criteo

ADVANCED

Understanding JVM GC

Jean-Philippe BEMPEL

WebScale
@jpbempel

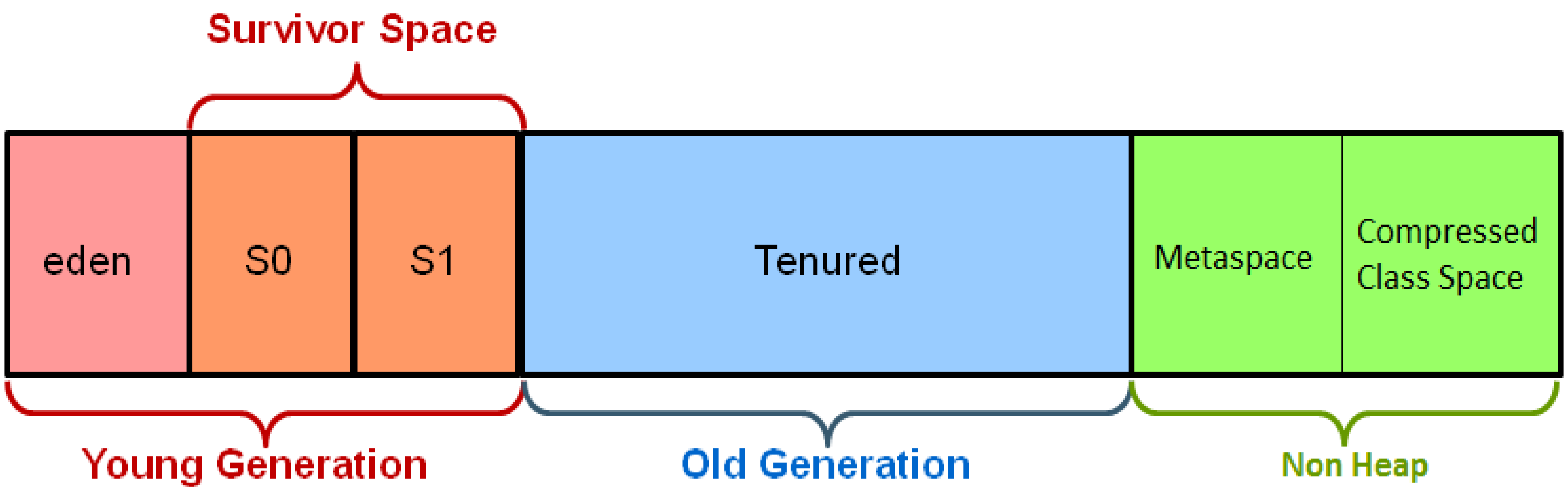
Understanding JVM GC: Advanced!

- GC basics
- G1
- Shenandoah
- Azul's C4
- ZGC
- How to choose a GC algorithm?

GC Basics

criteo

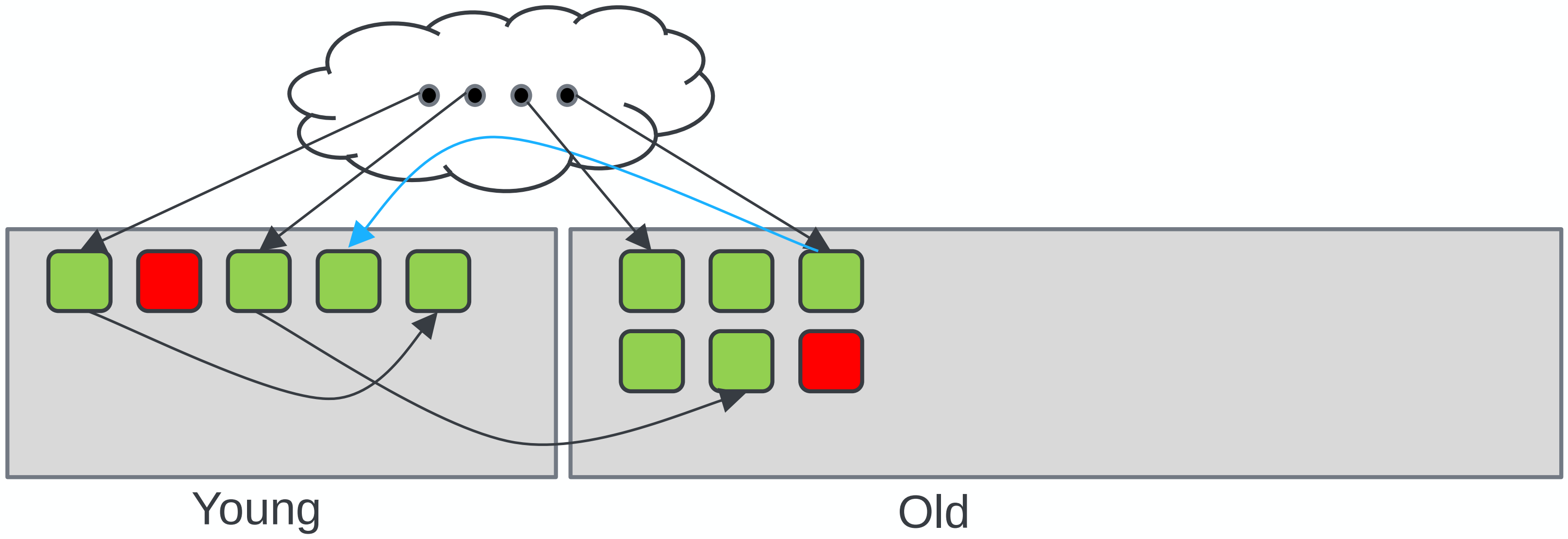
Generations



Marking for Minor GC

- Traversing references to mark live objects
  - Stopping when reaching old generation

- From GC roots (static fields, thread stack, JNI)



Card Table

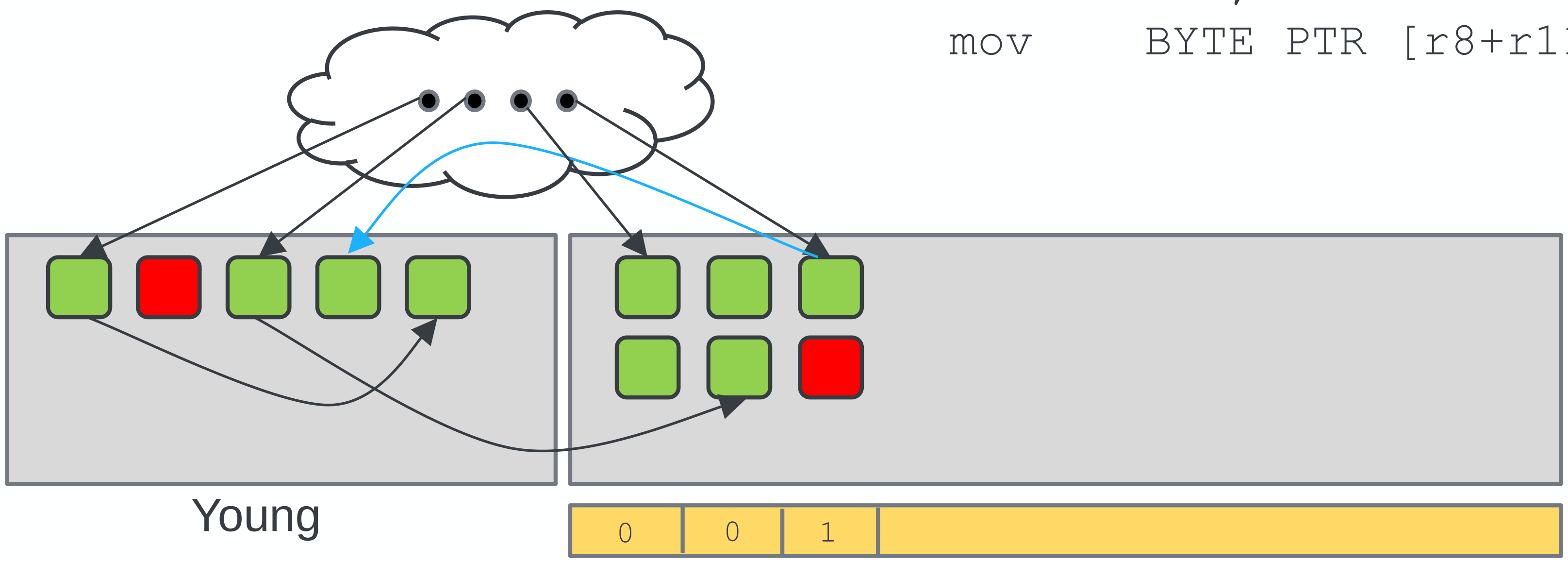
Card Table for references old -> young references

Write barrier to update card table on assignation

X.f = Y

```
CARD_TABLE[&X >> 9] = 1
```

```
mov     DWORD PTR [r10+0x6c],r8d
mov     r11,r10
shr     r11,0x9
mov     r8d,0x2383000
mov     BYTE PTR [r8+r11*1],r12b
```



