The Z Garbage Collector

An Introduction

Per Lidén & **Stefan Karlsson**
HotSpot Garbage Collection Team
FOSDEM 2018

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

Agenda

A Scalable Low Latency Garbage Collector

Goals

TB Multi-terabyte heaps

10ms Max GC pause time

Lay the foundation for future GC features

15% Max application throughput reduction

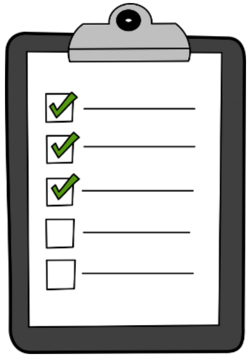
GC pause times **do not** increase with heap or live-set size

At a Glance

- New garbage collector
- Load barriers
- Colored pointers
- Single generation
- Partial compaction
- Region-based
- Immediate memory reuse
- NUMA-aware

- Concurrent
  ✓ Marking
  ✓ Relocation/Compaction
  ✓ Relocation Set Selection
  ✓ Reference Processing
  ✓ JNI WeakRefs Cleaning
  - StringTable/SymbolTable Cleaning
  - Class Unloading

Current Status



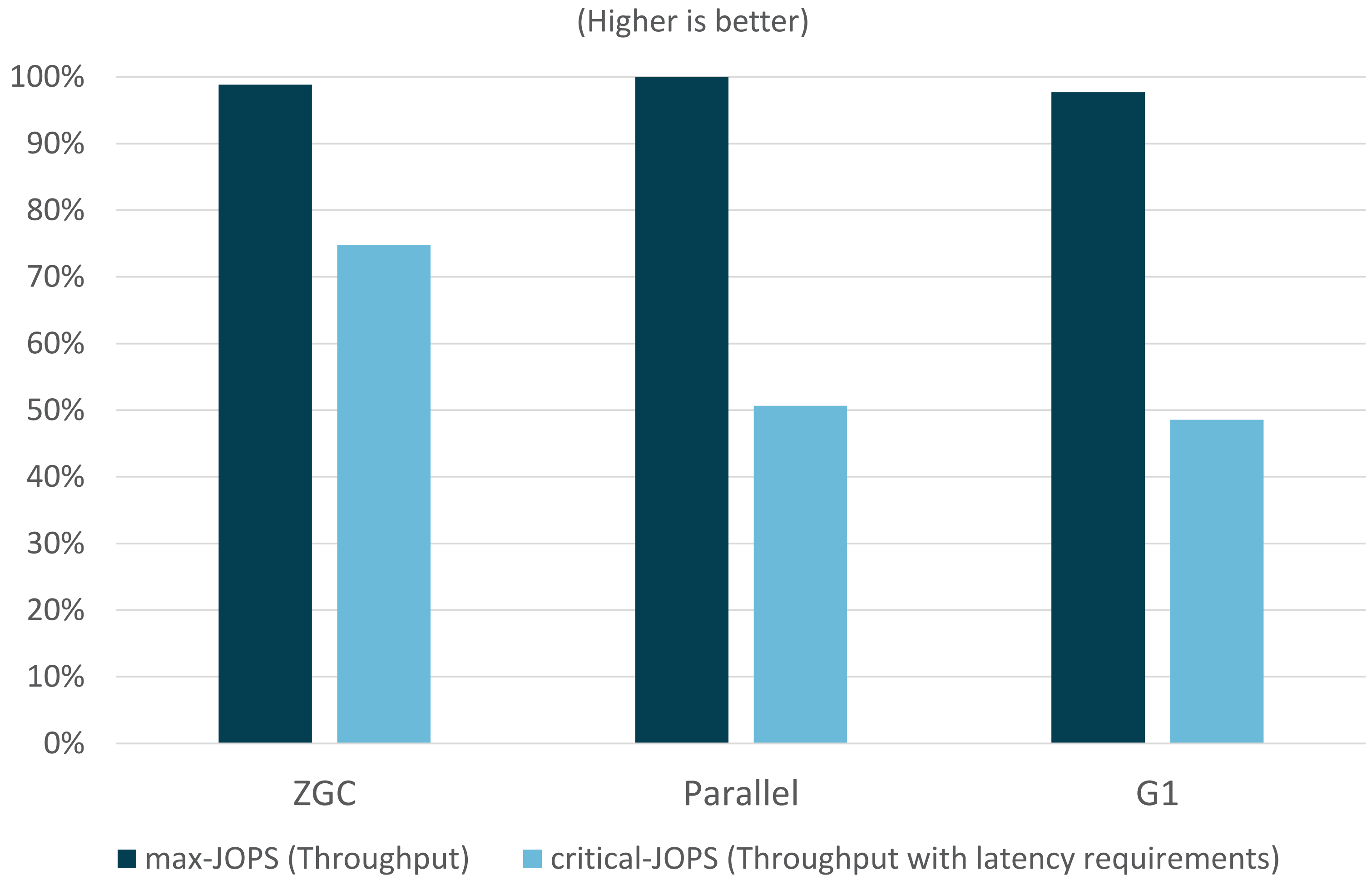
- Design and implementation approaching mature and stable

- Main focus on **Linux/x86_64**
  - Other platforms can be added if there's enough demand

