Adventures in Concurrent Garbage Collection

Partially based on slides or graphs by
Erik Österlund, Albert Mingkun Yang, Jonas Norlinder

Tobias Wrigstad at PLISS/FOJW 2023

The Concurrent Z Garbage Collector (ZGC)

In OpenJDK since 11 (experimental); 15 (prod.); 21 (generational)

- **Goals**: tail-latency, TB size heaps, <1ms pause times, no need for tuning
- Allows GC to run concurrent with mutators
  - Global STW pauses whose lengths are invariant of heap size
- Concurrent marking (as explained to us by Tony yesterday)
- Concurrent compaction (as to be explained to us by Tony (and myself))
- Concurrent weak reference processing (ask Richard during coffee)
- Developed by Oracle (Per Lidén, Stefan Karlsson, Erik Österlund, et al.)

Overview of Single-Generation ZGC

Mark + Remap
Reference Processing
EC Selection
Relocation
t

Mark all live objects

Fix all broken pointers

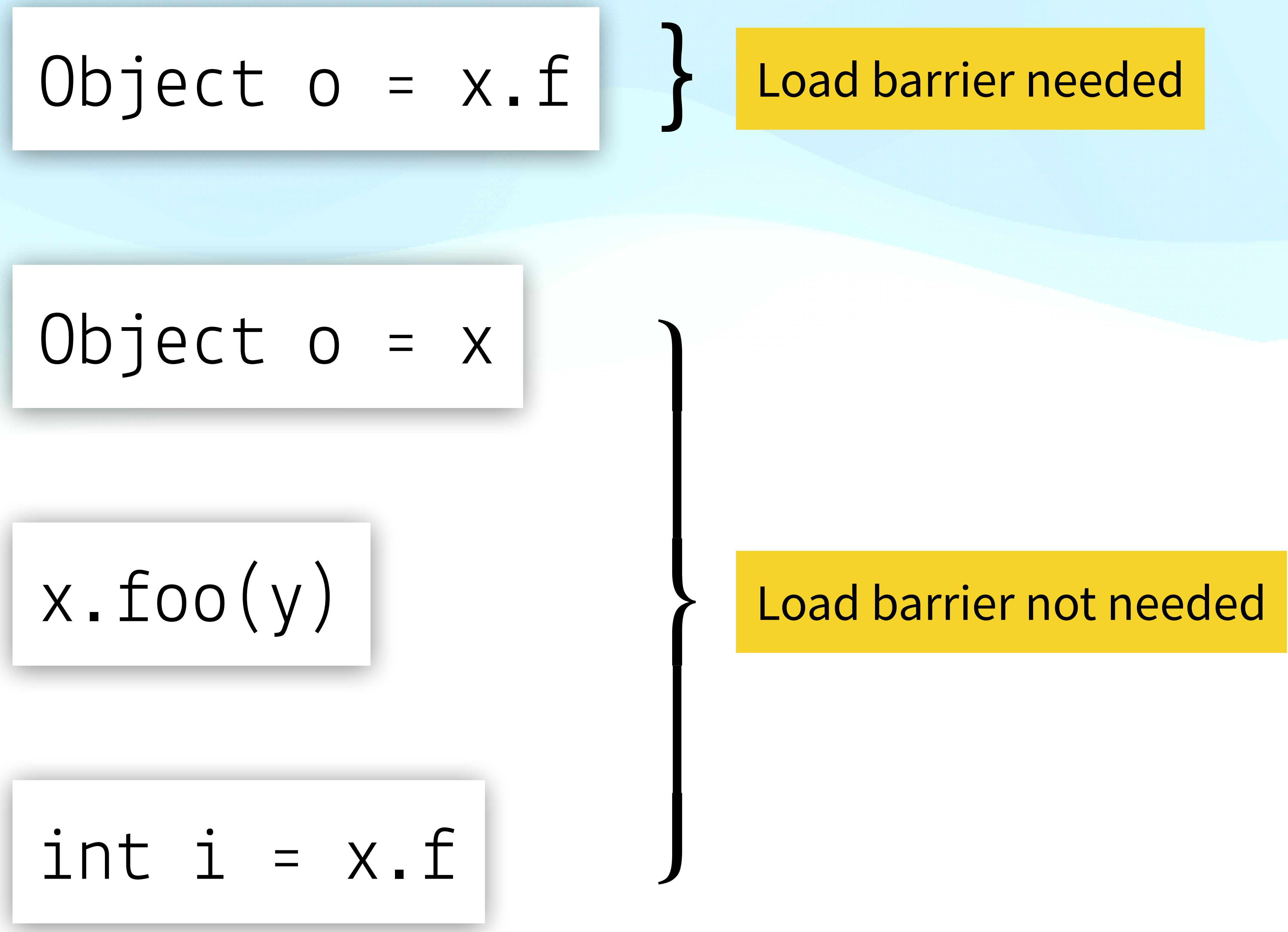
Deal with weak references
(Ignored in this talk)

Based on liveness,
select pages for
evacuation

Perform evacuation

Quick Load-Barrier Primer particular to ZGC

- ZGC is a concurrent GC in OpenJDK
  - GC threads are free to move objects around
  - **Program threads discover this in load barriers**
  - Load barrier slow paths heal dangling pointers
  - Load-barrier overhead ~2–3%



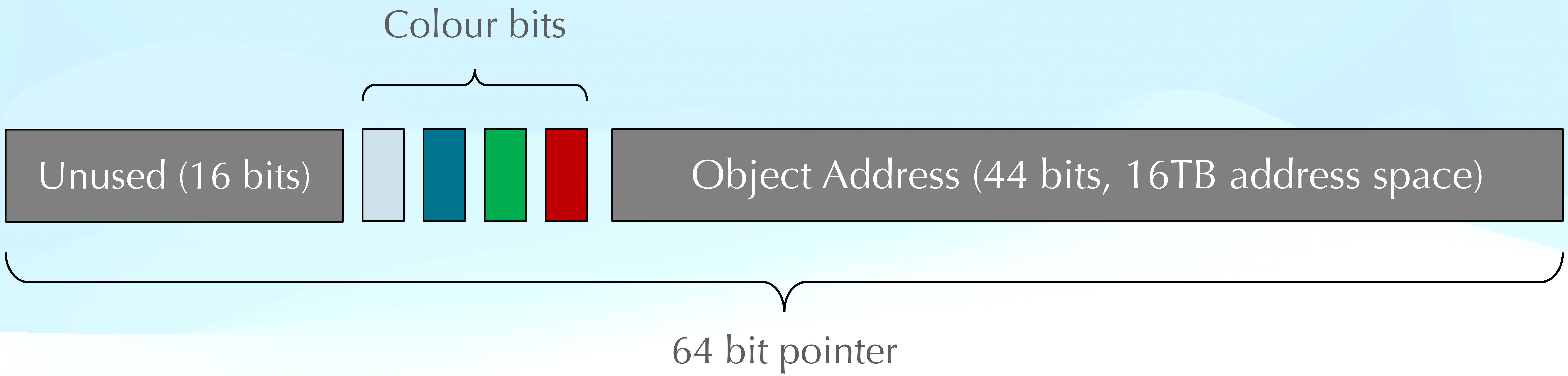
```
Object o = x.f
```

Load barrier needed

```
Object o = x
x.foo(y)
int i = x.f
```

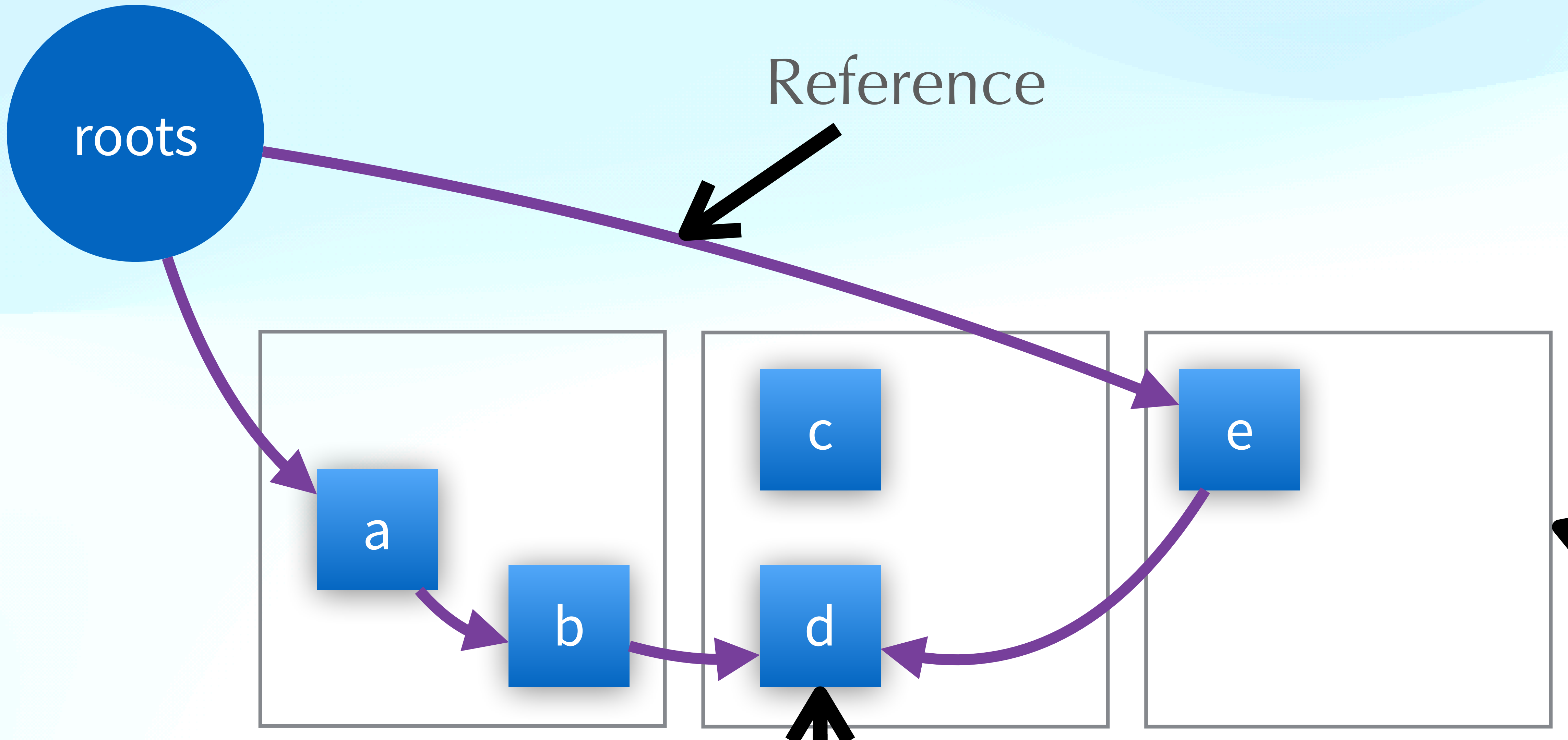
Load barrier not needed

Anatomy of a pointer in ZGC



- Use a load barrier when storing addresses from the heap to the stack
- Good colour — fast path, do nothing
- Bad colour — slow path, do something