G1

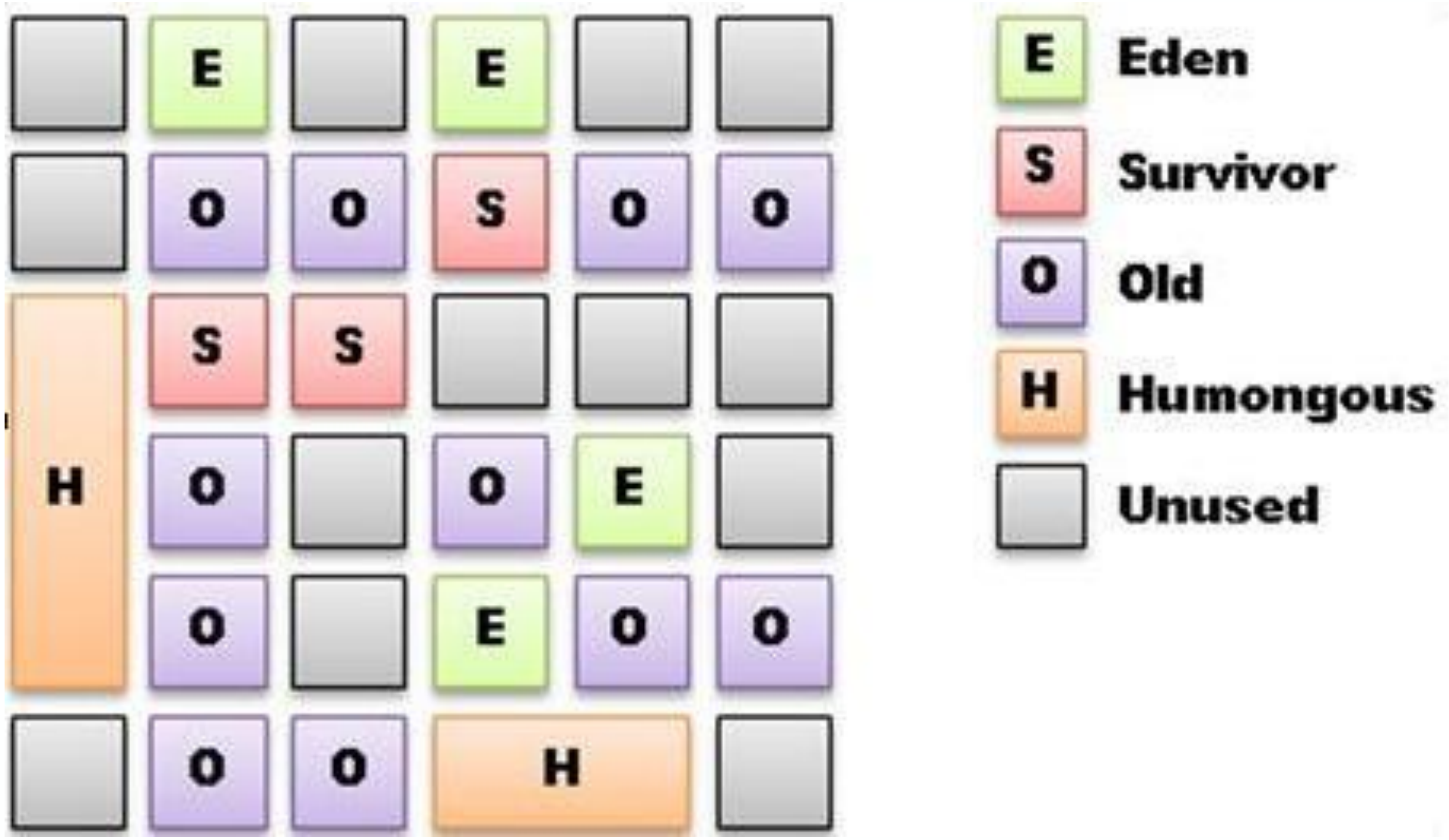
criteo

Garbage First

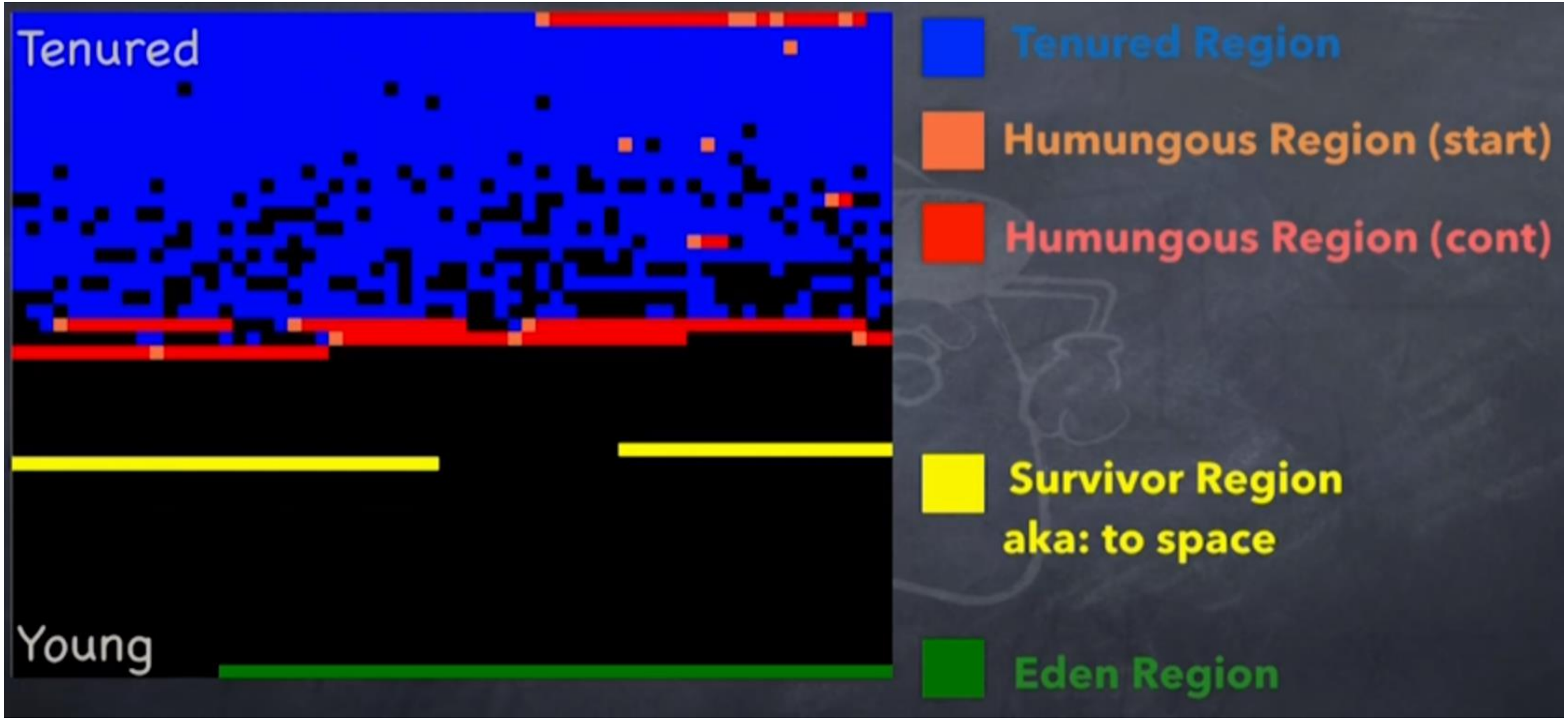
- Generational
- Region based
- Pause time target (soft real-time)
  - `-XX:MaxGCPauseMillis=n` (default 200)
- Default GC since JDK9

Regions

Heap divided into fixed-size regions



Regions



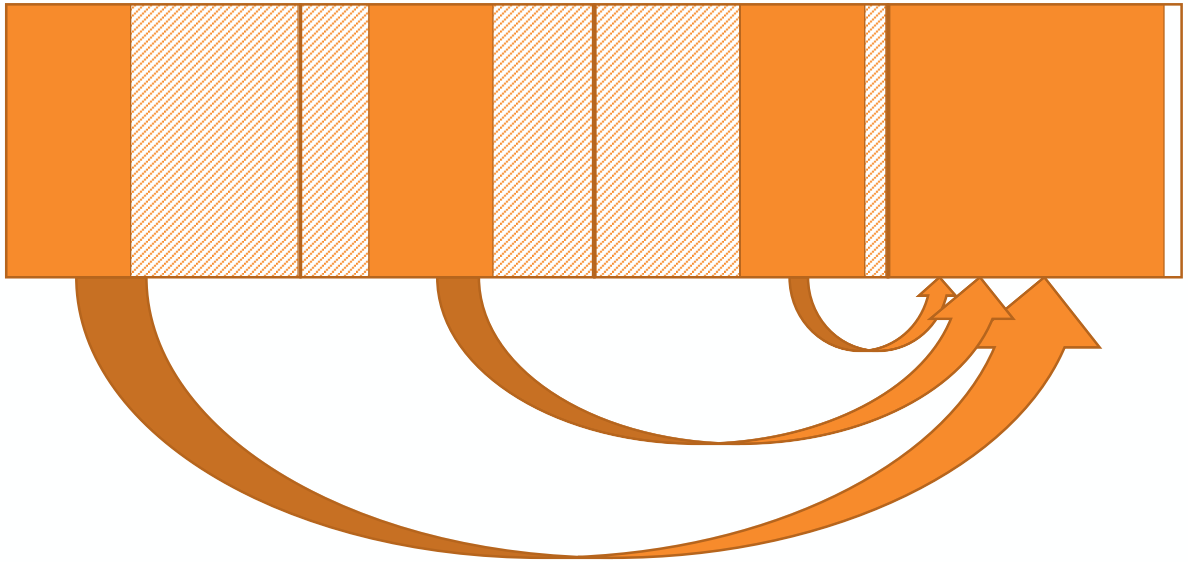
Credit: Kirk Pepperdine

G1 phases

- Young collection (STW)
- Initial Mark (STW)
- Concurrent Marking
- Final Remark (STW)
- Cleanup (STW)
- Mixed collection (STW)