- Performance looks very good
  - Both in terms of latency and throughput

Agenda

SPECjbb®2015 – Score



Mode: Composite

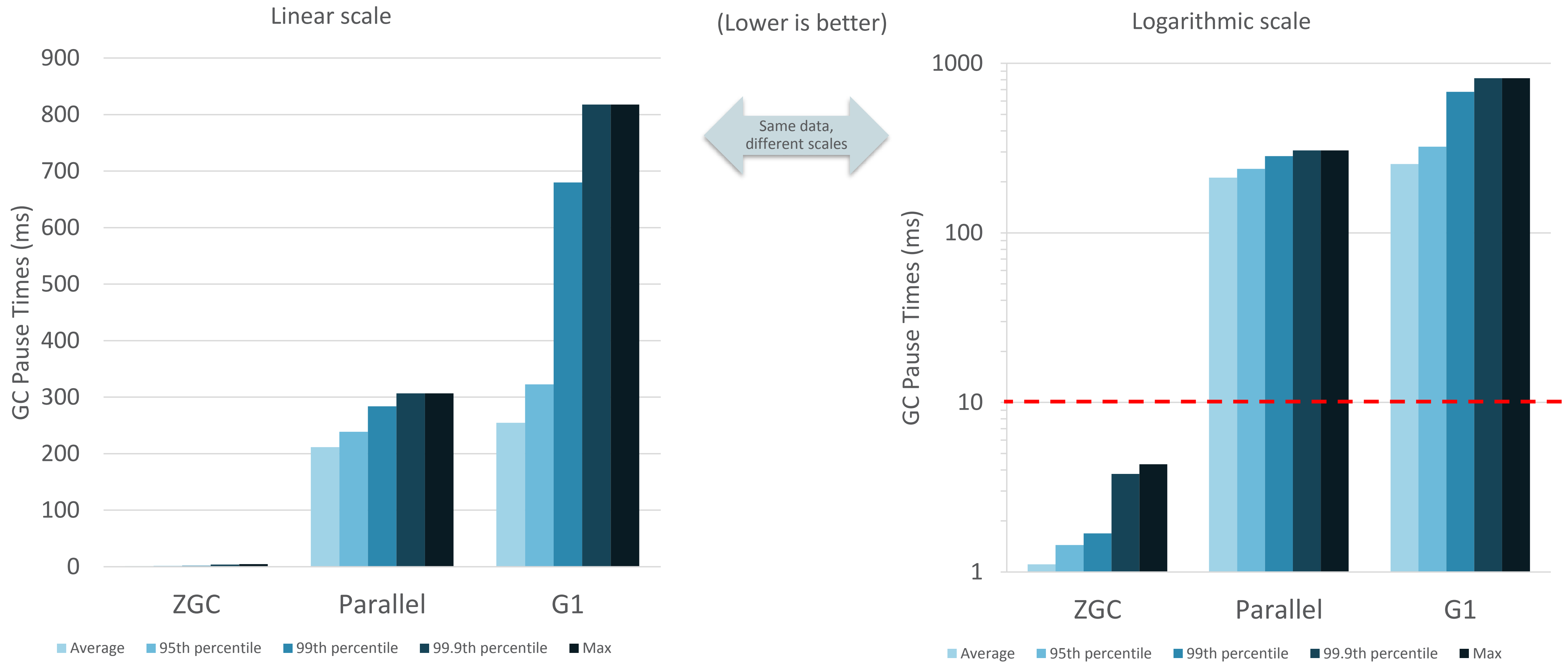
Heap Size: 128G

OS: Oracle Linux 7.4

HW: Intel Xeon E5-2690 2.9GHz
2 sockets, 16 cores (32 hw-threads)

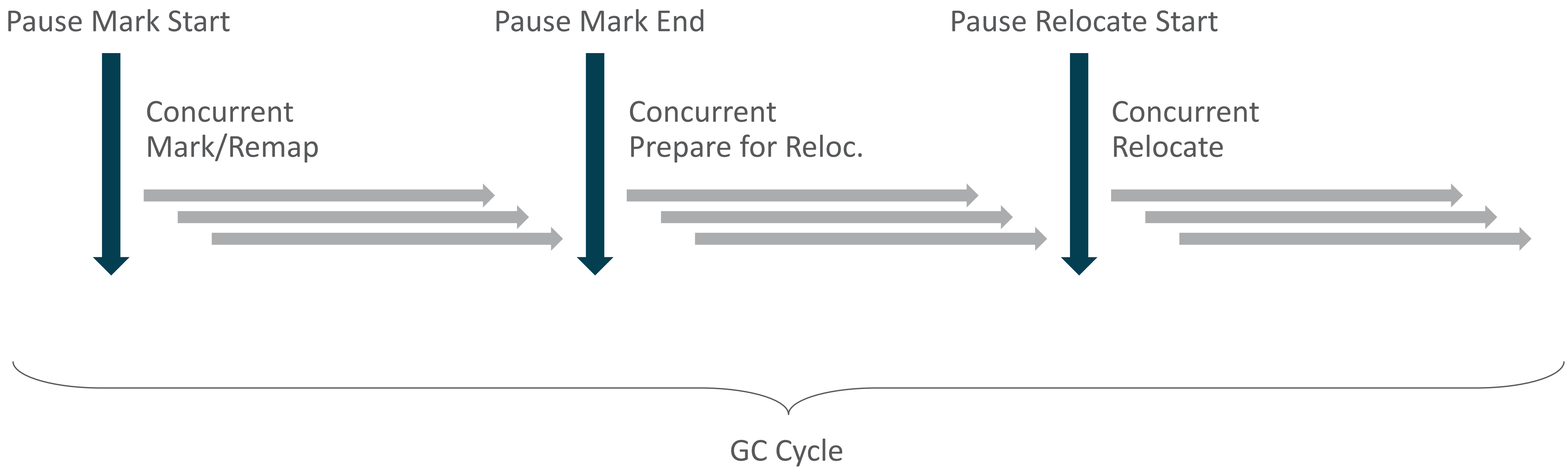
SPECjbb®2015 is a registered trademark of the Standard Performance Evaluation Corporation (spec.org). The actual results are not represented as compliant because the SUT may not meet SPEC's requirements for general availability.