Example of ZGC Cycle



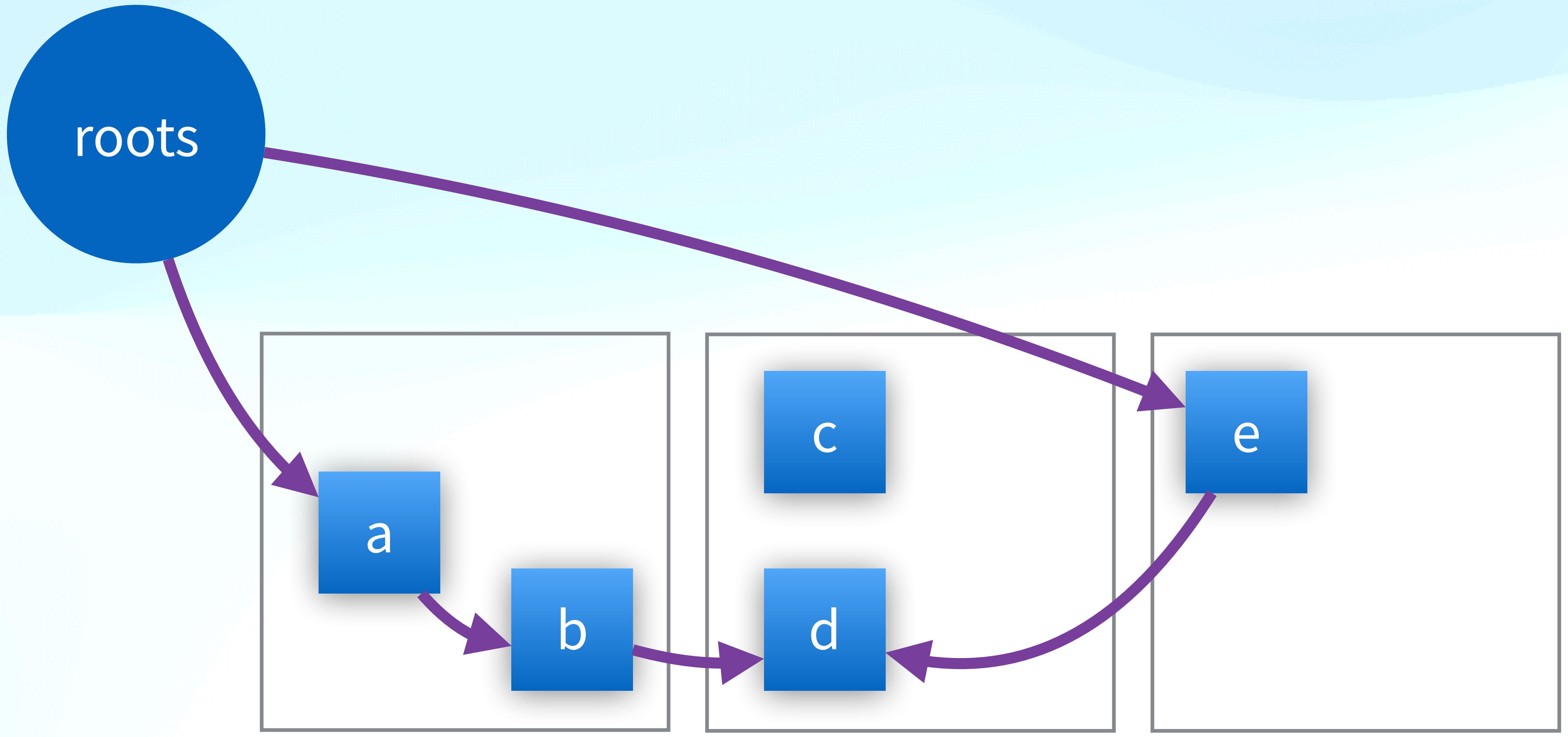
Reference

Object

Memory page (consecutive memory managed together)