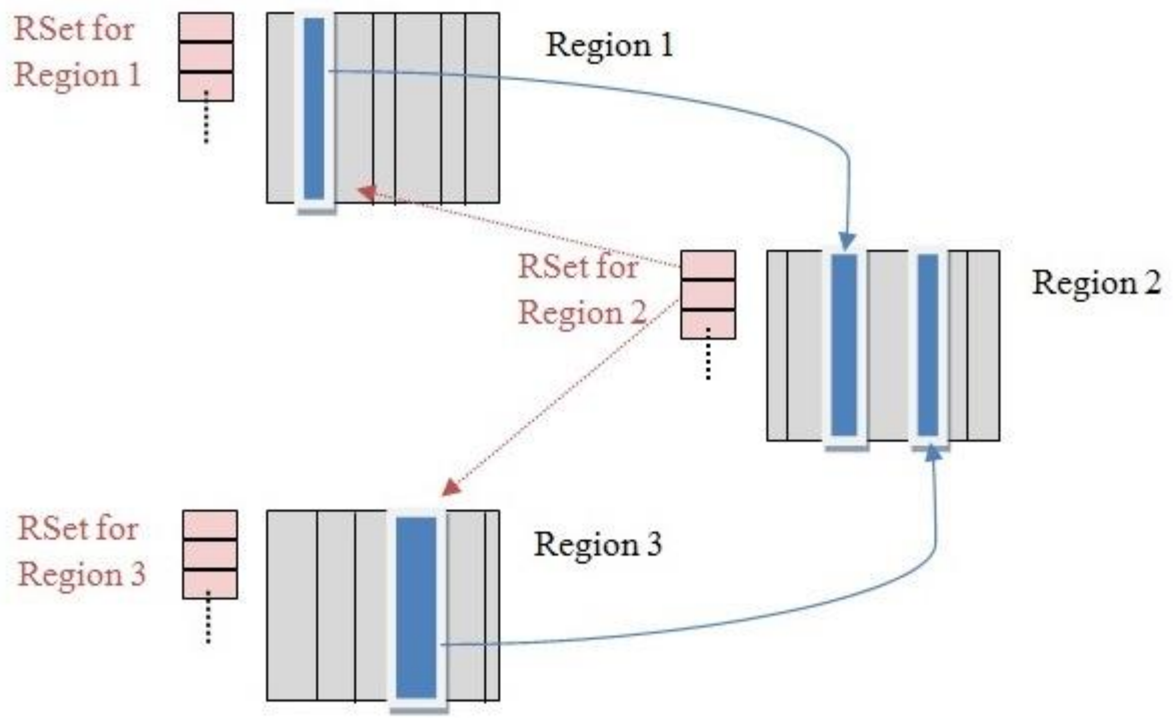
Young GC

- Stop-The-World event
- Evacuates live objects to Survivor or Old regions
- Only objects in young generation are considered



Remembered Sets

- Card table per region
- Avoid scanning the entire heap



Remembered Sets: Post Write Barrier

- For each reference assignation (X.f = Y) we need to check:
  - References (X & Y) are NOT in the same region
  - Y is not null
  - => enqueue for Remebered Set processing
- Refinement threads to process the queue
- Additional instructions added after assignation

```
if (!isInSameRegion(X, Y)
    && Y != null)
        RSEnqueue(X)
```

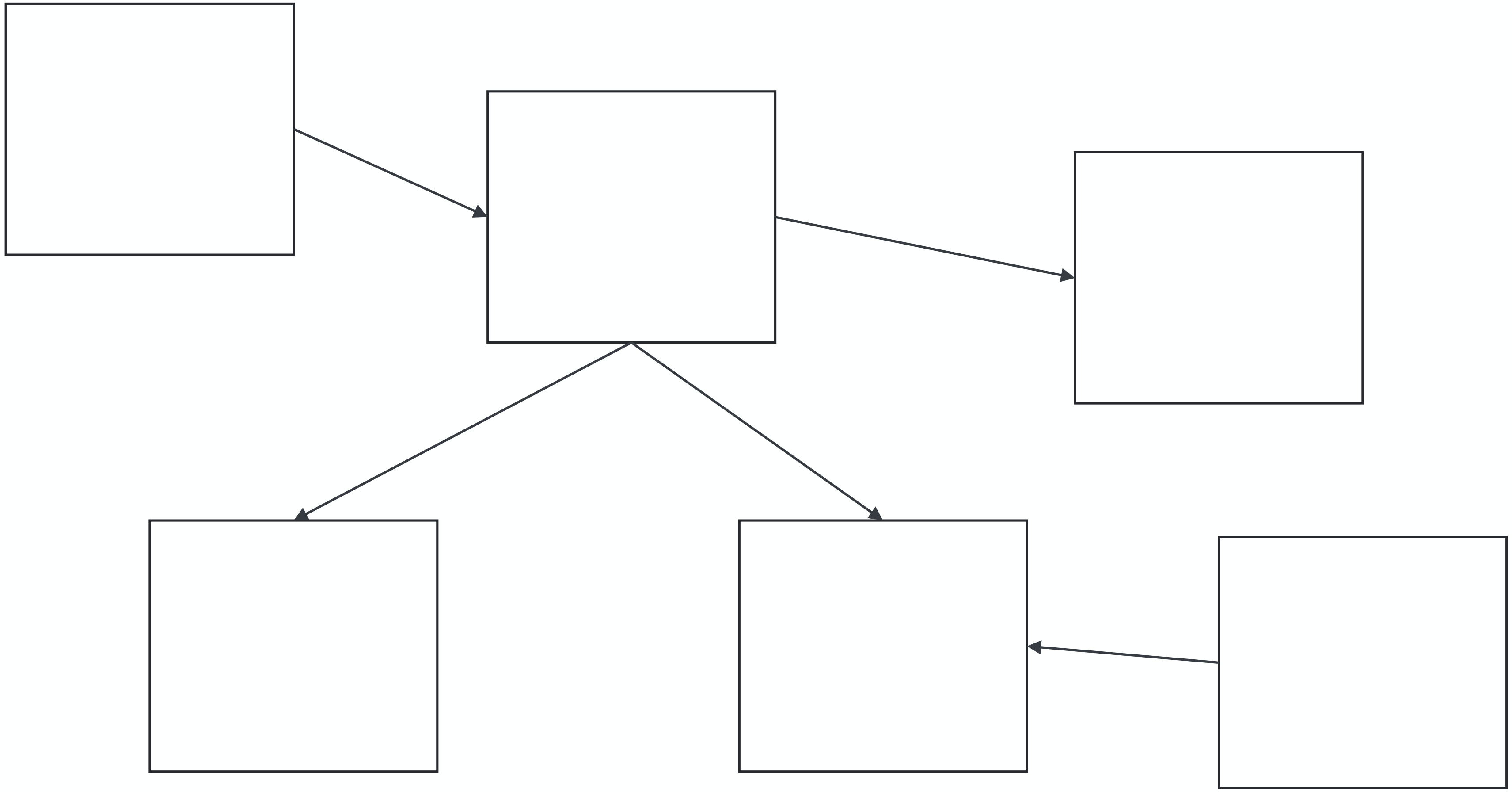
```
mov    DWORD PTR [rbp+0x74],r10d
mov    r11,rbp
mov    r8,r10
shl    r8,0x3
xor    r8,r11
shr    r8,0x14
test   r8,r8
je     cont
test   r10d,r10d
je     cont
shr    r11,0x9
movabs rcx,0x2965ecc3000
add    rcx,r11
cmp    BYTE PTR [rcx],0x20
je     cont
mov    r10,QWORD PTR [r15+0x70]
mov    r11,QWORD PTR [r15+0x80]
lock add DWORD PTR [rsp-0x40],0x0
cmp    BYTE PTR [rcx],0x0
je     cont
mov    BYTE PTR [rcx],0x0
test   r10,r10
jne    0x000002965edc62bc
mov    rdx,r15
movabs r10,0x7ffac2febc30
call   r10
jmp    cont
mov    QWORD PTR [r11+r10*1-0x8],rcx
add    r10,0xfffffffffffffff8
mov    QWORD PTR [r15+0x70],r10
```


criteo.