SPECjbb®2015 – Pause Times

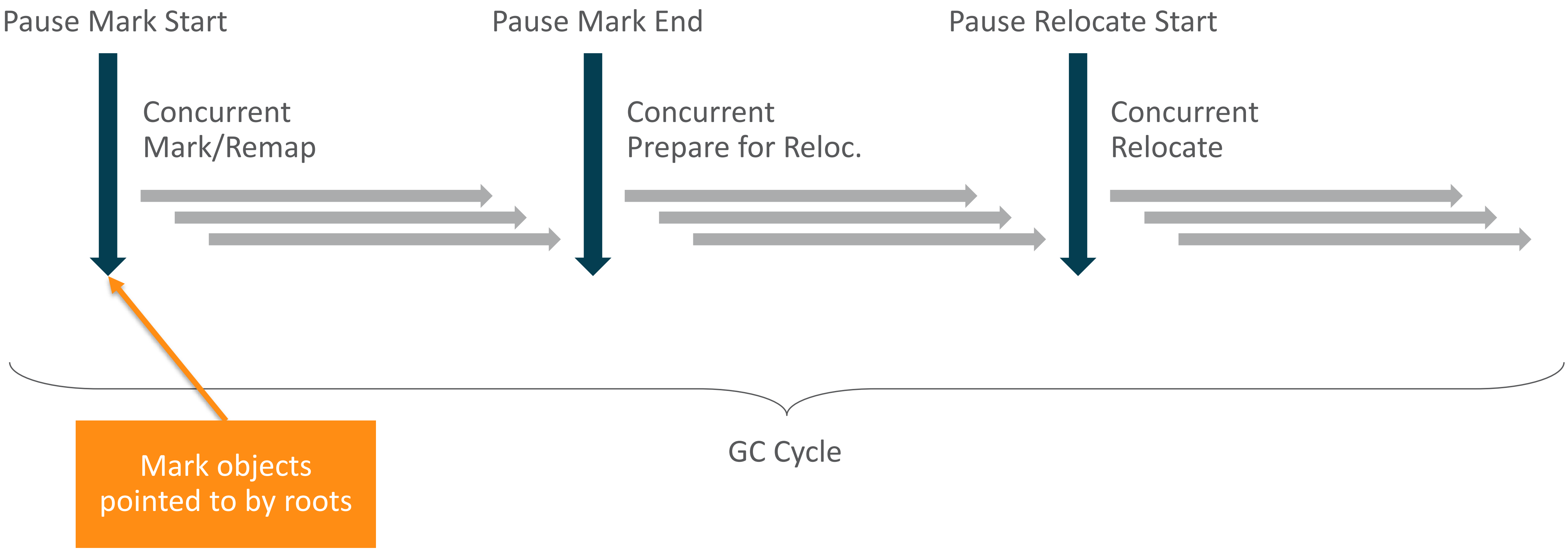


Agenda

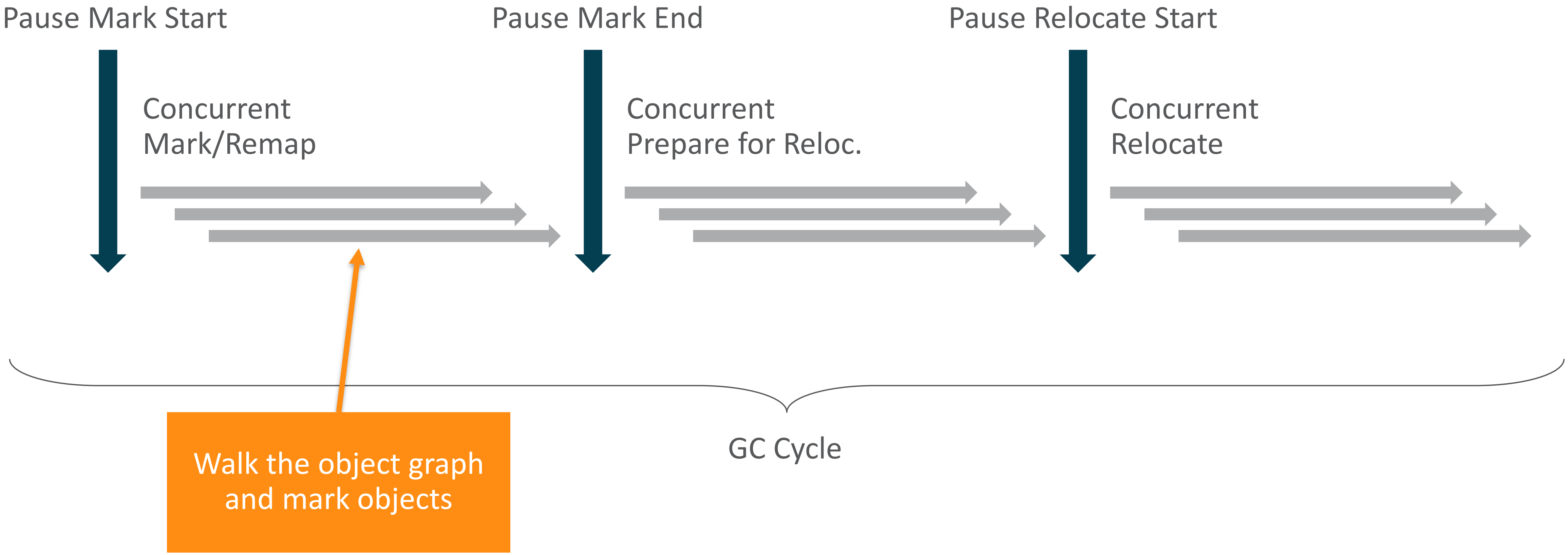
ZGC Phases


Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent Mark/Remap
Concurrent Prepare for Reloc.
Concurrent Relocate
GC Cycle

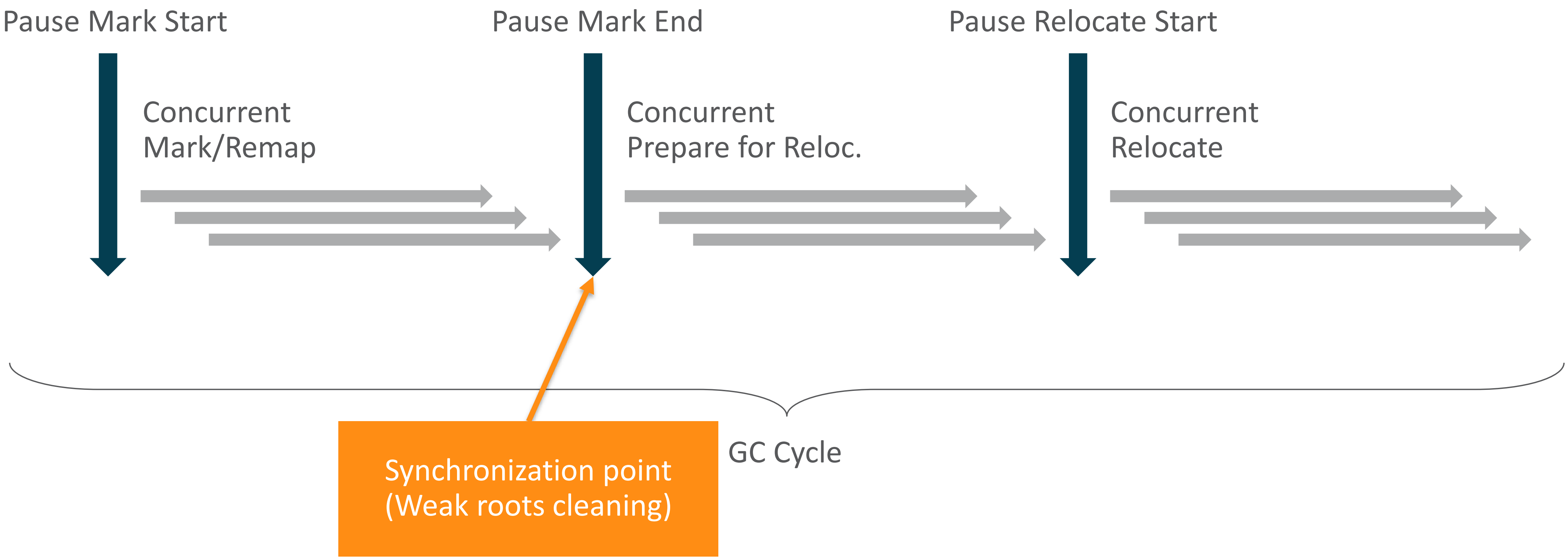
ZGC Phases


Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent Mark/Remap
Concurrent Prepare for Reloc.
Concurrent Relocate
GC Cycle
Mark objects pointed to by roots