Cycle 1

Initial state of the heap



roots

a

b

c

d

e

Mark + Remap

Reference Processing

EC Selection

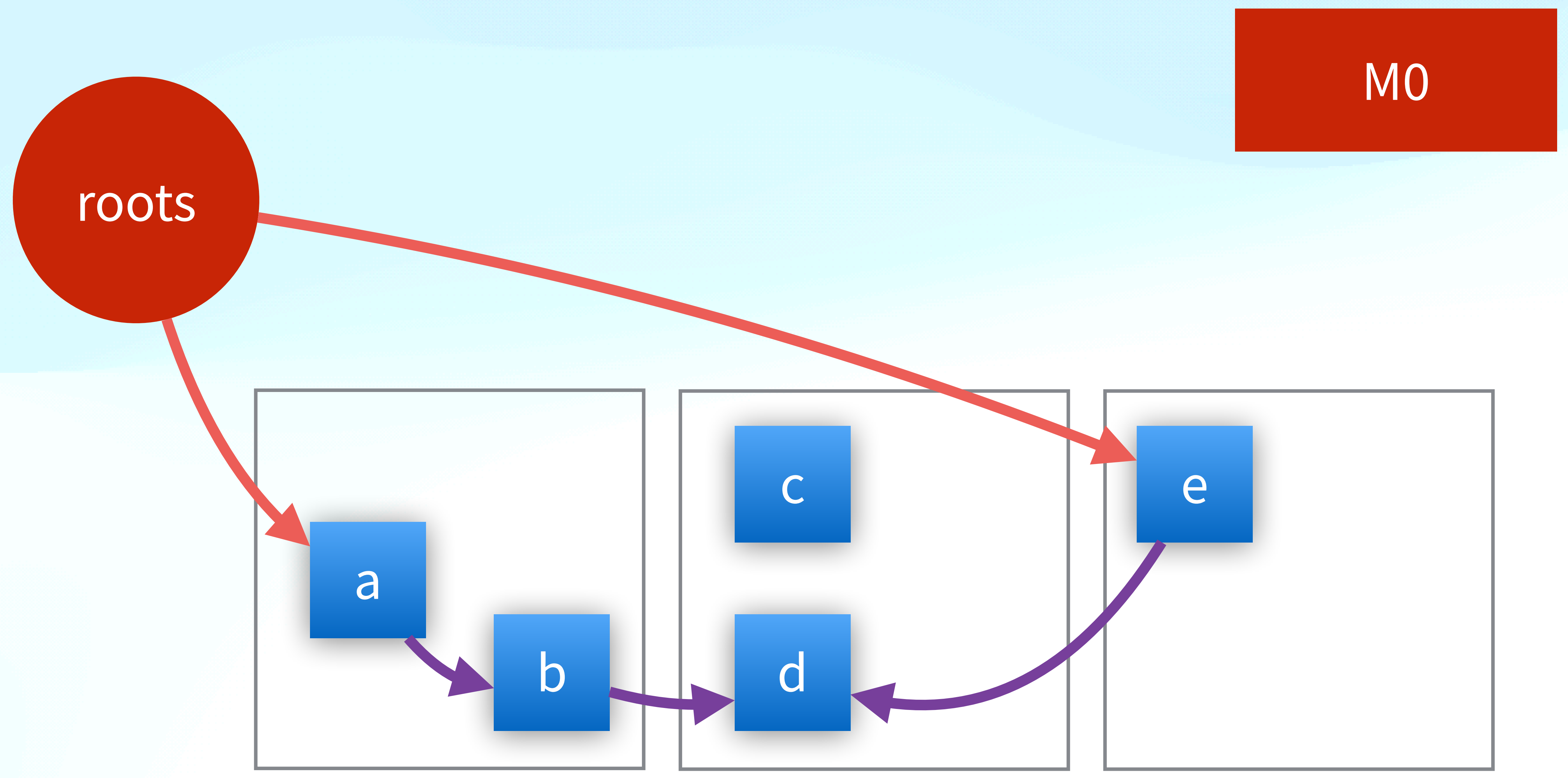
Relocation

t

Cycle 1

First STW Pause

All roots have good color

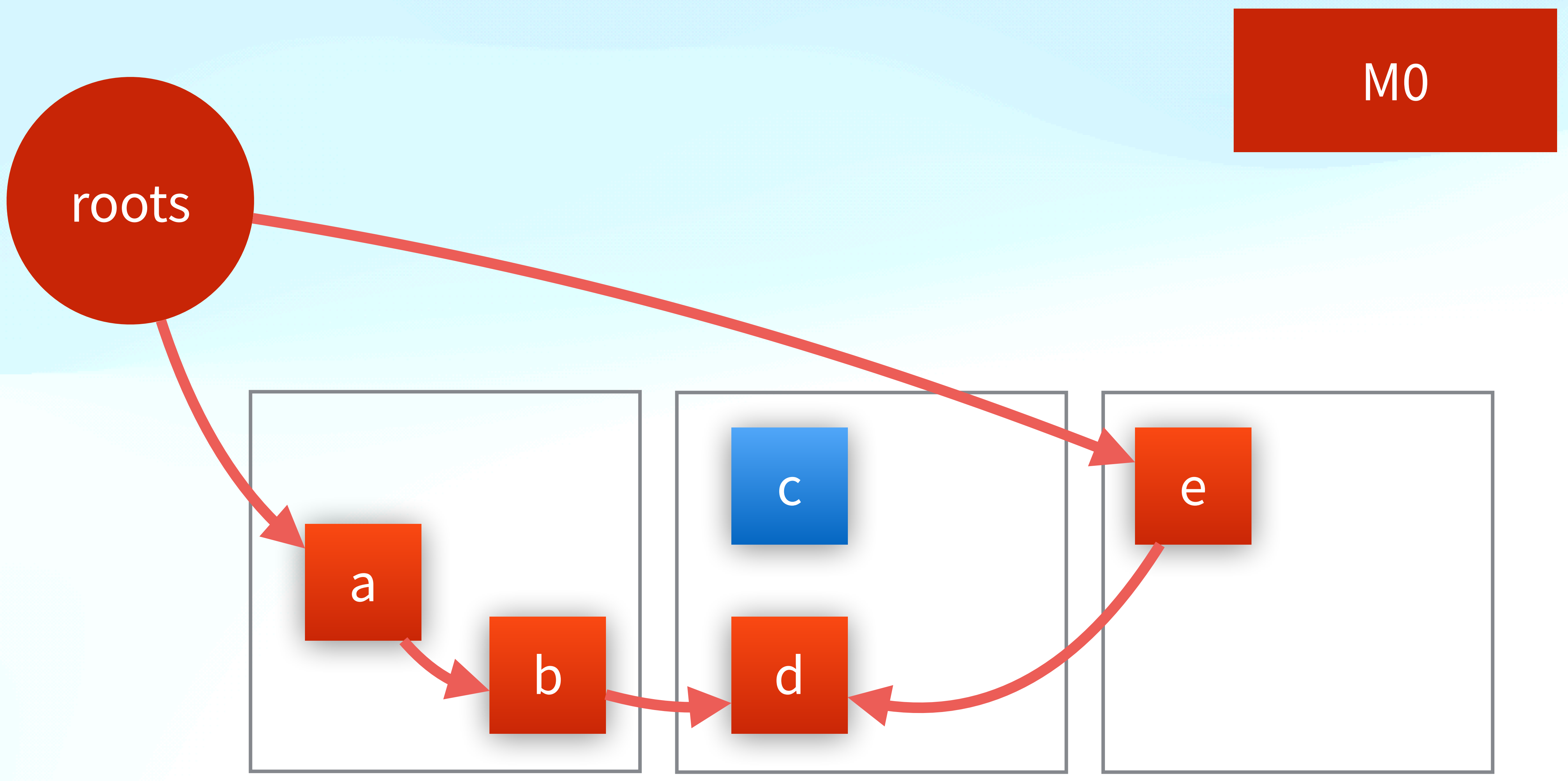


Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 1

Marking & Remapping

All reachable objects are marked



Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 1

Evacuation Candidate Selection

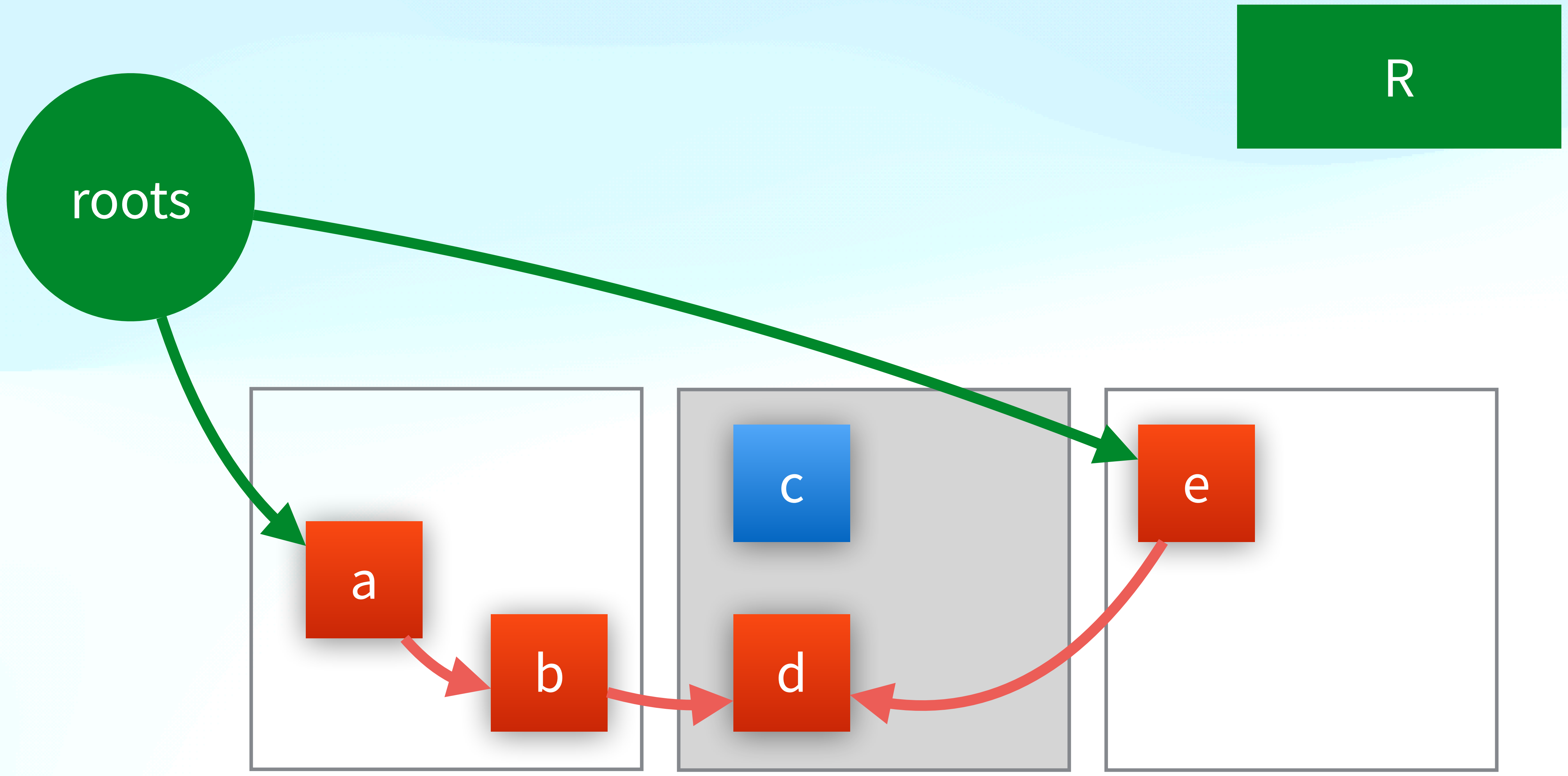
Middle page is selected for evacuation

M0
roots
c
e
a
b
d
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 1

Third STW Pause

All roots have good color

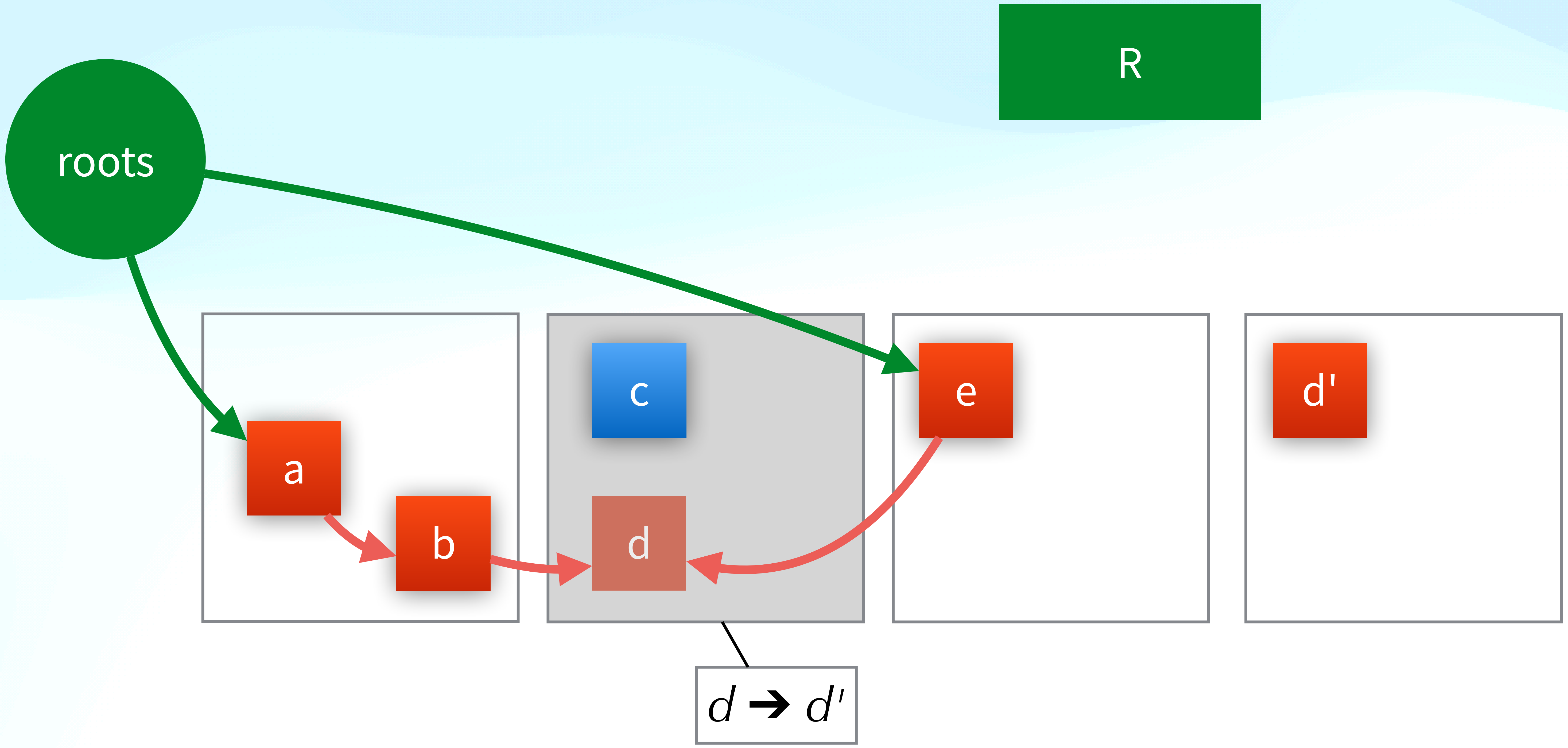


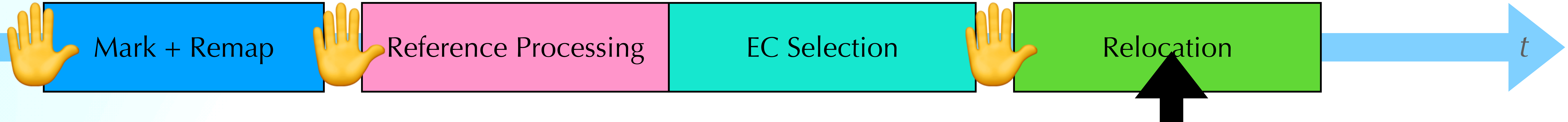
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 1

Relocation

EC Pages are evacuated (live objects copied to other pages)

