a = y;`

```
if (region(x.a) == region(y)) goto done;  // Ignore refs same region
if (y == nullptr) goto done;               // Ignore null
if (card(x.a) != CLEAN) goto done;         // Ignore non-clean card
*card(x.a) = DIRTY;                        // Mark the card
done:
```

Collection of old regions

Old generation is usually big

- Slow if collected in one go => collect incrementally

- Marking old gen
  - starts with marking young

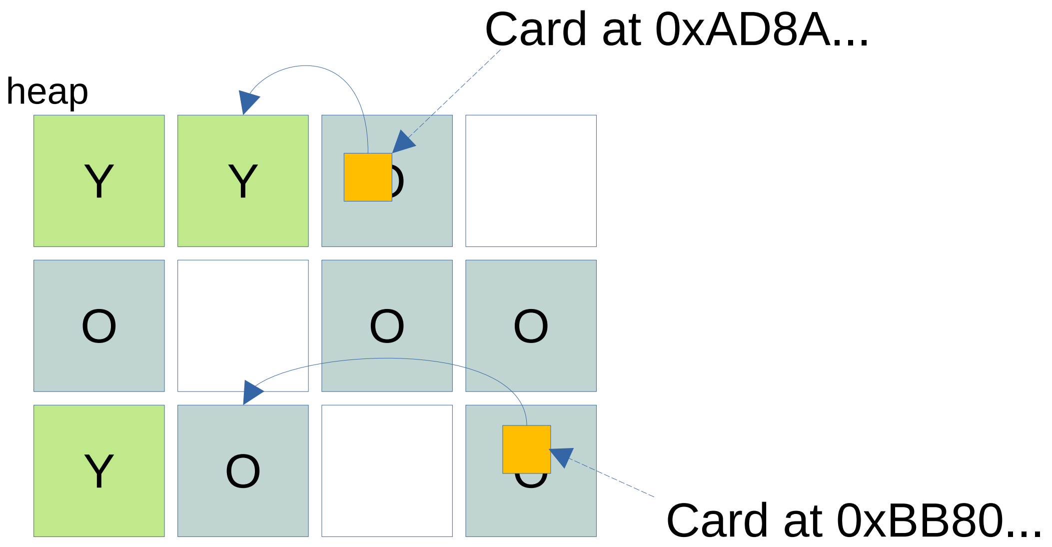
    \+ young to old pointers are additional roots
  - concurrent marking

    \+ compute the RememberSet in between each old regions

Transient RememberSet

Young regions share a RememberSet

- When doing a major collection
  - Each old region has its own transient RememberSet



Young RememberSet
0xAD8A...

Zone 9 RememberSet
0xBB80...

Evacuation Phase

G1 (Garbage First) GC

- Incremental evacuation
  - Sort young regions by % garbage
    - Sort old regions by % garbage and freed space
  - Evacuate incrementally the first regions in latency budget
    - Object headers in the from-space point to the new address
    - Non evacuated zones will be in next GC cycles

- GC must to be triggered before allocation pool exhausted

Shenandoah GC

G1 algo + concurrent evacuation and remap

Load Barrier

- application threads can also do object evacuation

GenShen (generational):

- Major marking = minor marking but ref to old gen are kept as root for old gen marking
- Old gen regions are evacuated incrementally by young collections

Load Barrier

Snippet of code inserted in front of each field read

- Copy object if necessary, fix source field pointer

- Load Barrier of Shenandoah

```
if ((thread.gc_state & HAS_FORWARDED) != 0)
  if ($in_cset_fast_test[region(x.a)])    // region in collection set ?
    slow_path();                          // copy? and fix x.a
```

Floating garbage ?

Even if a region if fully evacuated, it can not be re-used

- We have to wait until all pointers have been fixed

  => floating garbage

Heap has to be dimensioned accordingly

ZGC

Fully concurrent (mark + evacuation + remap)

- Application and GC work together
- 2 GCs in parallel (young GC, old GC)

Use pointer coloring + Load/Write Barriers

- Mark bits (if invalid => add to the grey set)
- Remap bits (if invalid => object must be copied)
- Free region even with dangling pointers

Colors of pointers

Store GC states in low bits of pointers of the heap

12 color bits:

- 4 bits **R**emap: 1000, 0100, 0010, 0001 (where it goes?)
- 4 bits **M**ark: Mark0, Mark1 * 2 (old or young)
- 2 bits **F**inalizer: 10 or 01 (reachable only using finalizer)
- 2 bits RememberSet (**r**): 01, 10, 11