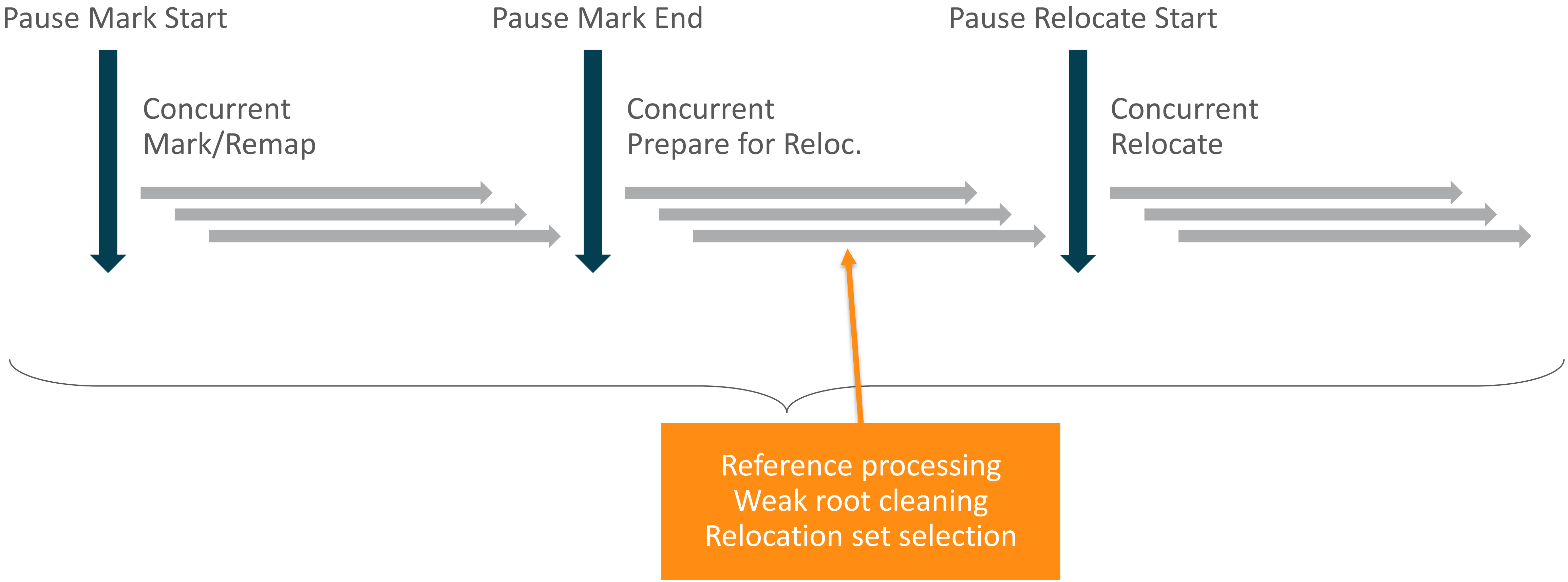
ZGC Phases


Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent Mark/Remap
Concurrent Prepare for Reloc.
Concurrent Relocate
GC Cycle
Walk the object graph and mark objects

ZGC Phases

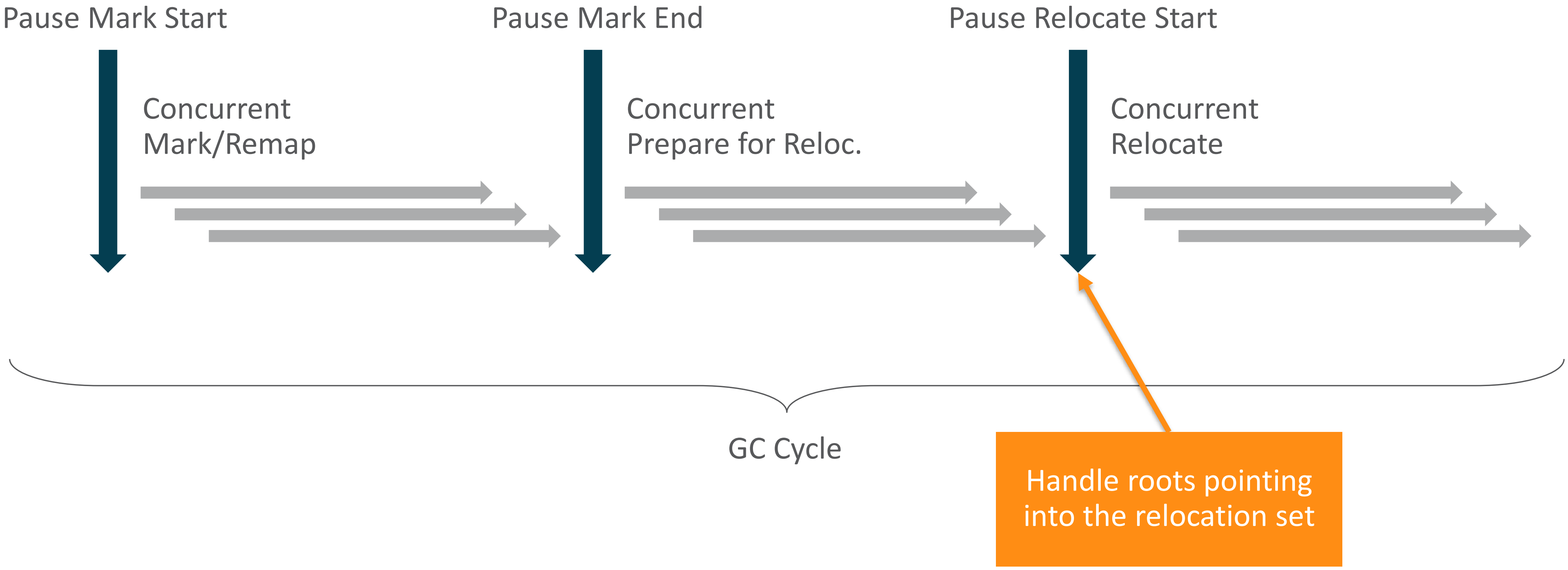

Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent
Mark/Remap
Concurrent
Prepare for Reloc.
Concurrent
Relocate
GC Cycle
Synchronization point
(Weak roots cleaning)