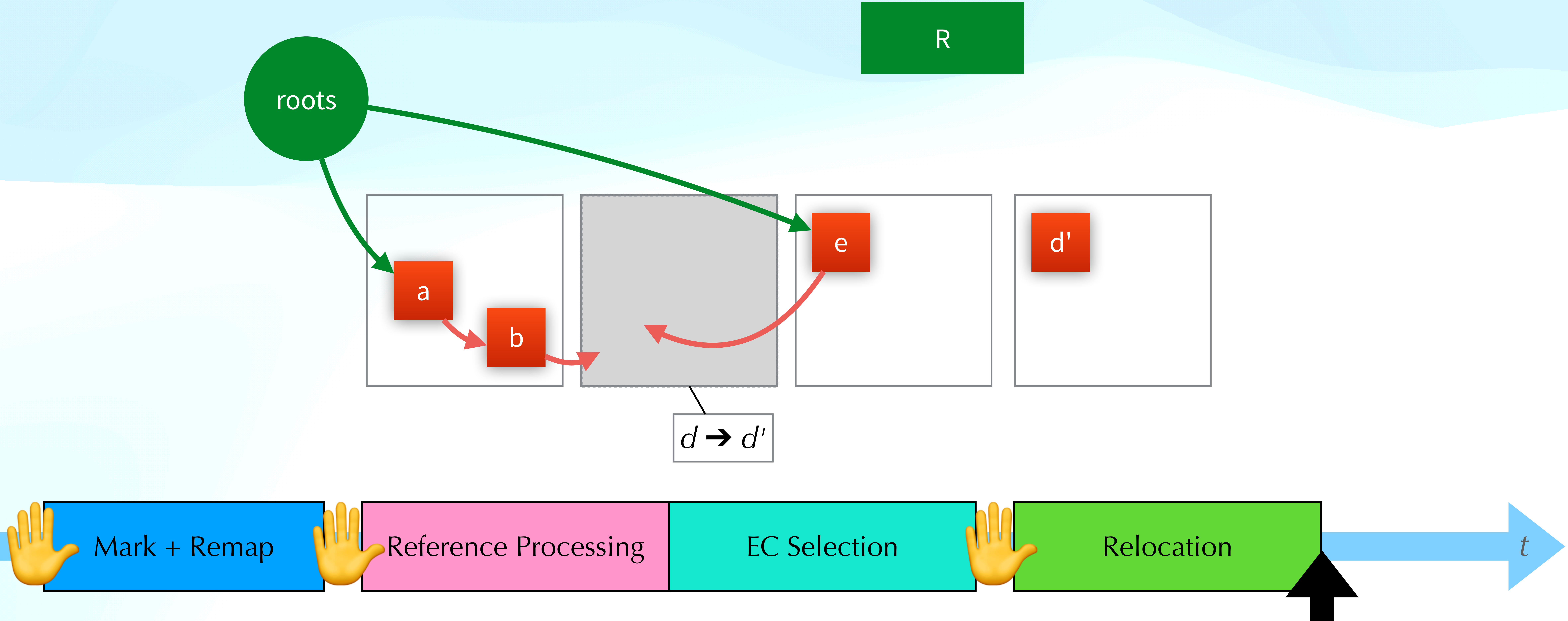




Cycle 1

Cycle End

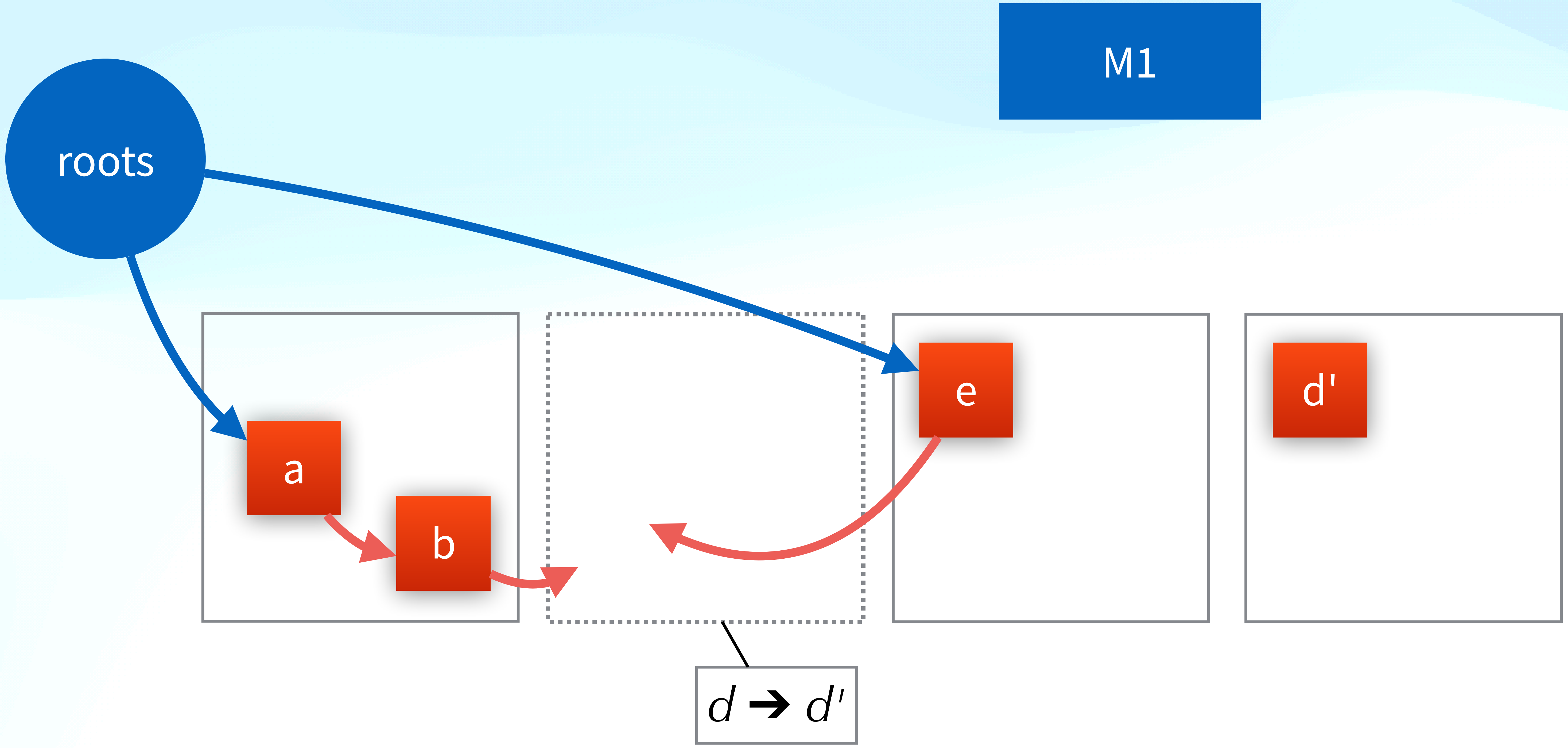
Pages in EC can be reclaimed



Cycle 2

First STW Pause

All roots have good color



Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 2

Mutator accesses *d* via e

Dangling Pointer with Bad Colour — Trapped by Load Barrier

M1
roots
a
b
e
d'
$d \rightarrow d'$
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 2

Mutator accesses *d* via *e*

Load barrier hits slow path — consults the forwarding table

M1
roots
a
b
e
d'
d ➔ *d'*
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 2

Mutator accesses *d* via *e*

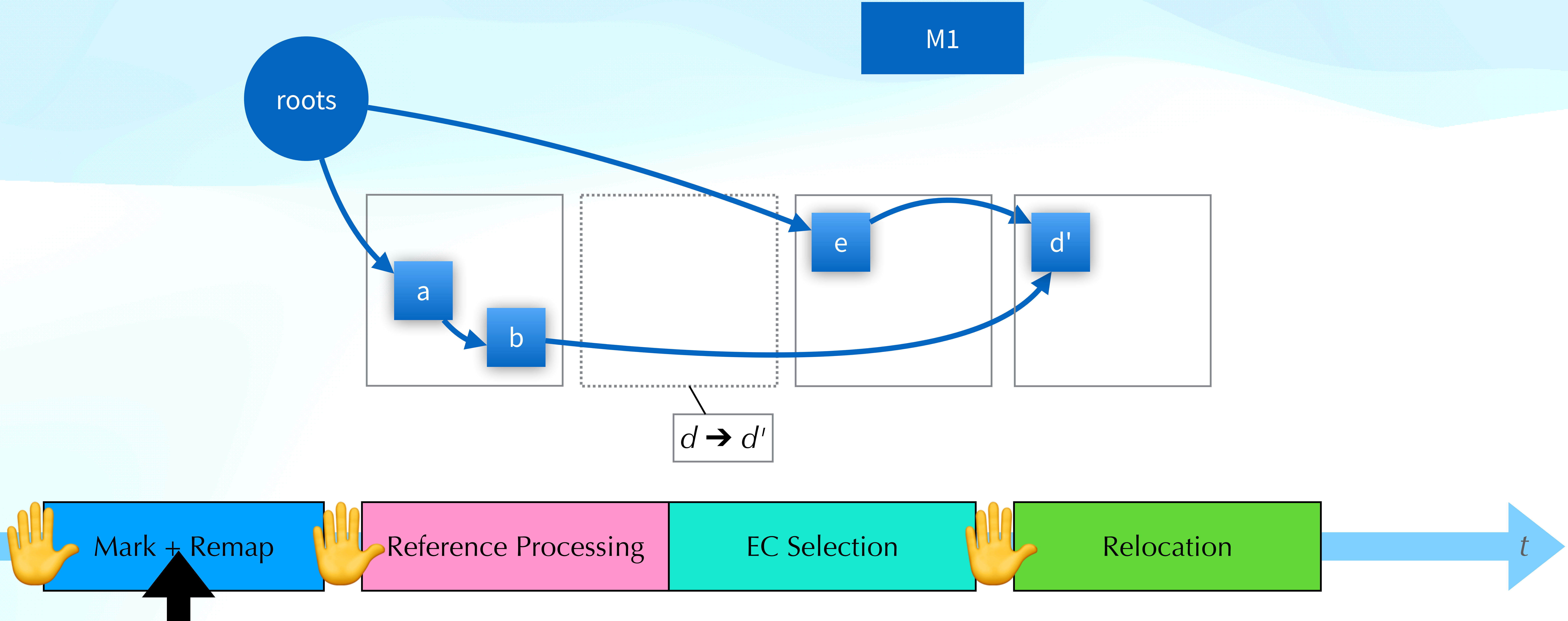
Performs self-healing to avoid future slow paths, and access continues

M1
roots
a
b
e
d'
$d \rightarrow d'$
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Cycle 2

Marking & Remapping

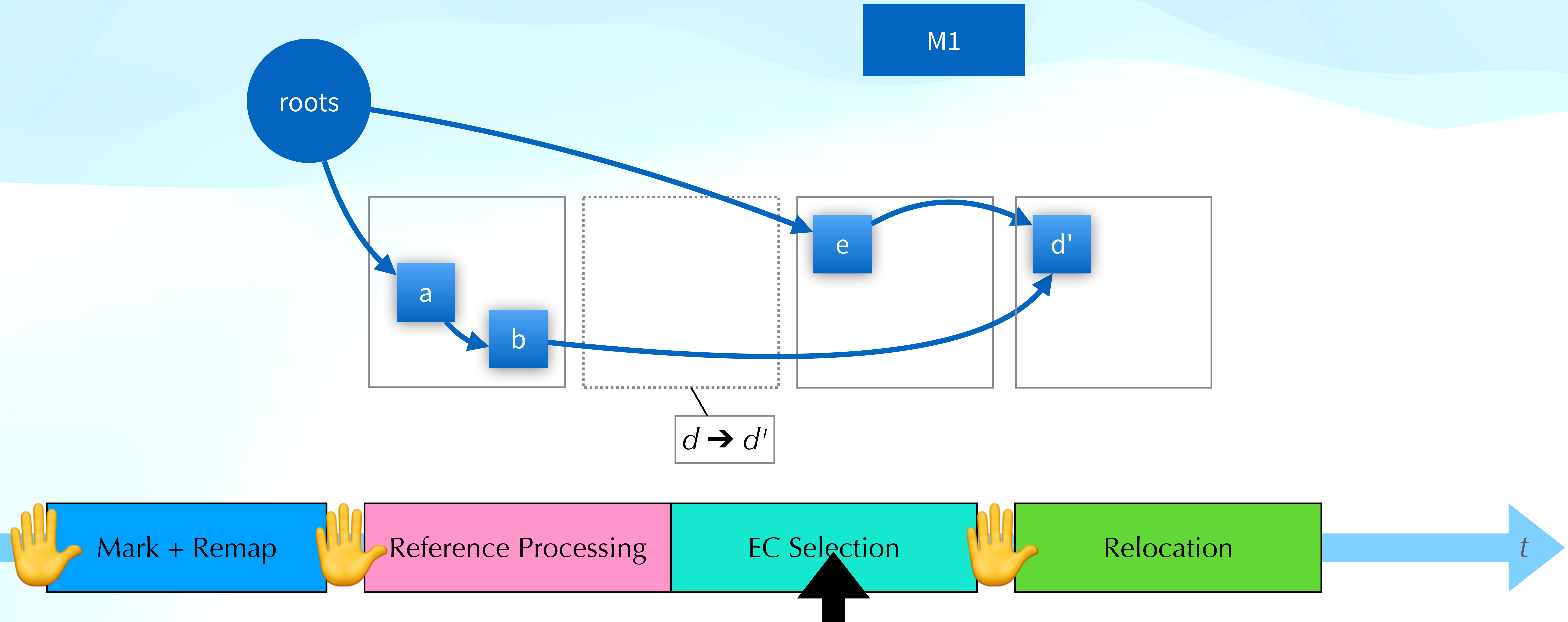
All outdated pointers are remapped



Cycle 2

Evacuation Candidate Selection

Forwarding tables from previous cycle are dropped



Summary Slide

- Threads synchronise briefly on what is the good colour
- All pointers are updated to have the good colour once per GC cycle (roots in each STW)
- Dangling pointers will be trapped by load barriers
- Evacuated pages can be dropped immediately
- Forwarding tables can be dropped after following marking phase

Aren’t load barriers too expensive?

- Nah; Compressed OOPS
- They can be — it depends on the load-barrier design

ZGC Load Barrier x86_64 Instructions

```
movq rax, 0x10(rbx)
testq rax, 0x20(r15)
jnz slow_path
```

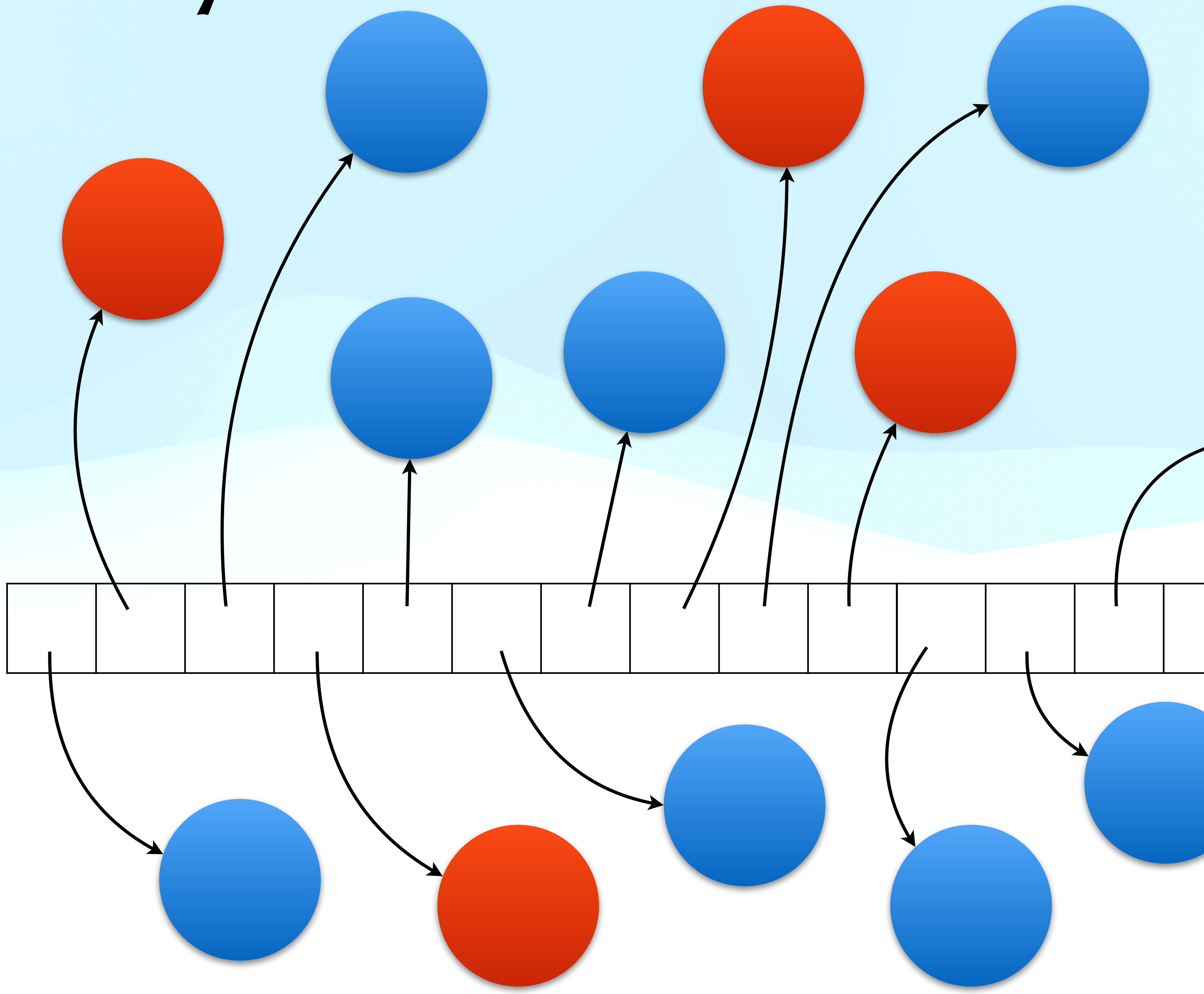
- ZGC used multi mapping to make coloured addresses valid