R0	R0	R1	R1	M0	M0	M1	M1	F	F	r	r

Good colors

Load Good

...	R0	R0	R1	R1	M0	M0	M1	M1	F	F	r	r

bit 0

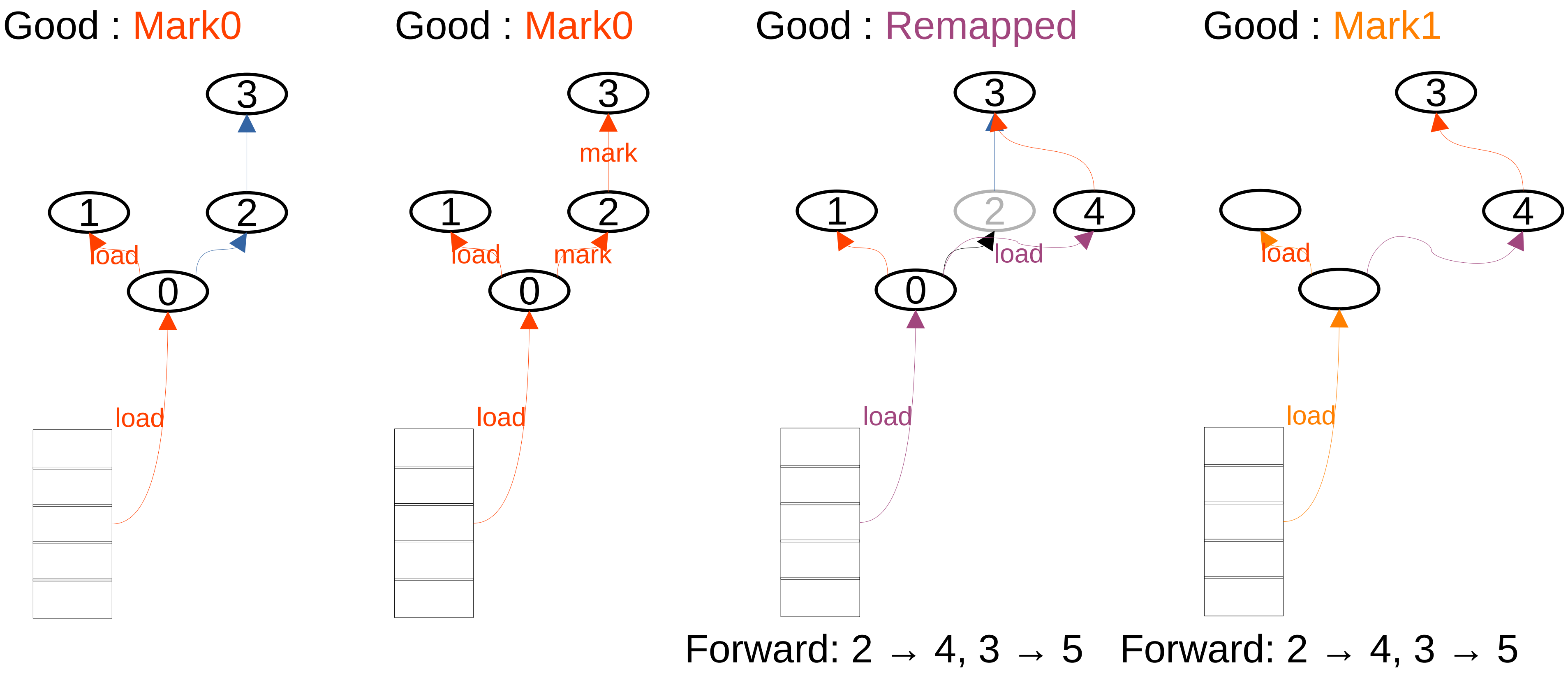
Mark Good

...	R0	R0	R1	R1	M0	M0	M1	M1	F	F	r	r

Store Good

...	R0	R0	R1	R1	M0	M0	M1	M1	F	F	r	r

Color during Marking/Remapping



ZGC Load/Write Barrier intel x64

Load Barrier

```
movq 0x10(%rax), %rcx
shrq $good_remap_bit_index, %rcx   // load good or null + uncolor
ja slow_path
```

Store Barrier

```
testq $store_bad_mask, 0x10(%rax)  // store good or null
jnz slow_path
shlq $good_remap_bit_index, %rcx
orq $store_good_color, %rcx            // recolor
movq %rcx, 0x10(%rax)
```

Summary

Java GCs

5 generational GCs :

- Serial STW, 1 GC thread, WB
- Parallel STW, n GC threads, WB
- G1 regional, concurrent mark, STW incremental evacuation, n GC threads, latency < 200ms, WB
- Shenandoah regional, concurrent, latency < 1ms, n GC threads, WB + LB (not generational by default)
- ZGC regional, concurrent, latency < 1ms, n GC threads, WB + LB (restricted to 64 bits pointers)