Concurrent Marking

- Triggered based on Initiating Heap Occupancy Percent flag (IHOP default to 45%)
- Try to mark the whole object graph concurrently with the application running
- Based on Tri-color abstraction & Snapshot-At-The-Beginning algorithm

Concurrent Marking: Tri-Color Abstraction



Concurrent Marking: Issues

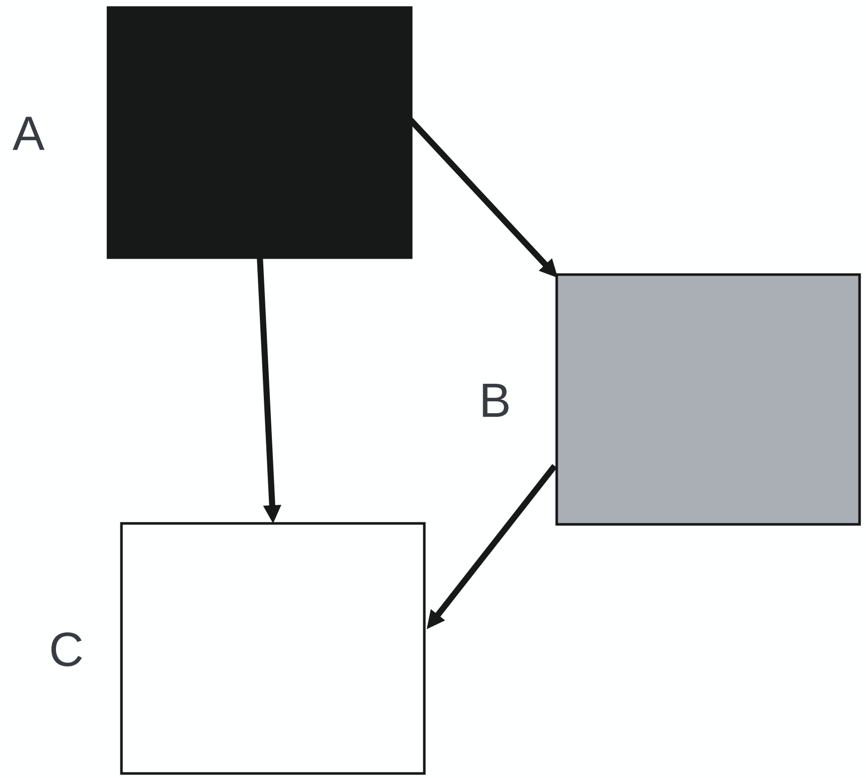
- New allocations during marking phase can be handled by:
  - Marking automatically object at allocation
  - Not considering new allocations for the current cycle

- Tri-Color abstraction provides 2 properties of missed object:

  1. The mutator stores a reference to a white object into a black object.
  2. All paths from any gray objects to that white object are destroyed.

http://www.memorymanagement.org/glossary/s.html#term-snapshot-at-the-beginning

Concurrent Marking: Issues




```
A.field1 = C;
B.field2 = null;
```

OOPS!

Concurrent Marking: Resolving misses

- 2 ways to ensure not missing any marking
- For SATB, Pre-Write Barriers, recording object for marking
- SATB barrier is only active when Marking is on (global state)

```
if (SATB_WriteBarrier) {
  if (X.f != null)
     SATB_enqueue(X.f);
}
```

```
cmp    BYTE PTR [r15+0x30],0x0
jne    0x000002965edc62e5
[...]
mov    r11d,DWORD PTR [rbp+0x74]
test   r11d,r11d
je     0x000002965edc6253
mov    r10,QWORD PTR [r15+0x38]
mov    rcx,r11
shl    rcx,0x3
test   r10,r10
je     0x000002965edc6318
mov    r11,QWORD PTR [r15+0x48]
mov    QWORD PTR [r11+r10*1-0x8],rcx
add    r10,0xfffffffffffffff8
mov    QWORD PTR [r15+0x38],r10
jmp    0x000002965edc6253
mov    rdx,r15
movabs r10,0x7ffac2febc50
call   r10
jmp    0x000002965edc6253
```

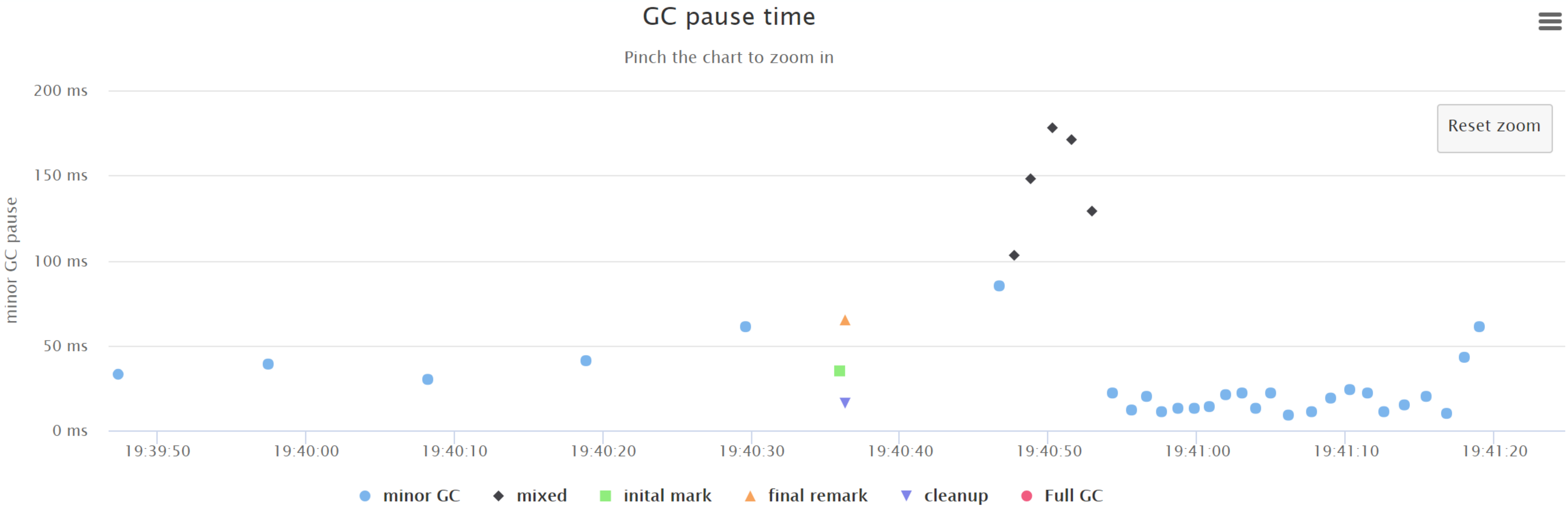
CollectionSet

- At the end of Marking, we have per region liveness information
- Regions are sorted by liveness (ascending)
- Regions full of garbage are collected during cleanup STW phase
- CollectionSet is built based on
  - Liveness, up until thresholds (G1HeapWastePercent, G1MixedGCLiveThresholdPercent)
  - Maximum number of regions (G1OldCSetRegionThresholdPercent)

Mixed GC

- Based on CollectionSet, G1 schedule to collect part of old regions
- When a Young is triggered, old regions to collect are piggy backed
- Not all old regions are considered to not waste time and reach the pause goal
- Several Young GCs can be used to collect old regions (mixed event)

Mixed GC



GC pause time
Pinch the chart to zoom in
Reset zoom
200 ms
150 ms
100 ms
50 ms
0 ms
minor GC pause
19:39:50
19:40:00
19:40:10
19:40:20
19:40:30
19:40:40
19:40:50
19:41:00
19:41:10
19:41:20
minor GC
mixed
inital mark
final remark
cleanup
Full GC

FullGC

- Still fallback to FullGC (serial < JDK10)

- Fragmentation can still happen (regions with lot of lived objects)

- Still unpredictable

Shenandoah

Shenandoah GC

- Non-generational (still option for partial collection)
- Region based
- Use Read Barrier: Brooks pointer
- Self-Healing
  - Cooperation between mutator threads & GC threads
  - Only for concurrent compaction
- Mostly based on G1 but with concurrent compaction

Shenandoah Phases

- Initial Marking (STW)
- Concurrent Marking
- Final Remark (STW)
- Concurrent Cleanup
- Concurrent Evacuation
- Init Update References (STW)
- Concurrent Update References
- Final Update References (STW)
- Concurrent Cleanup

Concurrent Marking

- SATB-style (like G1)
- 2 STW pauses for Initial Mark & Final Remark
- Conditional Write Barrier
  - To deal with concurrent modification of object graph

Concurrent Evacuation

- Same principle than G1:
  - Build CollectionSet with Garbage First!
  - Evacuate to new regions to release the region for reuse

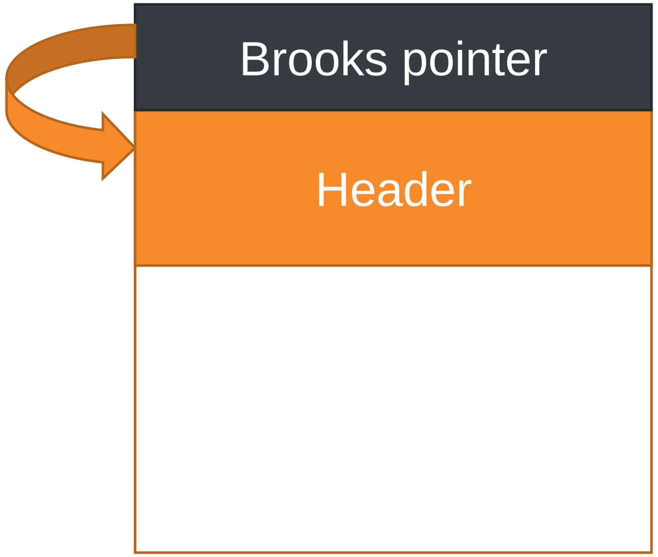
- Concurrent Evacuation done with the help of:
  - 1 Read Barrier : Brooks pointer
  - 4 Write Barriers