”If you screw up and make a load barrier that is more than 2 instructions, you’ve made your own mess”

Extending ZGC to improve locality

Yang, Österlund, and Wrigstad (PLDI 2020)

- Good spatial locality can hide memory latency
- Managed languages tend to abstract memory
    - Fewer bugs
    - Harder to optimise

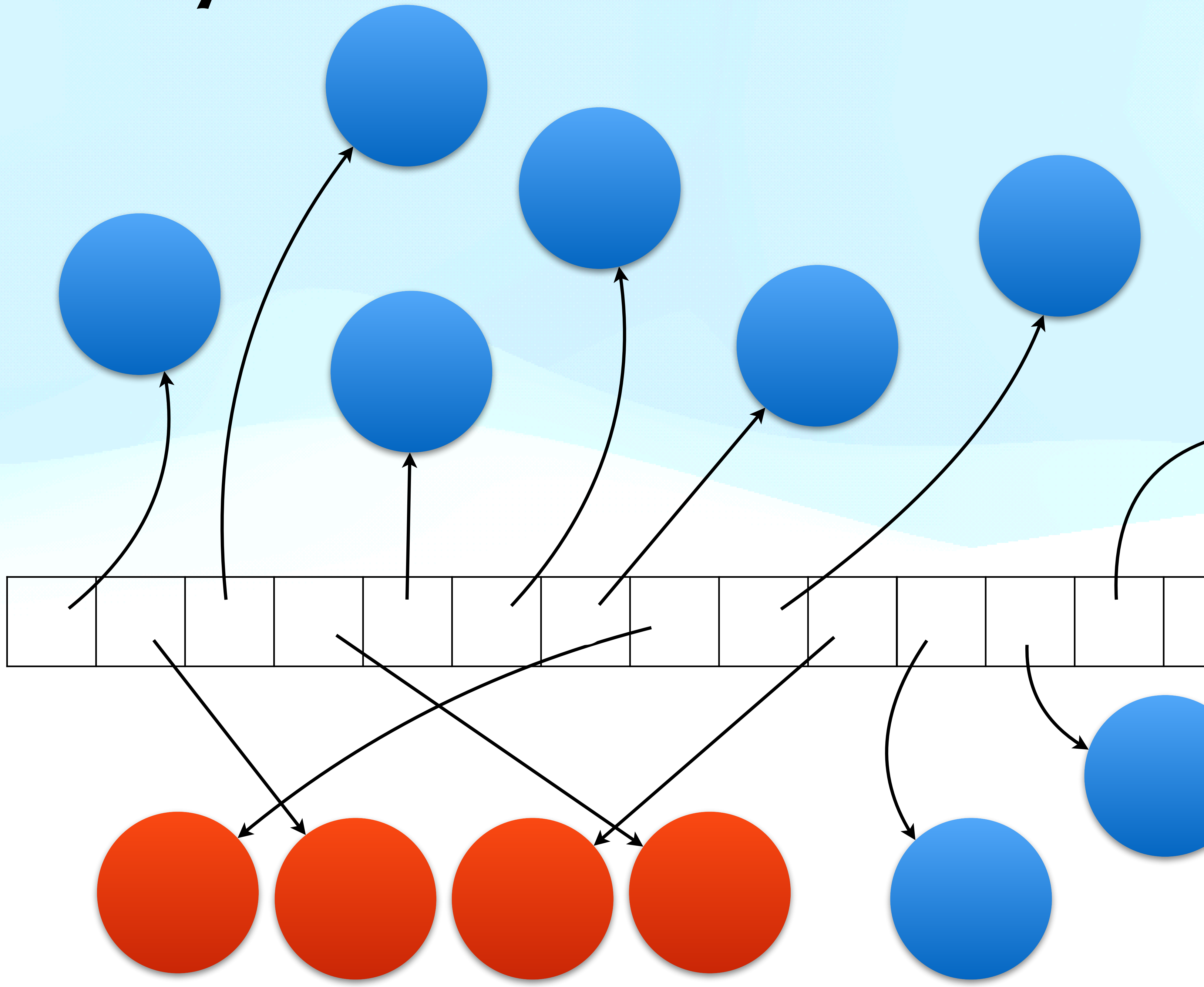


- **RQ:** how can we help programmers get good locality in managed languages?

Extending ZGC to improve locality

Yang, Österlund, and Wrigstad (PLDI 2020)

- Good spatial locality can hide memory latency
- Managed languages tend to abstract memory
  - Fewer bugs
  - Harder to optimise



- **RQ:** how can we help programmers get good locality in managed languages?

Extending ZGC to improve locality

Yang, Österlund, and Wrigstad (PLDI 2020)

What happens if a mutator accesses d before the GC during relocation?

R
roots
a
b
c
d
e
d'
$d \rightarrow d'$
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Design mismatches and mitigations

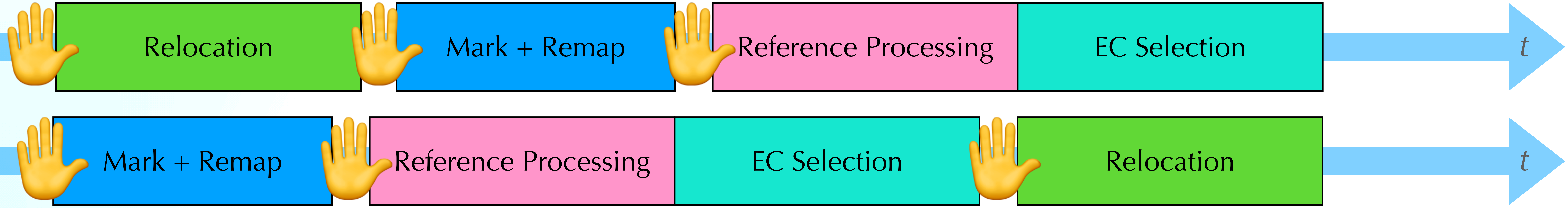
- In ZGC, relocation starts immediately after EC selection
  - Thus, mutators are competing with GC threads to move objects
  - **Question:** how can we help the mutators win that race?
- Only objects on pages in EC are movable
  - **Question:** how do we make more objects movable?

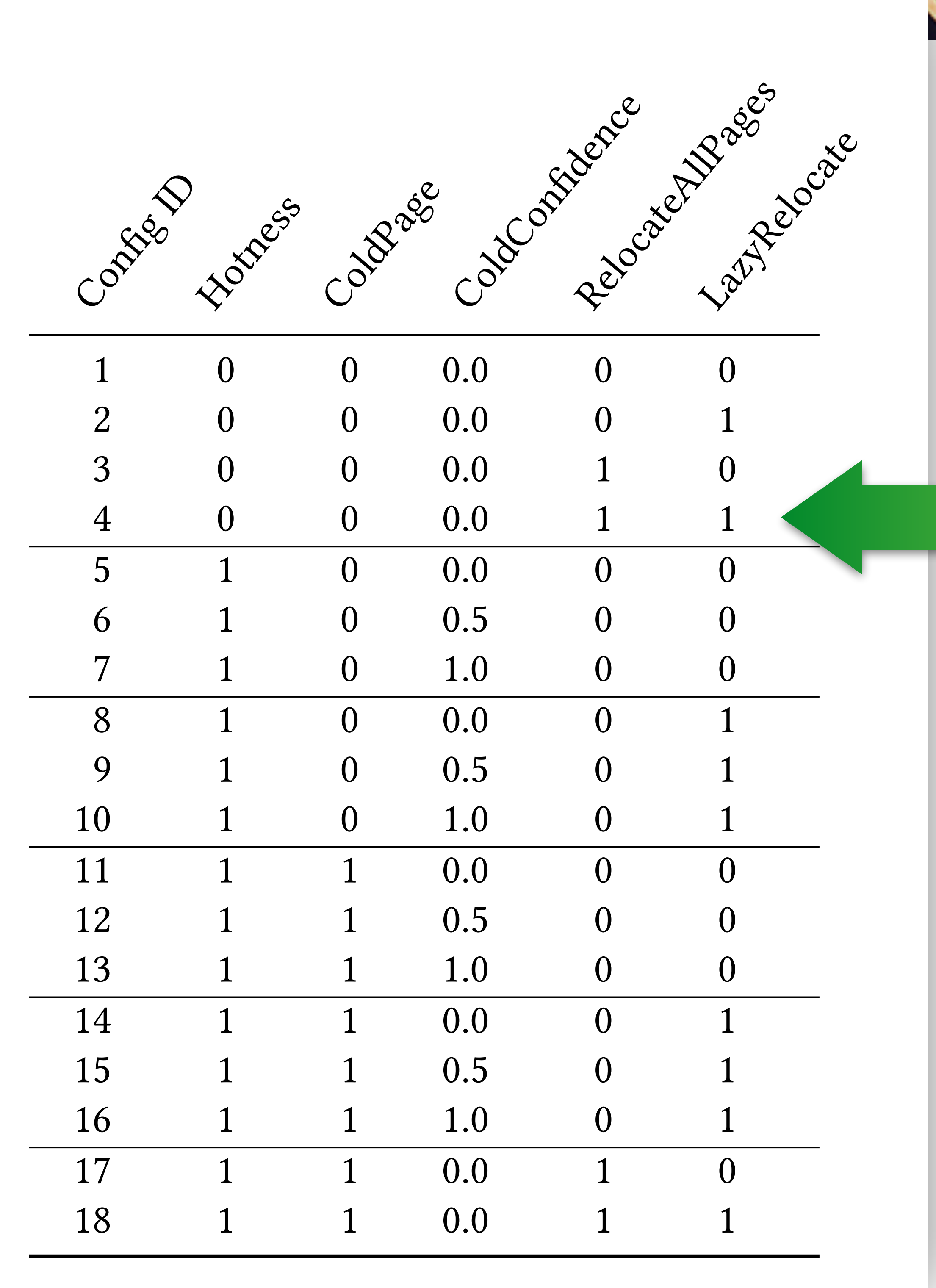
Mark + Remap
Reference Processing
EC Selection
Relocation
t

Design mismatches and mitigations

"Hacks": change phase order in the GC cycle; add all pages to EC

- In ZGC, relocation starts immediately after EC selection
  - Thus, mutators are competing with GC threads to move objects
  - **Question:** how can we help the mutators win that race?
- Only objects on pages in EC are movable
  - **Question:** how do we make more objects movable?