ZGC Phases

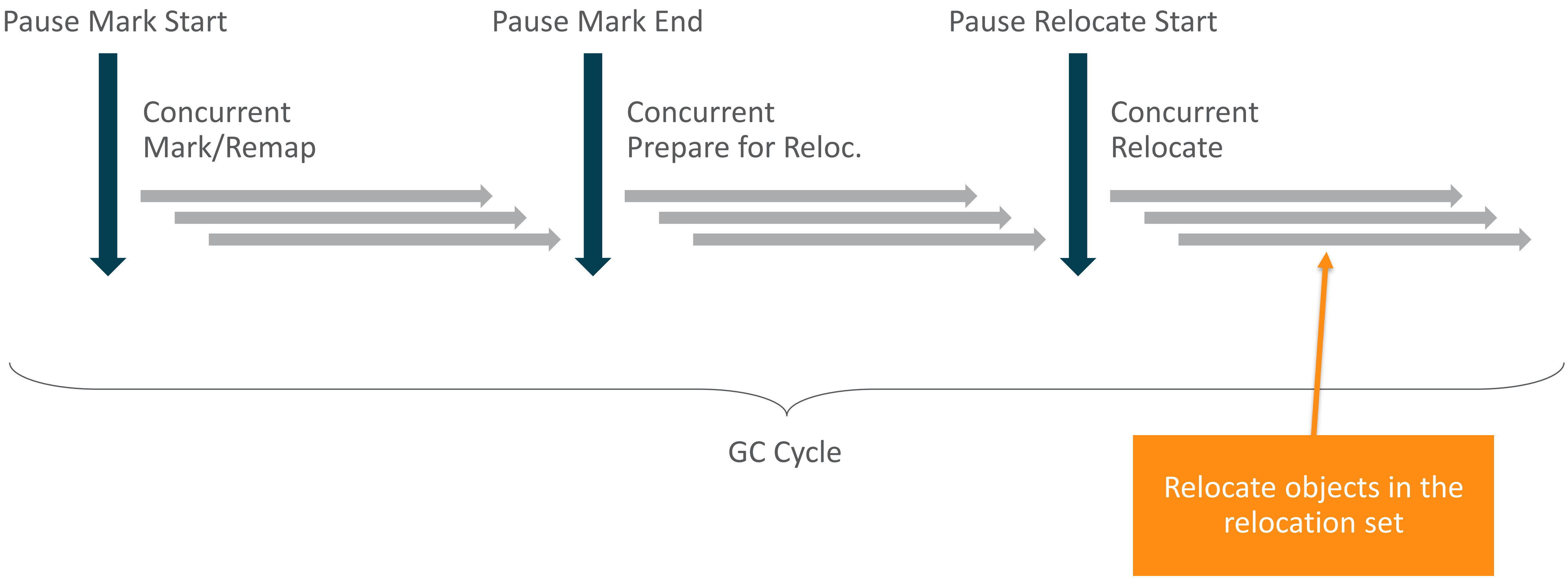


Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent
Mark/Remap
Concurrent
Prepare for Reloc.
Concurrent
Relocate
Reference processing
Weak root cleaning
Relocation set selection

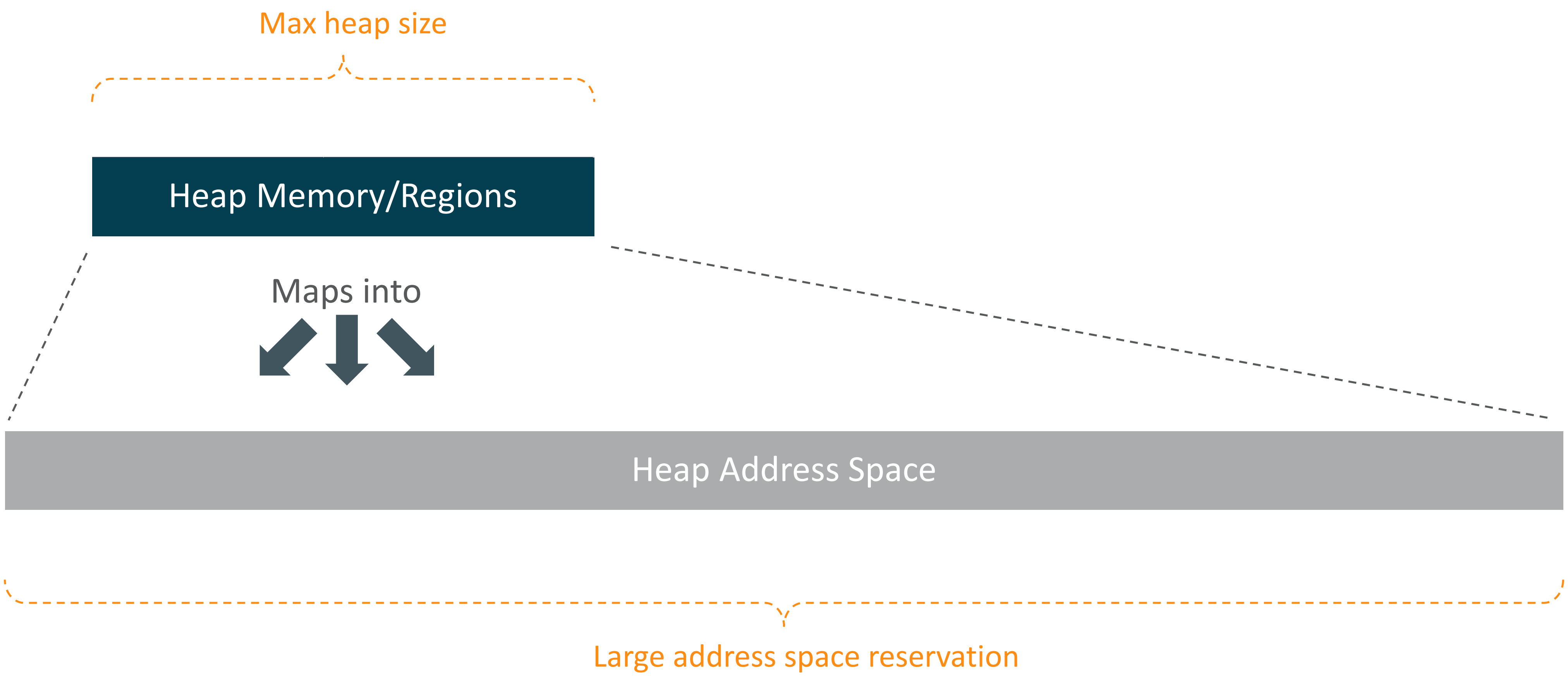
ZGC Phases


Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent
Mark/Remap
Concurrent
Prepare for Reloc.
Concurrent
Relocate
GC Cycle
Handle roots pointing
into the relocation set

ZGC Phases


Pause Mark Start
Pause Mark End
Pause Relocate Start
Concurrent Mark/Remap
Concurrent Prepare for Reloc.
Concurrent Relocate
GC Cycle
Relocate objects in the relocation set