- Barriers help to keep the to-space invariant:
  - All Writes are made into an object in to-space

criteo

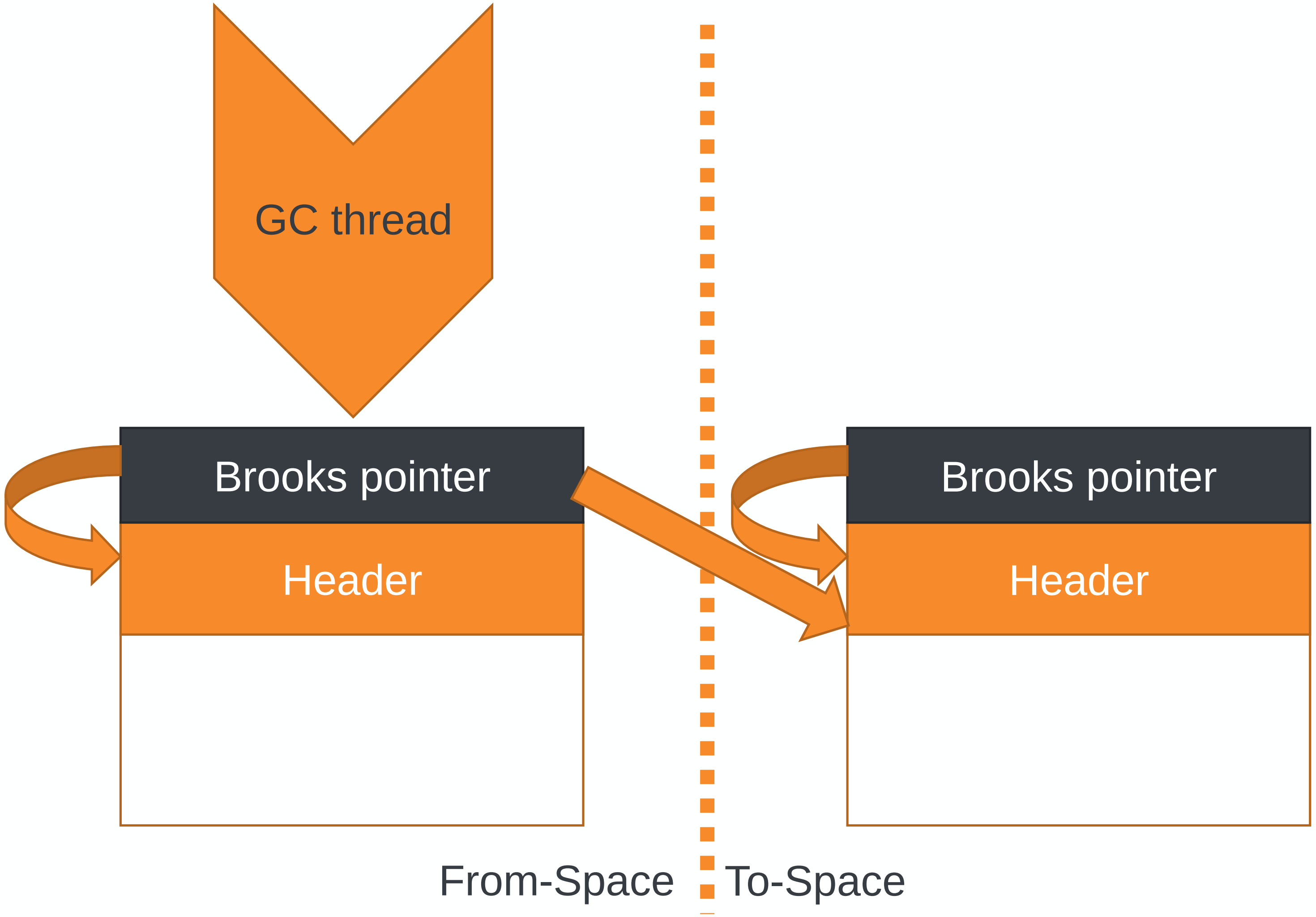
Brooks pointers

- All objects have an additional forwarding pointer
  - Placed before the regular object
- Dereference the forwarding pointer for each access
  - Memory footprint overhead
  - Throughput overhead

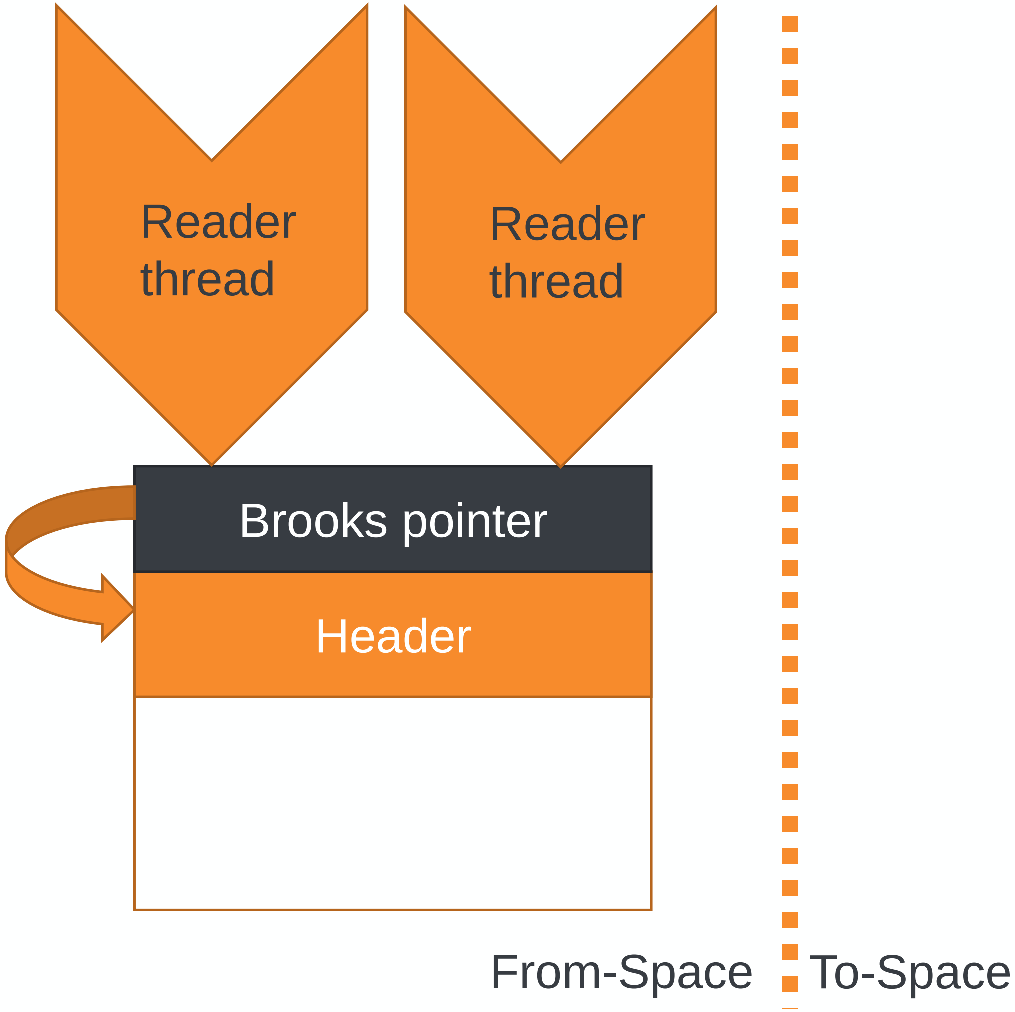



```
mov     r13,QWORD PTR [r12+r14*8-0x8]
```