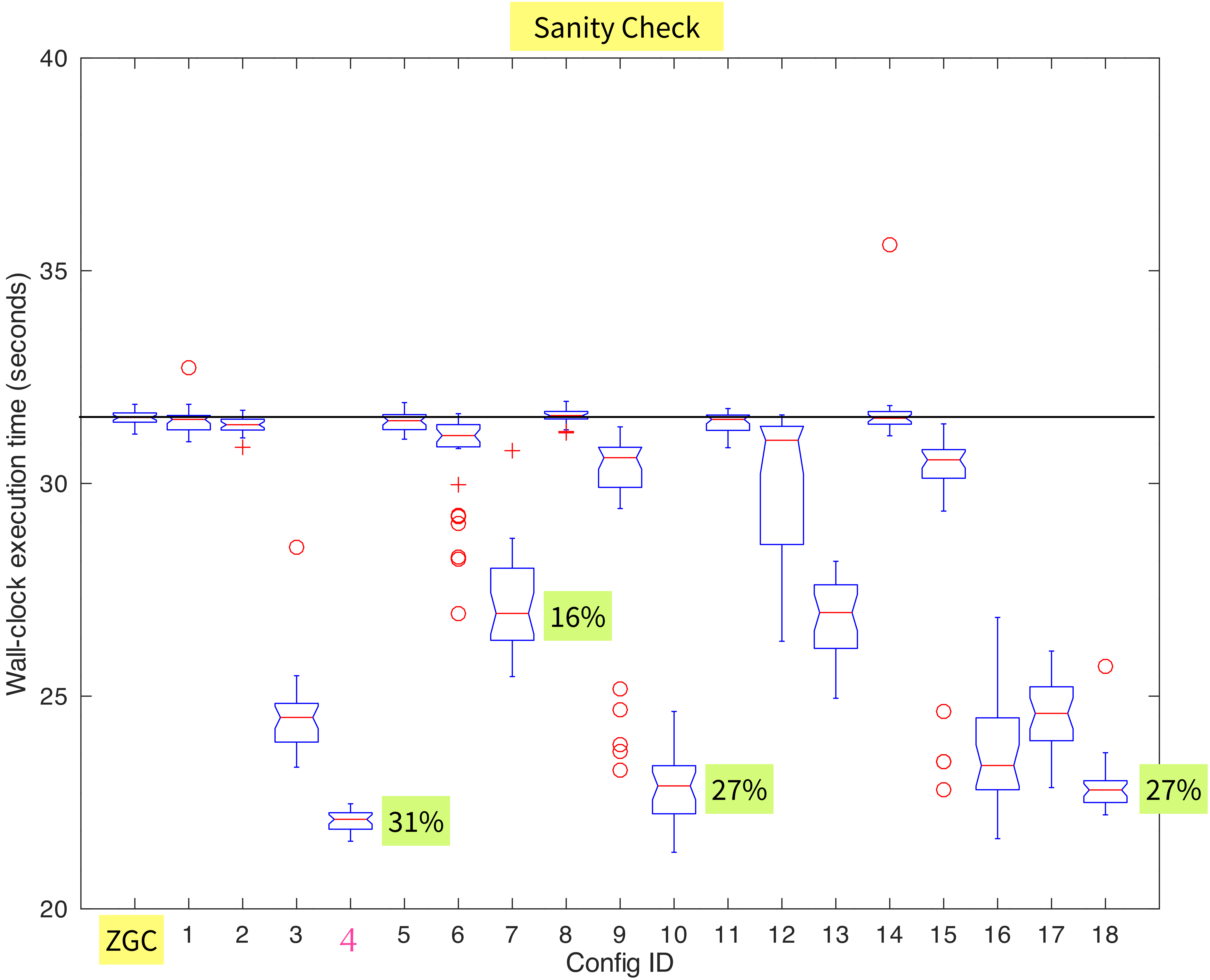


Config ID	Hotness	ColdPage	ColdConfidence	RelocateAllPages	LazyRelocate
1	0	0	0.0	0	0
2	0	0	0.0	0	1
3	0	0	0.0	1	0
4	0	0	0.0	1	1
5	1	0	0.0	0	0
6	1	0	0.5	0	0
7	1	0	1.0	0	0
8	1	0	0.0	0	1
9	1	0	0.5	0	1
10	1	0	1.0	0	1
11	1	1	0.0	0	0
12	1	1	0.5	0	0
13	1	1	1.0	0	0
14	1	1	0.0	0	1
15	1	1	0.5	0	1
16	1	1	1.0	0	1
17	1	1	0.0	1	0
18	1	1	0.0	1	1

4

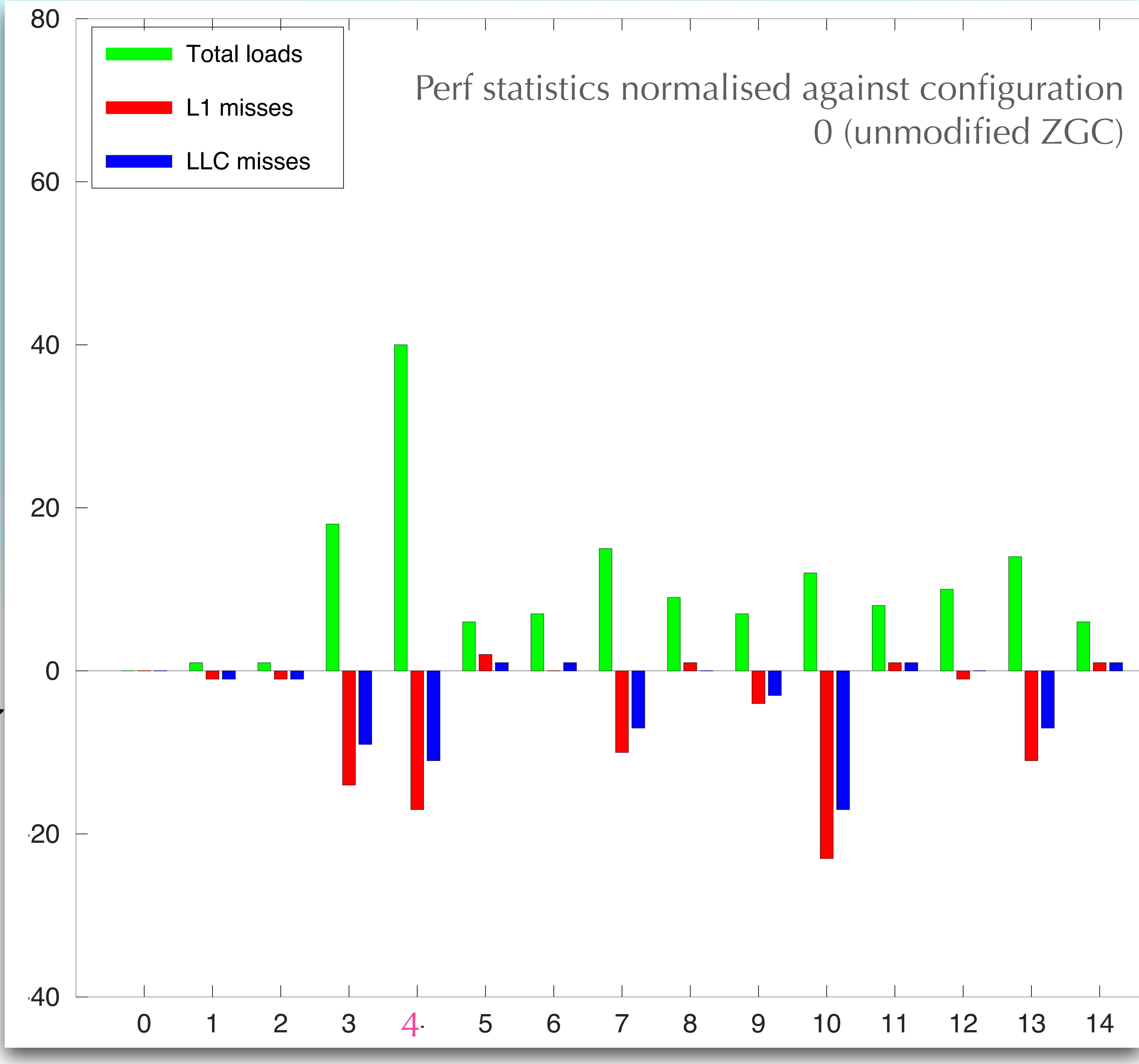
Sanity Check Result

(Expect better performance)



HCSGC

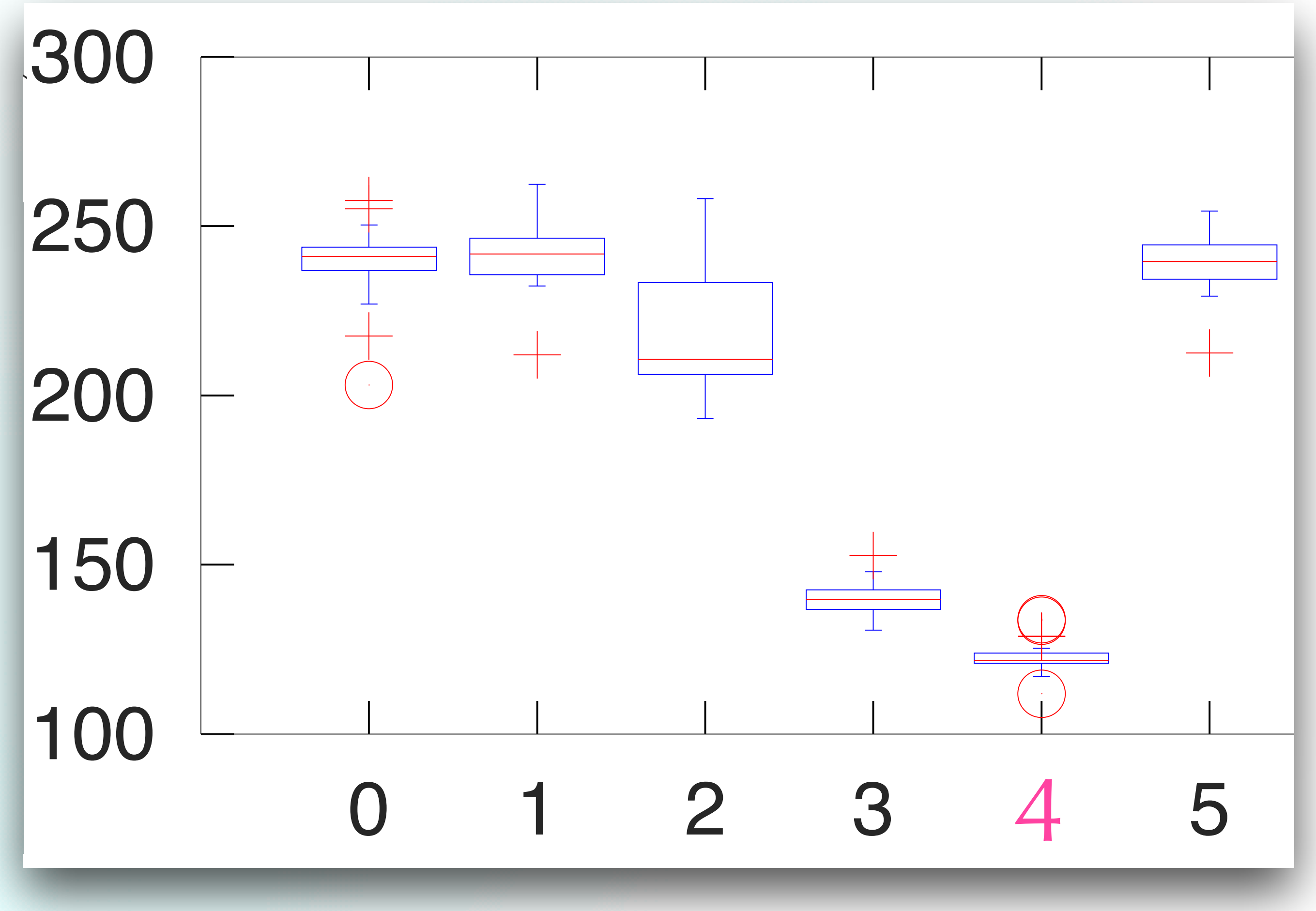
Why?