Heap Address Space

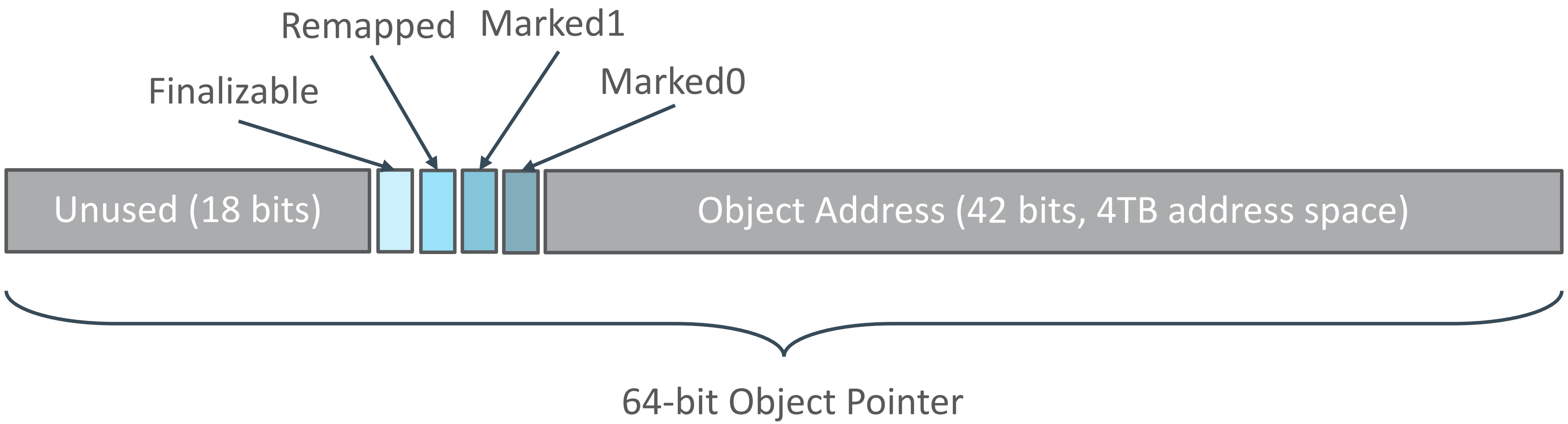


Colored Pointers

- Core design concept in ZGC
- **Metadata** stored in unused bits in 64-bit pointers
  - No support for 32-bit platforms
  - No support for CompressedOops
- **Virtual Address-masking** either in hardware, OS or software
  - Heap multi-mapping on Linux/x86_64
  - Supported in hardware on Solaris/SPARC