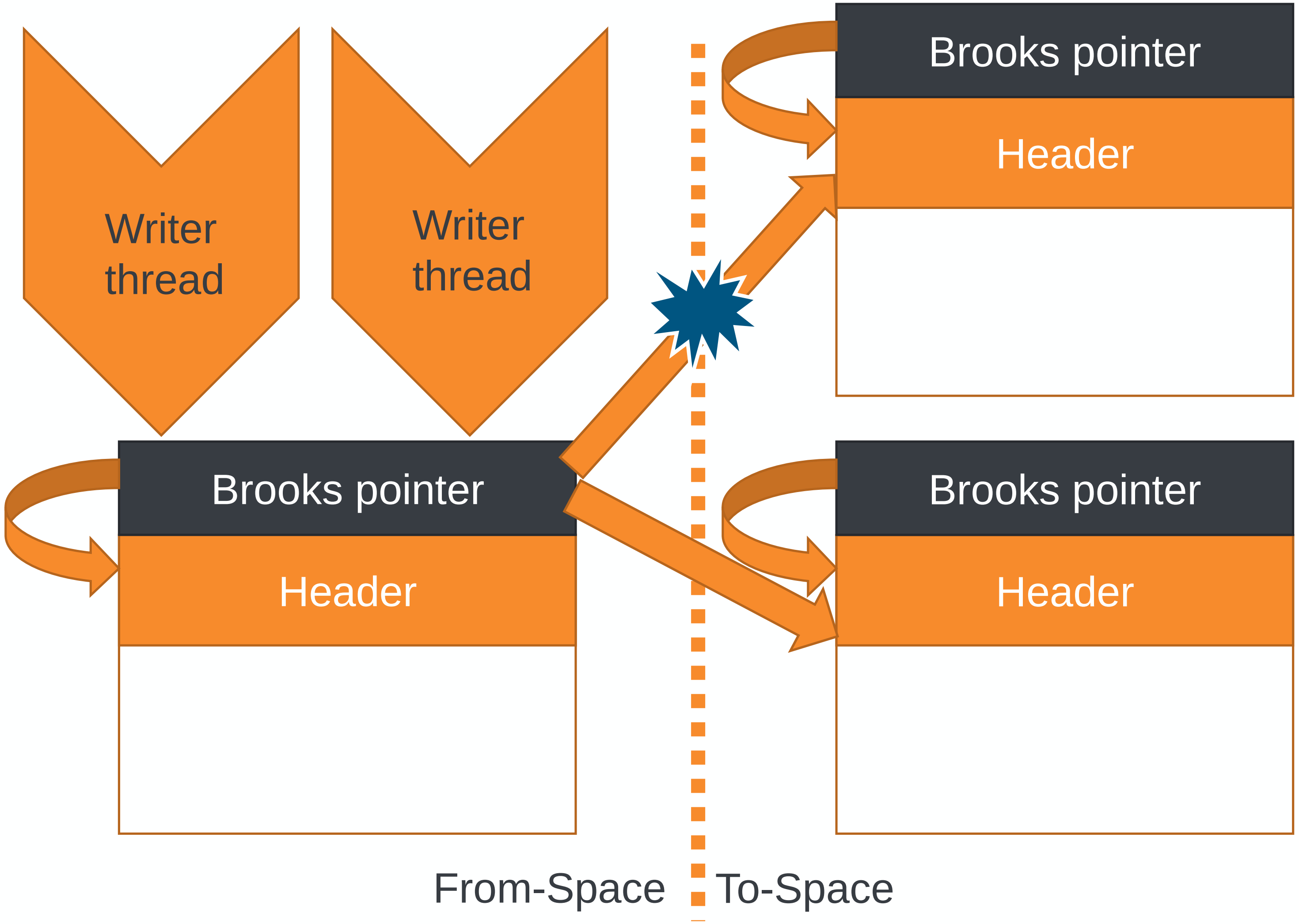
Concurrent Copy: GC thread



Concurrent Copy: Reader threads



Concurrent Copy: Writer threads



criteo

Write Barriers

- Any writes (even primitives) to from-space object needs to be protected

```
if (evacInProgress
    && inCollectionSet(obj)
    && notCopyYet(obj)) {
      evacuateObject(obj)
}
```

```
test    BYTE PTR [r15+0x3c0],0x2
jne     0x000000000281bcbc
[...]
mov     r10d,DWORD PTR [r13+0xc]
test    r10d,r10d
je      0x000000000281bc2b
mov     r11,QWORD PTR [r15+0x360]
mov     rcx,r10
shl     rcx,0x3
test    r11,r11
je      0x000000000281bd0d
[...]
mov     rdx,r15
movabs  r10,0x62d1f660
call    r10
jmp     0x000000000281bc2b
```

- Exotic barriers:
  - acmp (pointer comparison)
  - CAS
  - clone

criteo.

Extreme cases

- Late memory release
  - Only happens when all refs updated (Concurrent Cleanup phase)

- Allocations can overrun the GC

- Failure modes:
  - Pacing
  - Degenerated GC
  - FullGC

criteo.

Azul's C4

criteo

Continuously Concurrent Compacting Collector

- Generational (young & old)
- Region based (pages)
- Use Read Barrier: Loaded Value Barrier
- Self-Healing
  - Cooperation between mutator threads & GC threads
- Pauseless algorithm but implementation requires safepoints
- Pauses are most of the time < 1ms

LVB

- Baker-style Barrier
  - move objects through forwarding addresses stored aside
  - Applied at load time, not when dereferencing

- Ensure C4 invariants:
  - Marked Through the current cycle
  - Not relocated

- If not => Self-healing process to correct it
  - Mark object
  - Relocate & correct reference

- Checked for each reference loads
  - Benefits from JIT optimization for caching loaded value (registers)

LVB

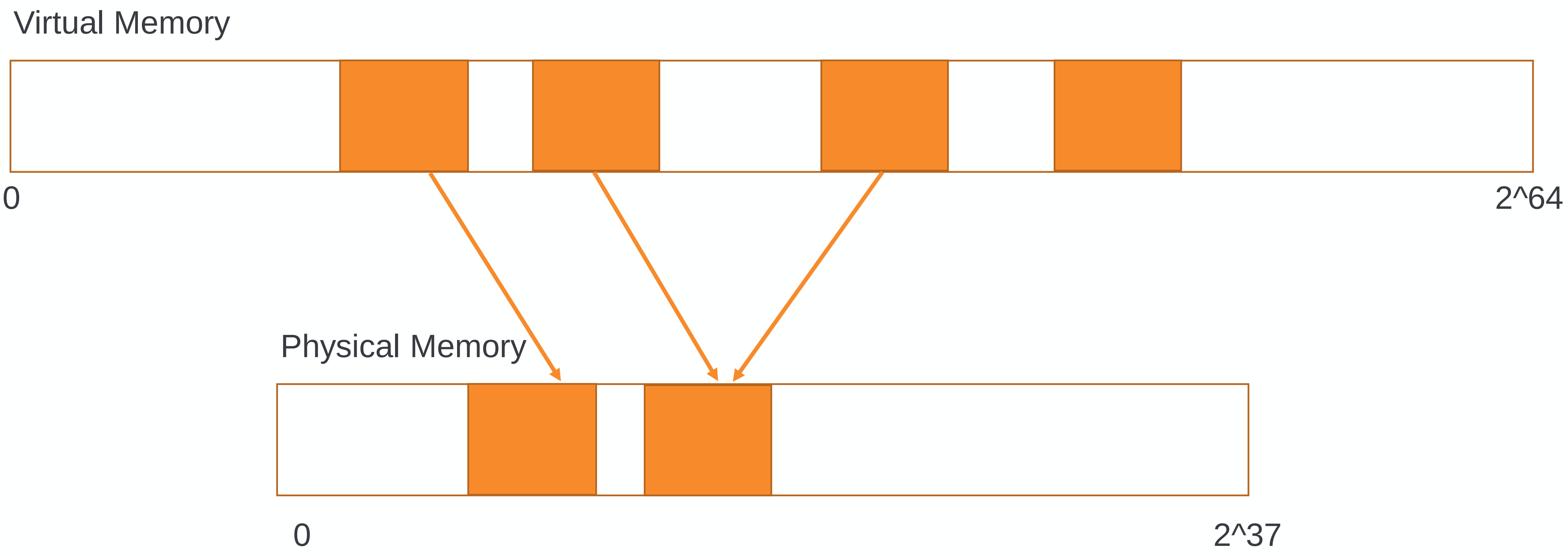
- States of objects stored inside reference address => Colored pointers
  - NMT bit
  - Generation

- Checked against a global expected value during the GC cycle
  - Thread local, almost always L1 cache hits
  - Register

```
test     r9, rax
jne     0x3001443b
mov     r10d, dword ptr [rax + 8]
```

- Relocated: x86 Implementation use trap from VM memory translation Guest/Host
  - Intel EPT
  - AMD NPT