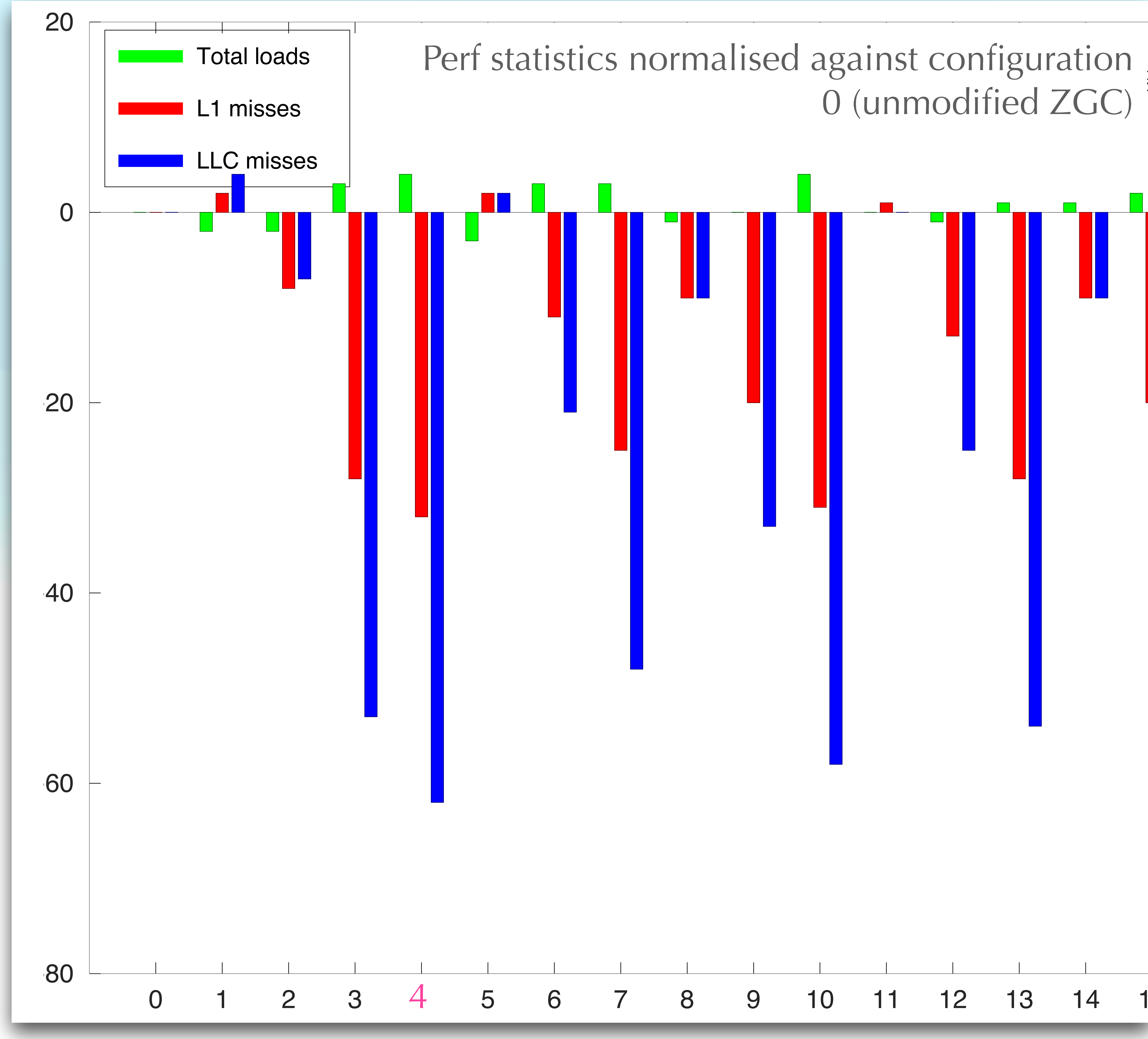


Cache performance

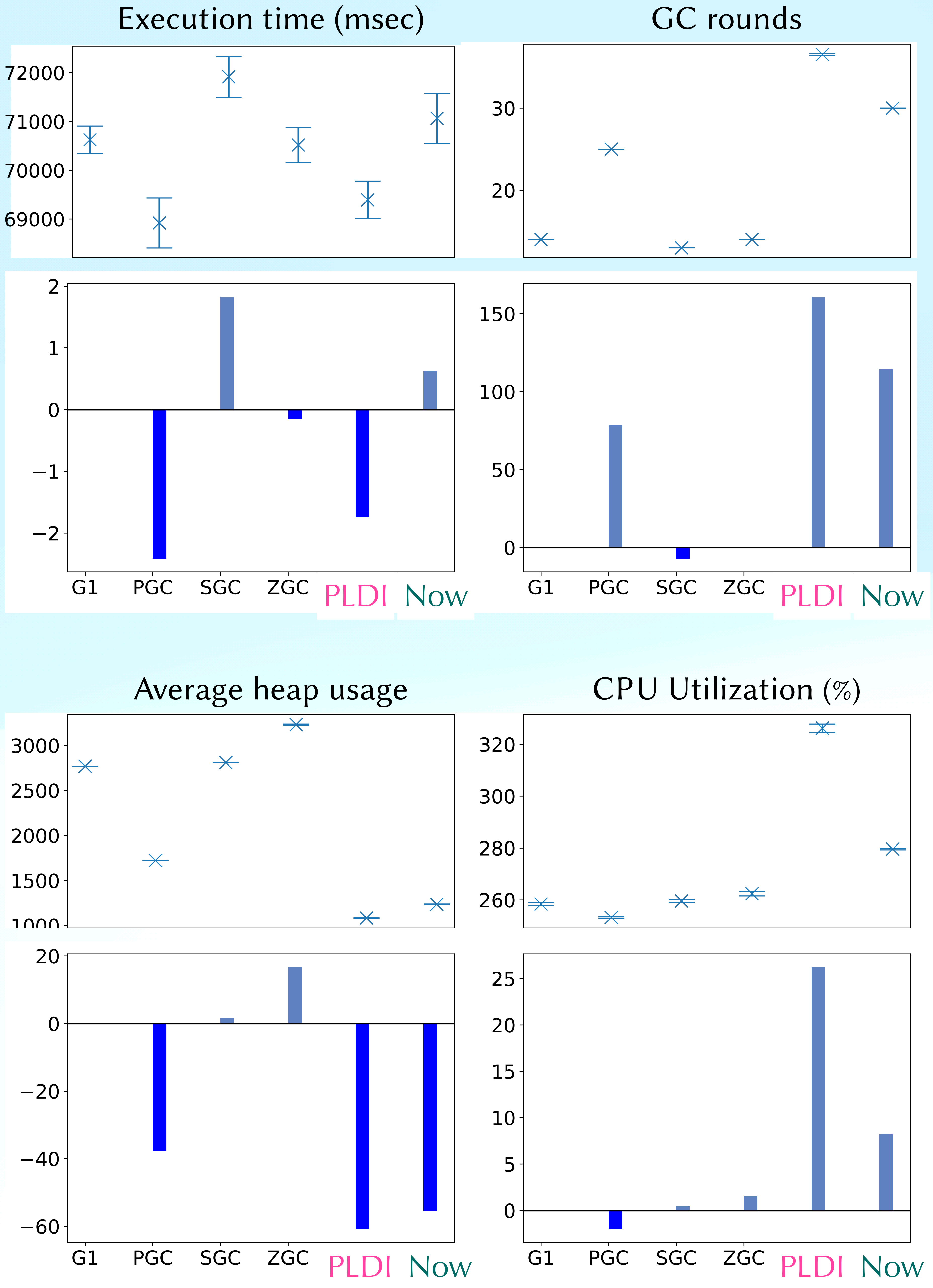
JGraphT Results

Best case for our technique

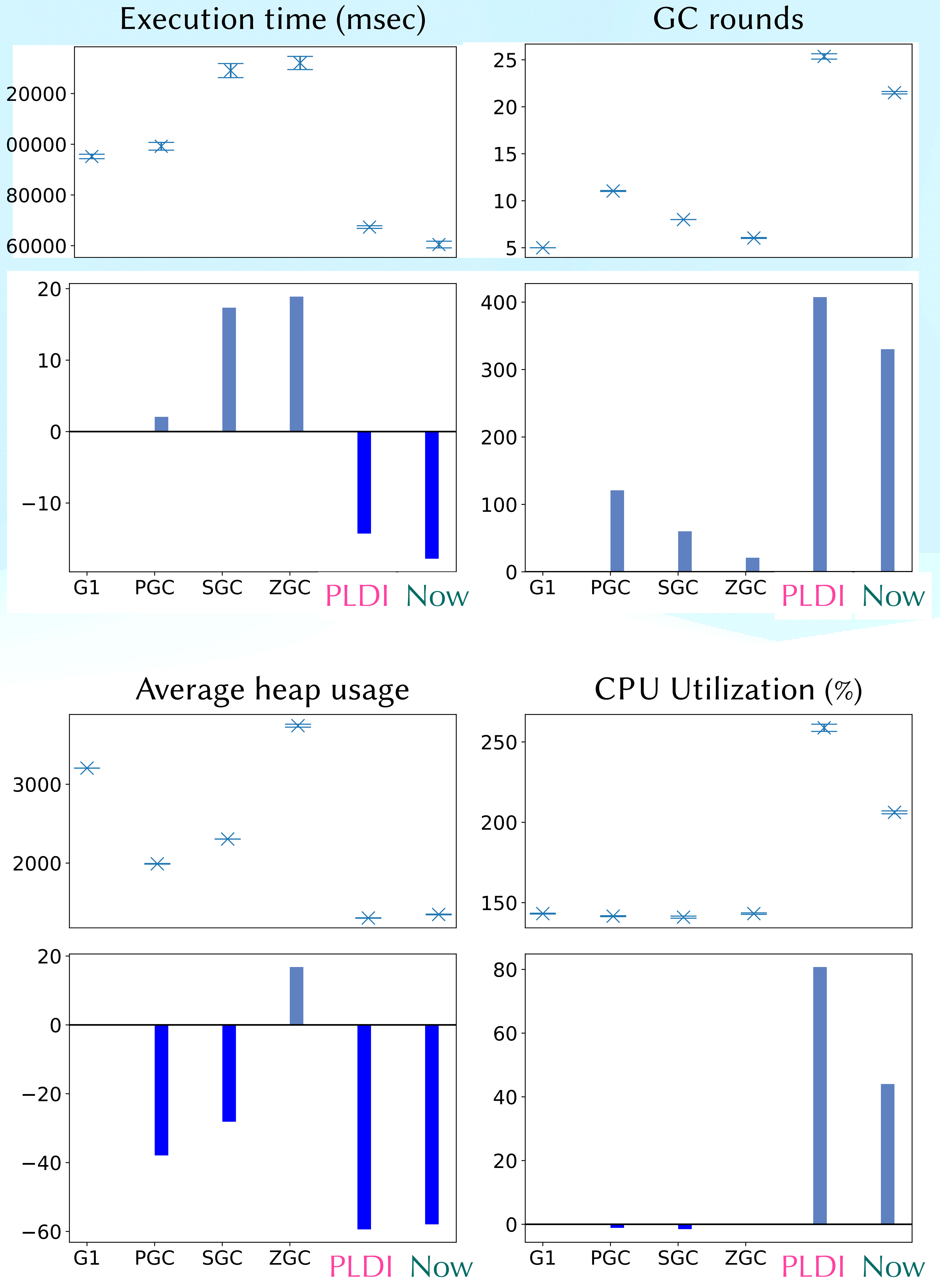




Continuation of this work



tradebeans



h2

Speculative Optimisation

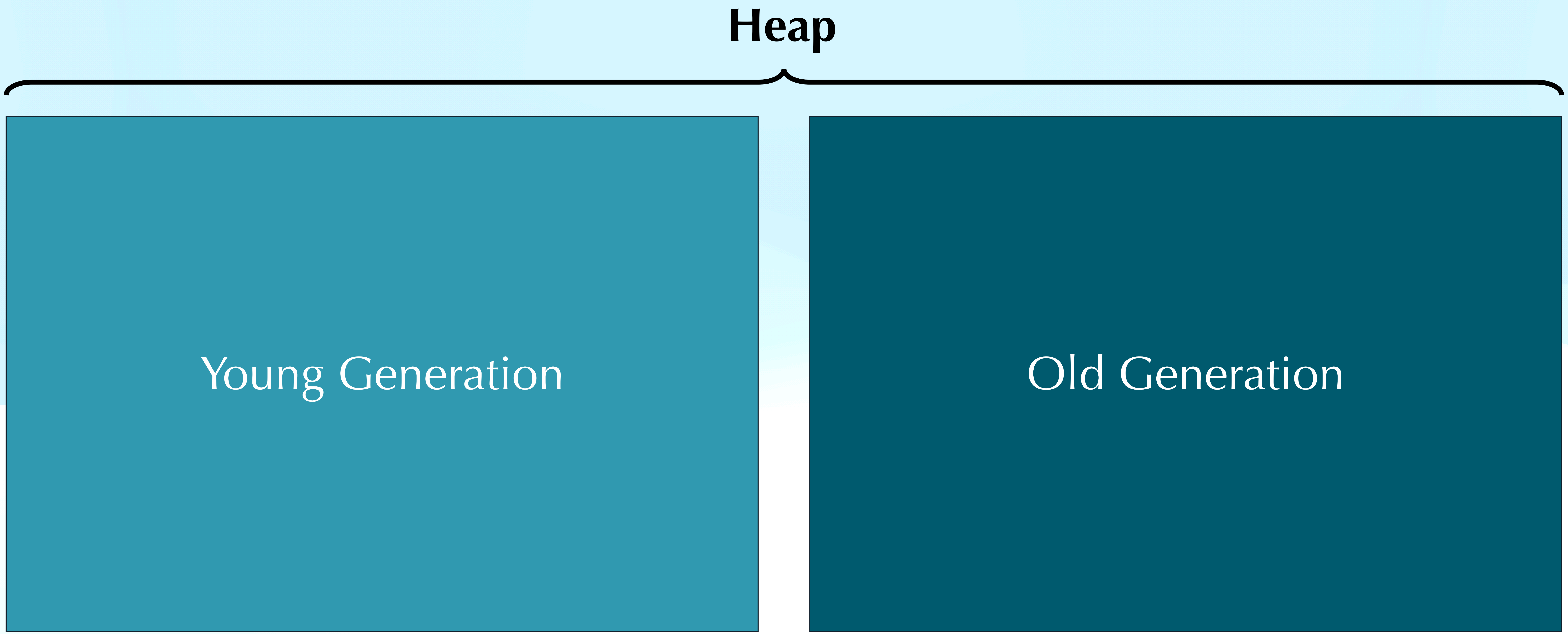
- We are letting the mutator copy objects hoping that will improve performance
- Filtering is key — don’t want to copy objects…
  - …that are too large (almost no point)
  - …that are already well-placed
  - …that won’t be accessed again
  - …etc.
- **RQ:** how can we make that filtering (and efficiently)?