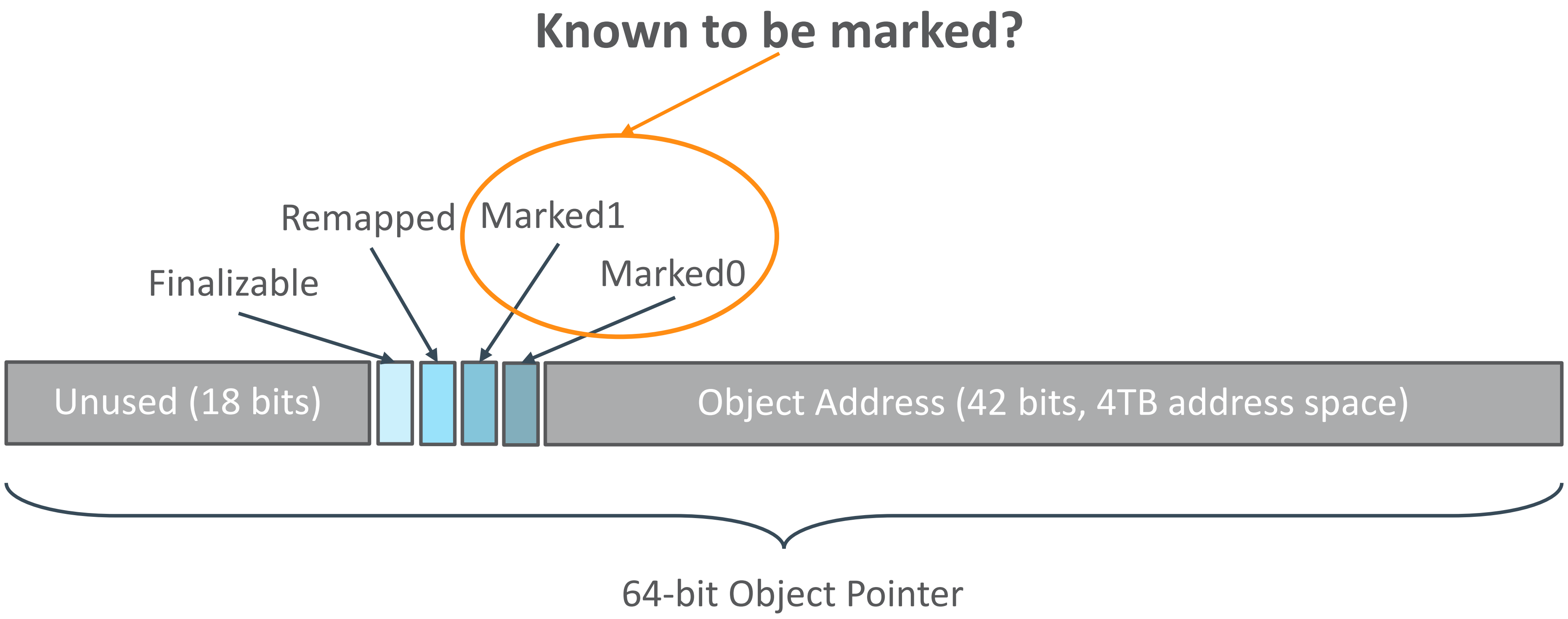
Colored Pointers

Layout on x86_64



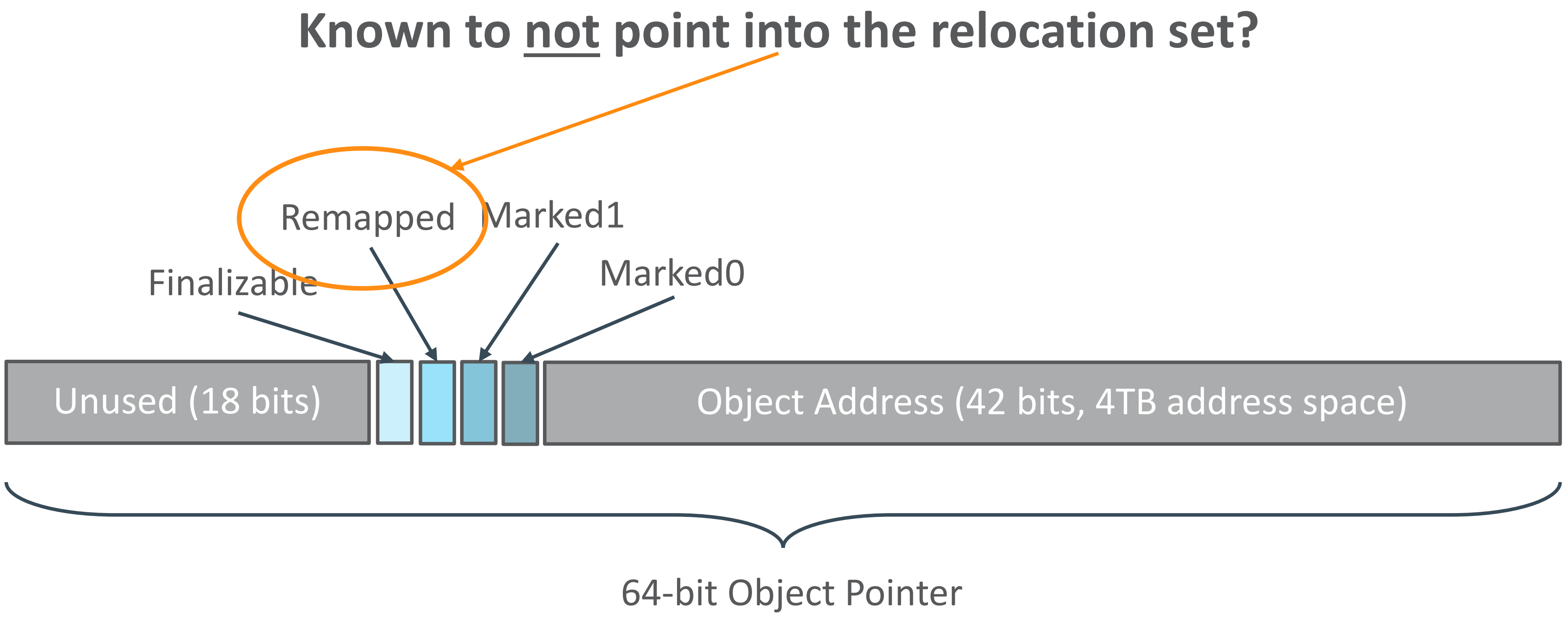
Colored Pointers

Layout on x86_64



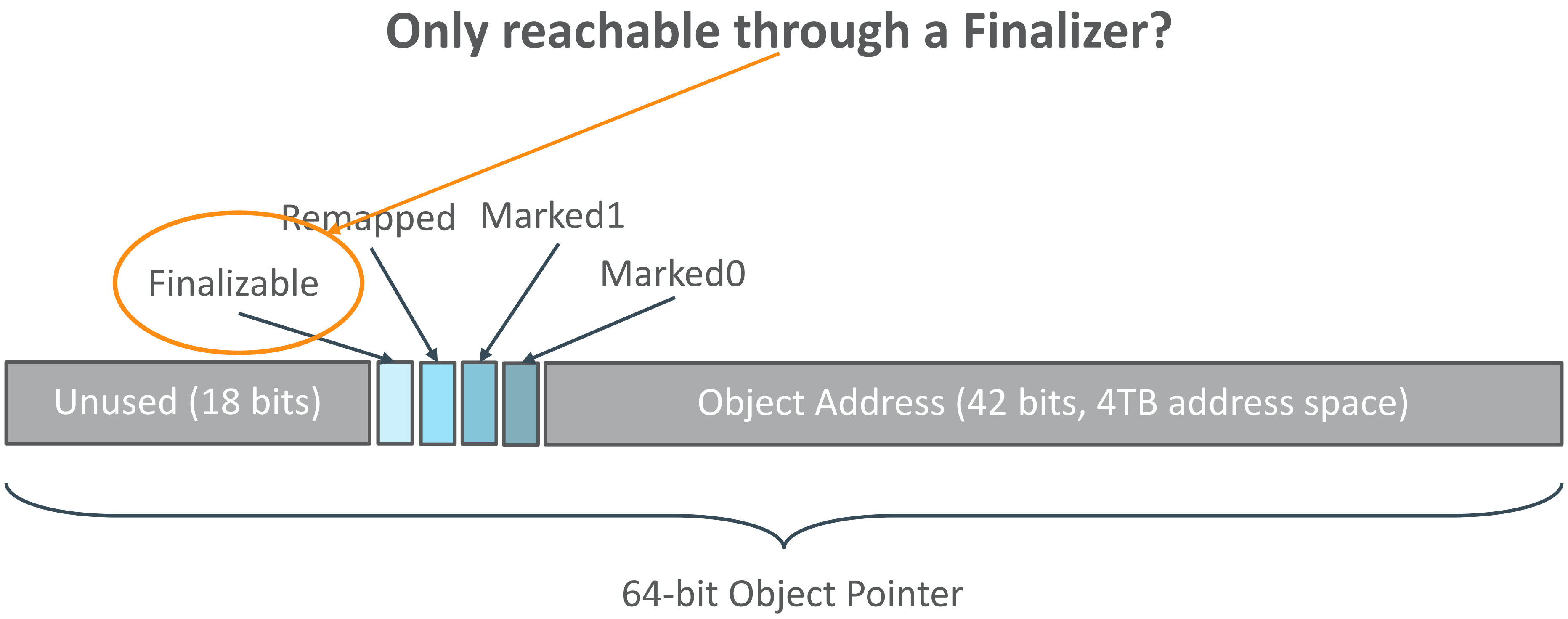
Colored Pointers

Layout on x86_64



Colored Pointers

Layout on x86_64



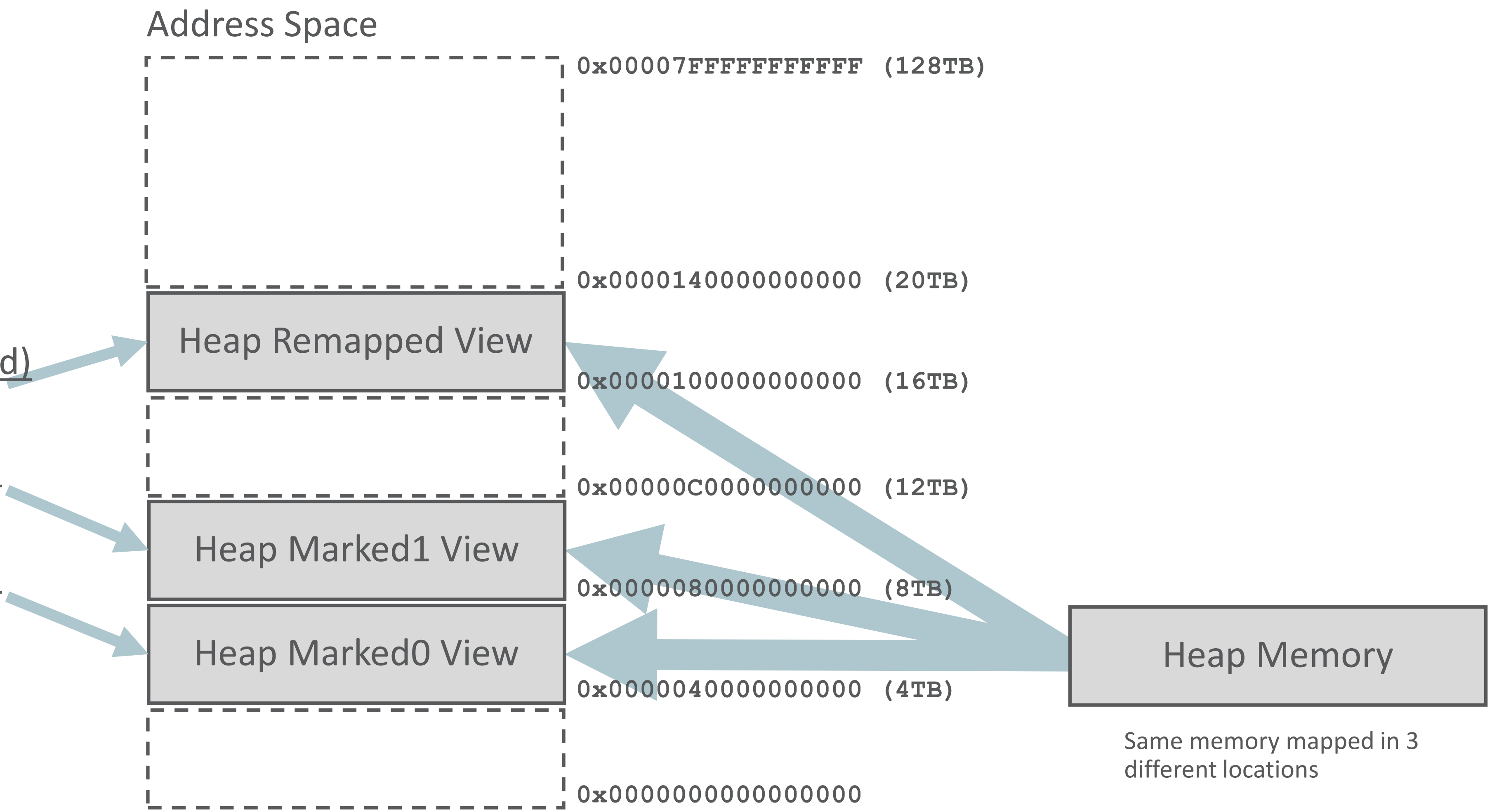
Heap Multi-Mapping on Linux/x86_64

Colorless pointer
0x0000000012345678

Colored pointer (Remapped)
0x0000100012345678

Colored pointer (Marked1)
0x0000080012345678

Colored pointer (Marked0)
0x0000040012345678



Load Barrier

- Applied when **loading an object reference** from the heap
  - **Not** when later using that reference to access the object
  - Conceptually similar to the decoding of compressed oops
- Looks at the color of the pointer
  - Take action if the pointer has a "**bad**" color (mark/relocate/remap)
  - Change to the "**good**" color (repair/heal)
- Optimized for the common case
  - Most object references will have the "**good**" color

Load Barrier

```
Object o = obj.fieldA;          // Loading an object reference from heap
```

Load Barrier

```
Object o = obj.fieldA;          // Loading an object reference from heap
<load barrier needed here>
```

Load Barrier

```
Object o = obj.fieldA;          // Loading an object reference from heap
<load barrier needed here>
Object p = o;                   // No barrier, not a load from heap
o.doSomething();                // No barrier, not a load from heap
int i = obj.fieldB;             // No barrier, not an object reference
```

Load Barrier

```
Object o = obj.fieldA;          // Loading an object reference from heap
<load barrier needed here>
```

Load Barrier

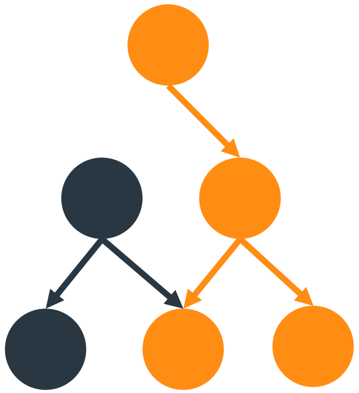
```
mov     0x20(%rax), %rbx          // Object o = obj.fieldA;
test    %rbx, (0x16)%r15          // Bad color?
jnz     slow_path                 // Yes -> Enter slow path and
                                  // mark/relocate/remap, adjust
                                  // 0x20(%rax) and %rbx
```

Load Barrier

```
mov     0x20(%rax), %rbx           // Object o = obj.fieldA;
test    %rbx, (0x16)%r15           // Bad color?
jnz     slow_path                  // Yes -> Enter slow path and
                                   // mark/relocate/remap, adjust