Virtual Memory vs Physical Memory



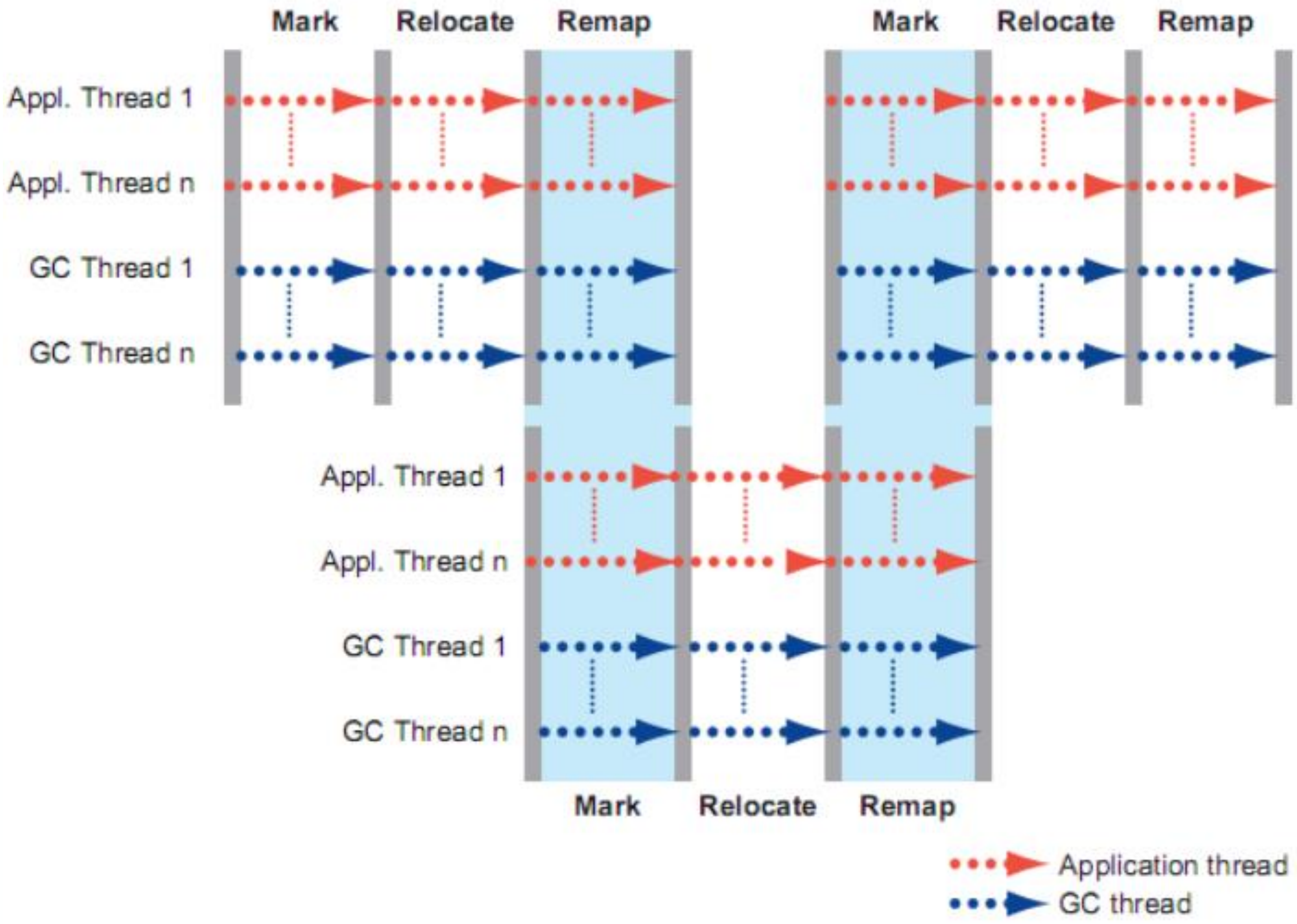
C4 Phases

- All phases are fully parallel & concurrent

- No "rush" to finish phases
  - No constraint about STW pause to be short
  - Physical memory released quickly in relocation phase
  - Can be reused for new allocations
  - Plenty of virtual space vs physical memory

C4 Phases

- Mark
  - Marking all objects in graph
- Relocation
  - Moving objects to release pages
- Remap
  - Fixup references in object graph
  - Folded with next mark cycle

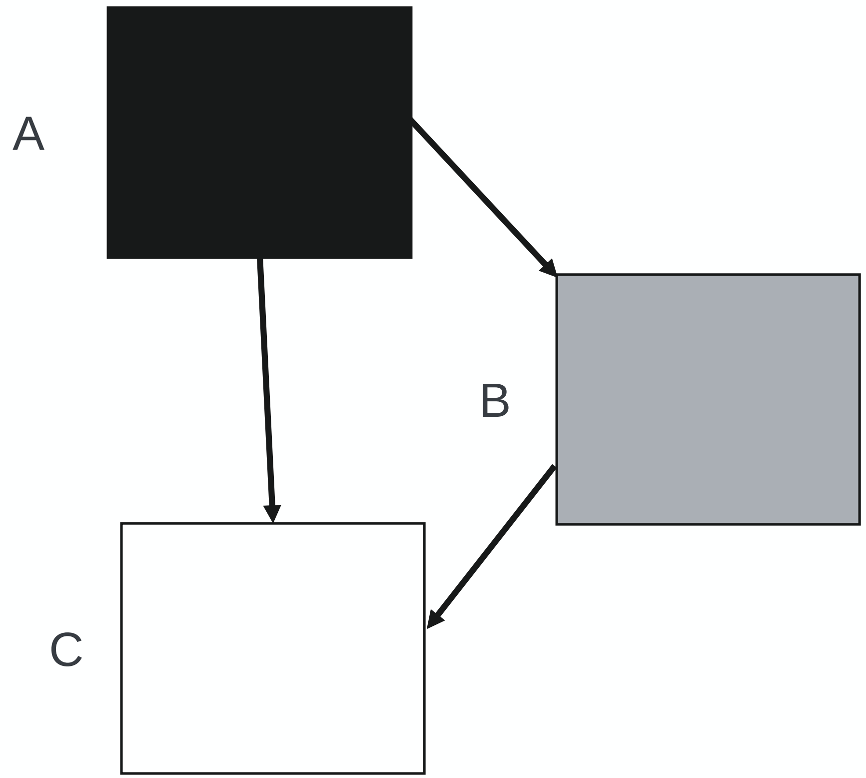


criteo.

Mark Phase

- Incremental Update Marking (vs SATB)
- Single pass
- No final mark/remark
- Self-Healing: Mark object that are not marked for the current cycle

Mark Phase: Concurrent Modification




```
A.field1 = C;
B.field2 = null;
```

Mark Phase

- Scanning roots (Static var, Thread stacks, register, JNI handles)
  - GC threads scans stalled threads
  - Running threads scans their own stack stopping **individually** at Safepoint

- Scanning object graph like a parallel collector

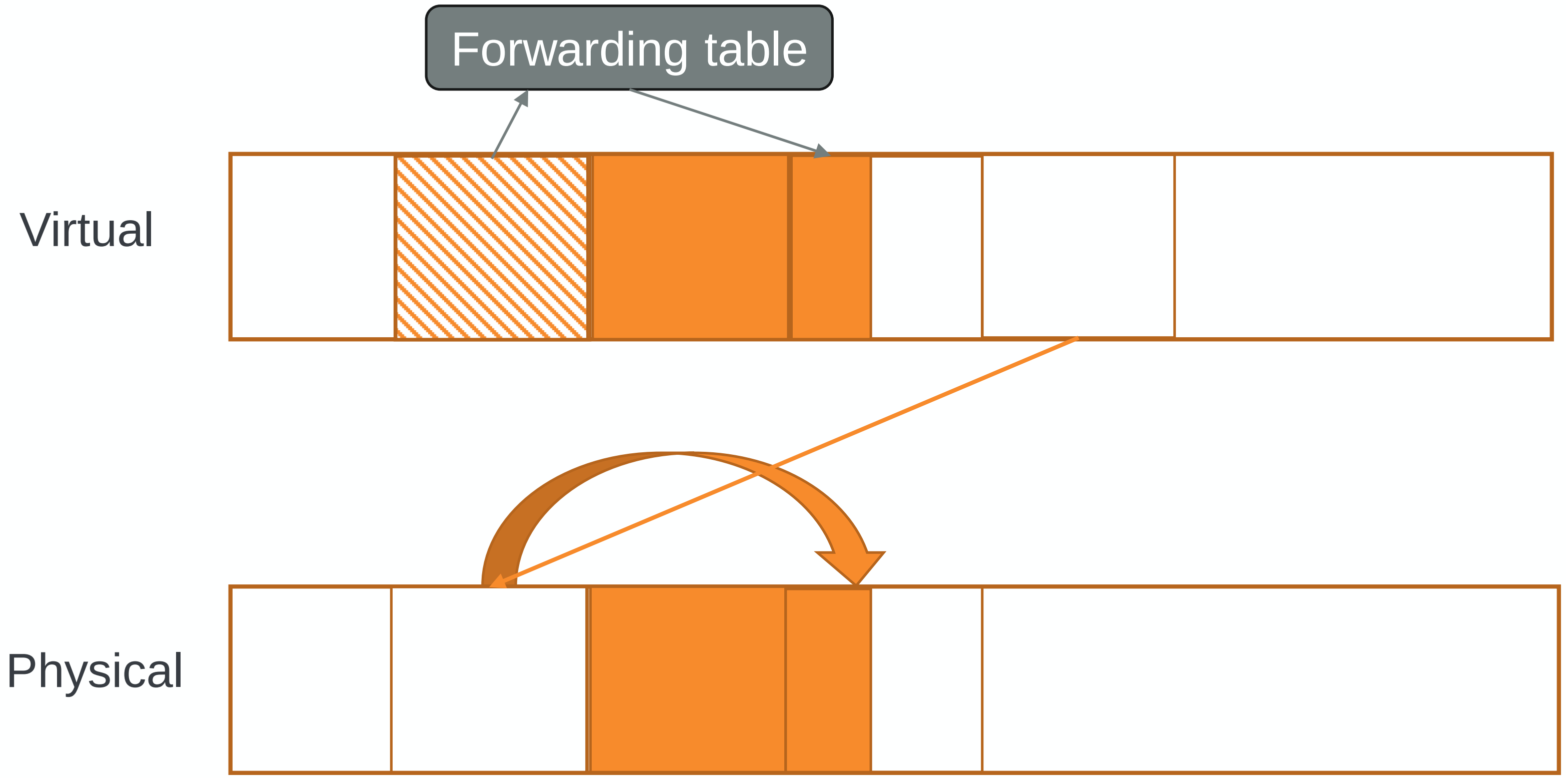
- Newly allocated objects into new pages, not considered for reclaim (relocation)

- For each page, summing live data bytes, used to select page to reclaim

Relocation Phase

- Select pages with the greatest number of dead objects (garbage first!)
- Protect page selected from being accessed by mutators thread
- Move objects to new allocated pages
- Build side arrays (off heap tables) for forwarding information
- Self-Healing: As protected, LVB will trigger a trap to:
  - Copy object to the new location if not done
  - Use forward pointer to fix the reference

Relocation Phase



Relocation Phase

- Few chances mutators stall on accessing a ref as processing mostly dead pages

- Once object copy done, physical memory is released (Quick Release)
  - Can be immediately reused (remapped) to satisfy new allocations

- Pages evacuated are still mapped & protected to help remap phase
  - Cannot be released until all objects are remapped
  - Not a problem as we have a **huge** virtual address space

Remap Phase

- Traverse Object Graph and fixup references
- Execute LVB barrier for each object
- Self-Healing: fixup references using forward information
- As we traverse again, mark for the next phase
  - Mark & Remap phases are folded!