                                   // 0x20(%rax) and %rbx
```

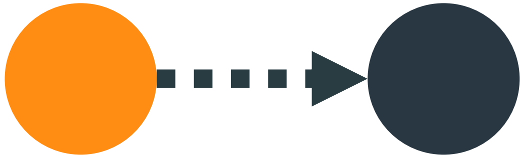
~4% execution overhead on **SPECjbb®2015**

Mark



- Concurrent & Parallel
- Load barrier
  – Detects loads of non-marked object pointers
- Finalizable mark
  – Enabler for Concurrent Reference Processing
- Thread local handshakes
  – Used to synchronize end of concurrent mark

Relocation



- Concurrent & Parallel
- Load barrier
  - Detects loads of object pointers pointing into the relocation set
  - Java threads help out with relocation if needed
- Off-heap forwarding tables
  - No forwarding information stored in old copies of objects
  - Important for immediate reuse of heap memory

Agenda

1. What is ZGC?
2. Some Numbers
3. A Peek Under The Hood
4. Going Forward
5. How To Get Started

In The Works

- GC Barrier API
  - Make it **easier** to plug in new GCs (ZGC, Shenandoah, Epsilon)
- Concurrent class unloading & weak roots
  - Traditionally done in a Stop-The-World pause
  - Impacts **JITs** and **Runtime** subsystems
- Addressing non-GC induced latencies
  - Time to safepoint/unsafepoint, object monitor deflation, etc.

Foundation for Future GC Features

Colored Pointers + Load Barriers

- Thread local GC scheme
- Track heap access patterns
- Use non-volatile memory for rarely used parts of the heap
- Compress or archive parts of the heap
- Object properties encoded in pointers
- Allocation tricks
- etc.

Agenda

1. What is ZGC?
2. Some Numbers
3. A Peek Under The Hood
4. Going Forward
5. **How To Get Started**

How To Get Started

Download

- Official **early access builds** will be available soon-ish, but until then…
- Download & build

```
$ hg clone http://hg.openjdk.java.net/zgc/zgc
$ cd zgc
$ sh configure
$ make images
```

- Run

```
$ ./build/linux-x86_64-<...>/images/jdk/bin/java
```

How To Get Started

JVM Options

- Enable ZGC: **`-XX:+UseZGC`**
- Tuning
  - If you care about latency, do **not** overprovision your machine
  - Max heap size: **`-Xmx<size>`**
  - Number of concurrent GC threads: **`-XX:ConcGCThreads=<number>`**
- Logging
  - Basic logging: **`-Xlog:gc`**
  - Detailed logging useful when tuning: **`-Xlog:gc*`**

Feedback Welcome!

http://openjdk.java.net/projects/zgc/

OpenJDK