Remap – Kernel module

- Algorithm requires a sustainable rate or remapping operations

- Linux limitations:
    - TLB invalidation
    - Only 4KB pages can be remapped
    - Single threaded remapping (write lock)

- Kernel module implements API for the Zing JVM to increase significantly the remapping rate

- Implements also virtual address aliasing for addressing objects with metadata

Generational

- Young & Old collections done by same algorithm and can be concurrent
- Size of the generation are dynamically adjusted
- Card Marking with write barrier (Stored Value Barrier)
- Old collection is based on young-to-old roots generated by previous young cycle
- Young collection will perform card scanning per page
  - hold an eventual concurrent Old collection per page scanned

C4 @ Criteo

- Used by Hadoop Name Node
- 580GB Heap
- Very hard to tune with G1
- No issue so far regarding GC since production roll out (Oct 2017)

Z GC

Z GC

- Non generational
- Region based (zPages, dynamically sized)
- Concurrent Marking, Compaction, Ref processing
- Use Colored Pointers & Read/Load Barrier

```
mov     r10,QWORD PTR [r11+0xb0]
test    QWORD PTR [r15+0x20],r10
jne     0x00007f9594cc54b5
```

- Self-Healing
  - Cooperation between mutator threads & GC threads
- Experimental in JDK 11 (`-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`)

criteo

Z GC

Cliff Click
@cliff_click

Abonné

Ok, quick read of ZGC and it basically screams "GPGC done by Oracle". Can anybody closer to the truth fill in any details?

Traduire le Tweet

22:41 - 27 sept. 2018

5 Retweets **12** J'aime

2 5 12

criteo

Z GC phases:

- Initial Mark (STW)
- Concurrent Mark/Remap
- Final Mark (STW)
- Concurrent Prepare for Relocation
- Start Relocate (STW)
- Concurrent Relocate

Colored Pointers

- Store metadata in unused bits of reference address

- 42 bits for addressing (4TB)

- 4 bits for metadata
  - Marked0
  - Marked1
  - Remapped
  - Finalizable

```
 6                 4 4 4  4 4                                             0
 3                 7 6 5  2 1                                             0
+-------------------+-+----+-----------------------------------------------+
|00000000 00000000 0|0|1111|11 11111111 11111111 11111111 11111111 11111111|
+-------------------+-+----+-----------------------------------------------+
|                   | |    |
|                   | |    * 41-0 Object Offset (42-bits, 4TB address space)
|                   | |
|                   | * 45-42 Metadata Bits (4-bits)  0001 = Marked0
|                   |                                 0010 = Marked1
|                   |                                 0100 = Remapped
|                   |                                 1000 = Finalizable
|                   |
|                   * 46-46 Unused (1-bit, always zero)
|
* 63-47 Fixed (17-bits, always zero)
```


criteo

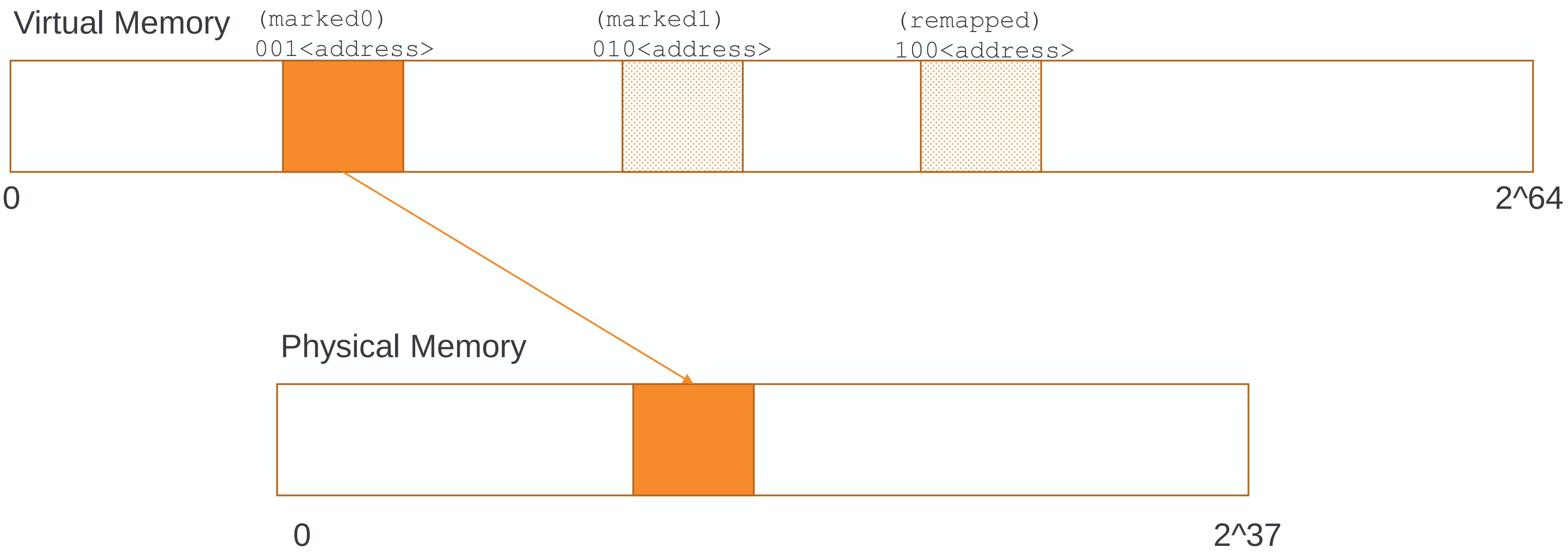
Multi-Mapping

- Colored pointers needs to be unmasked for dereferencing
  - Some HW support masking (SPARC, Aarch64))
  - On linux/windows, overhead if done with classical instructions

- Only one view is active at any point

- Plenty of Virtual Space

Multi-Mapping



Page Allocations

- Pages are multiple of 2MB

- 3 different groups
    - Small: 2MB pages with object size <= 256KB
    - Medium: 32MB pages with object size <= 4MB
    - Large: 2MB pages, objects span over multiple of them

- Objects in Large group are meant to not to be relocated (too expensive)

criteo

Difference between C4 & Z GC

- Handling remapping
  - C4: Memory protection + trap
  - Z: mask in colored pointer

- Unmasking ref addresses
  - C4: Kernel module aliasing
  - Z: Multi-mapping or HW support

- Pages & Relocation
  - C4:
    - Page are fixed to match OS size (mem protection)
    - relocation for large objects by remapping
  - Z:
    - zPages are dynamic, a zPage can be 100MB large
    - No relocation for large objects

How to choose a GC algorithm

Throughput vs Latency

- Case 1:
  - Need maximum of work done in a time frame (offline job)
  - Can afford FullGC of several seconds

$\Rightarrow$ Use a throughput collector like ParalleGC or G1

- Case 2:
  - Have time constraint per unit of work (online job)
  - Cannot afford FullGC of several seconds

$\Rightarrow$ Use a low latency collector like C4, Shenandoah or Z

Low latency GCs

- You have to run on Windows
  - Shenandoah

- Battlefield tested GC (maturity)
  - C4
  - Shenandoah

- Minimizing any kind of JVM pauses
  - C4
  - Z

- You don’t want pay for it:
  - Shenandoah
  - Z

criteo.

References

References GC Basics

- Java Garbage Collection distilled by Martin Thompson
- The Java GC mini book
- Oracle's white paper on JVM memory management & GC
- What differences JVM makes by Nitsan Wakart
- Memory Management Reference
- IBM Pause-Less GC

References G1

- Garbage-First Garbage Collection (2004)
- G1 One Garbage Collector to rule them all by Monica Beckwith
- Tips for Tuning The G1 GC by Monica Beckwith
- G1 Garbage Collector Details and Tuning by Simone Bordet
- Write Barriers in Garbage-First Garbage Collector by Monica Beckwith

References Shenandoah

- Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK
- Shenandoah: The Garbage Collector That Could by Aleksey Shipilev
- Shenandoah GC Wiki

References C4

- The Pauseless GC algorithm (2005)
- C4: Continuously Concurrent Compacting Collector (2011)
- Azul GC in Detail by Charles Humble
- 2010 version source code

References ZGC

- ZGC - Low Latency GC for OpenJDK by Per Liden
- Java's new Z Garbage Collector (ZGC) is very exciting by Richard Warburton
- A first look into ZGC by Dominik Inführ
- Architectural Comparison with C4/Pauseless

Thank You!

@jpbempel

criteo