Generational ZGC

Generational Garbage Collectors

For programs that satisfy the generational hypothesis

- Divide the heap into multiple (two) spaces (young, old) managed separately
- Objects in young space likely to die — high ROI on GC in young space
- Objects in old space less likely to die — lower ROI on GC in old space
- Reduced Effort to Collect Garbage
  - Withstand higher allocation rates
  - Lower heap headroom
  - Lower CPU usage

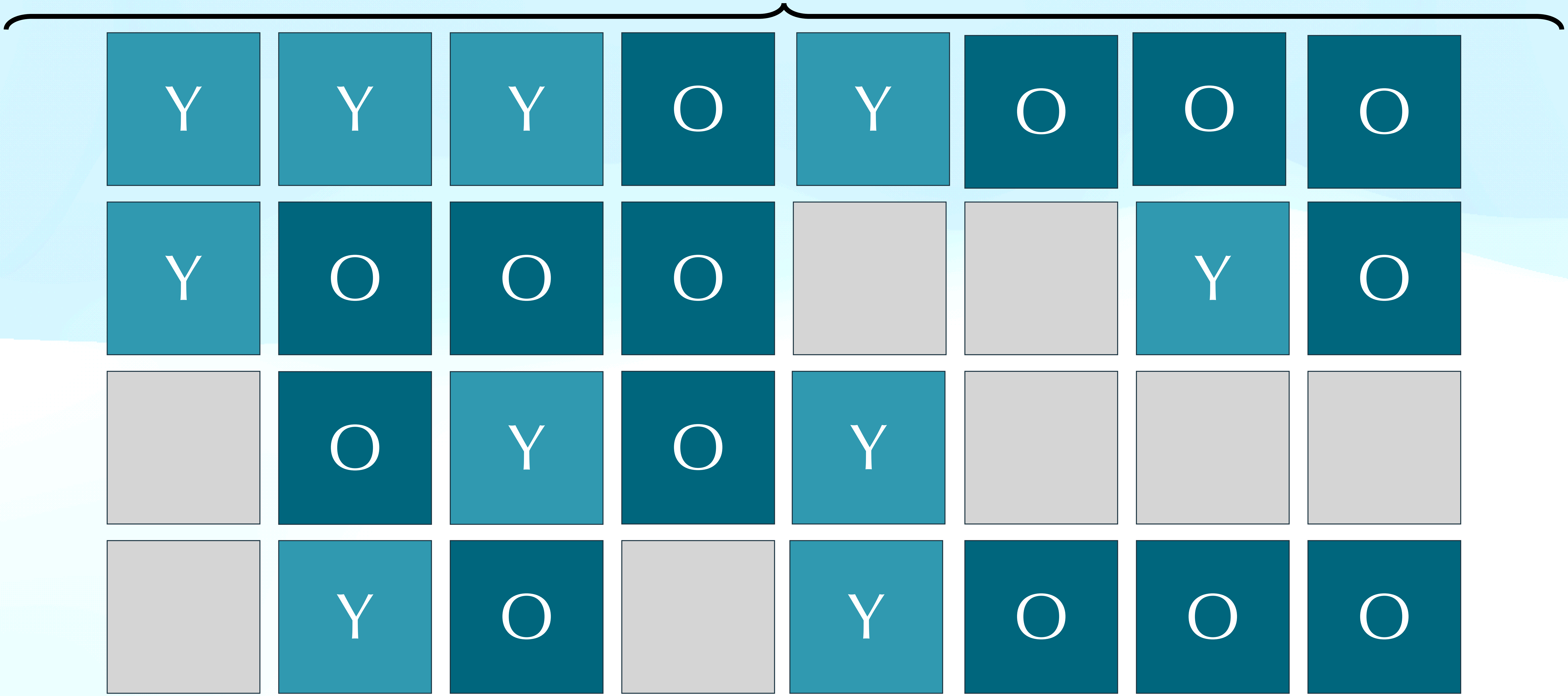


Heap

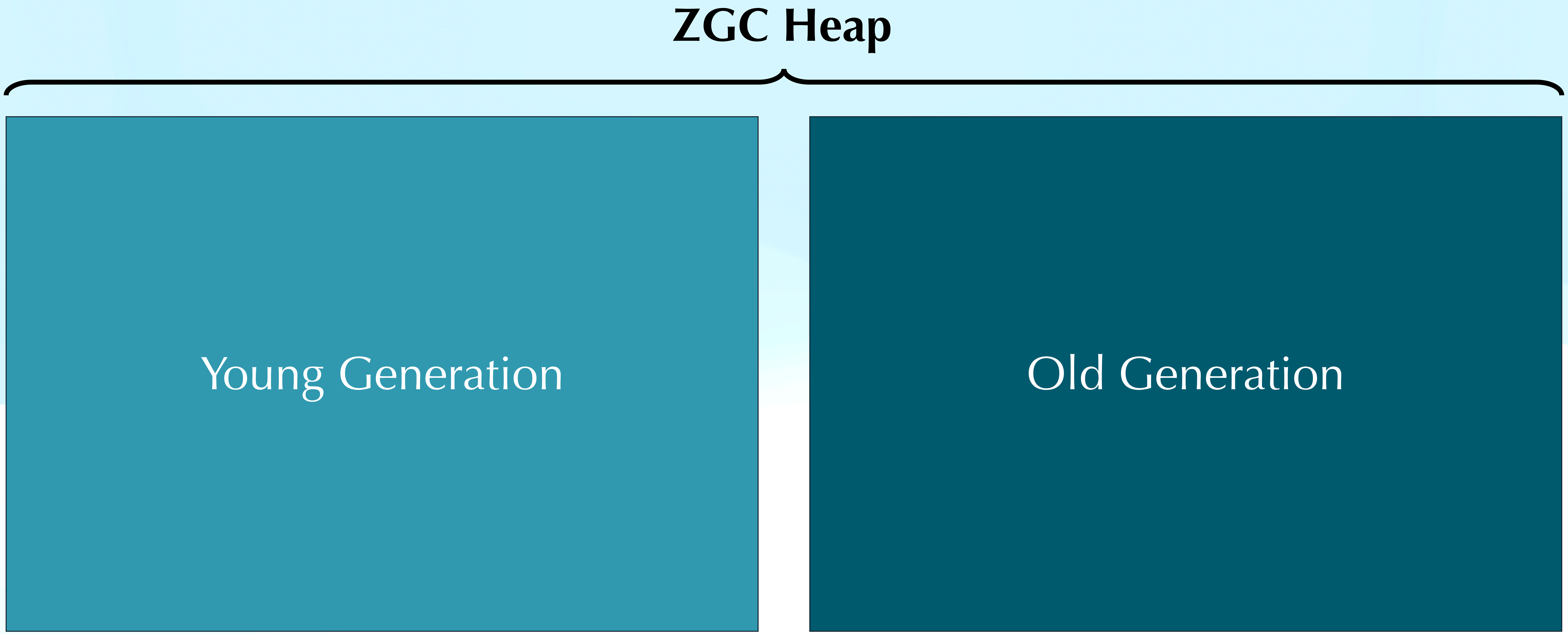
Young Generation

Old Generation

ZGC Heap



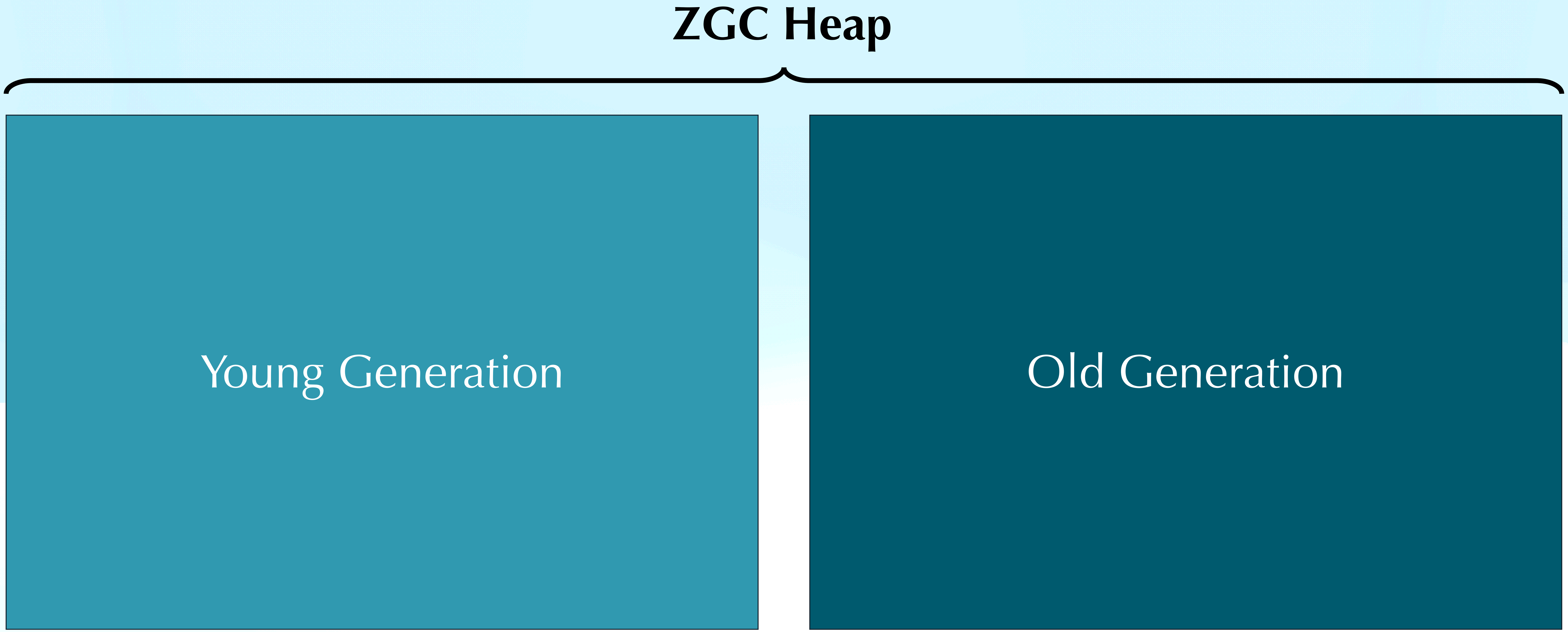
Y	Y	Y	O	Y	O	O	O
Y	O	O	O			Y	O
	O	Y	O	Y			
	Y	O		Y	O	O	O



ZGC Heap

Young Generation

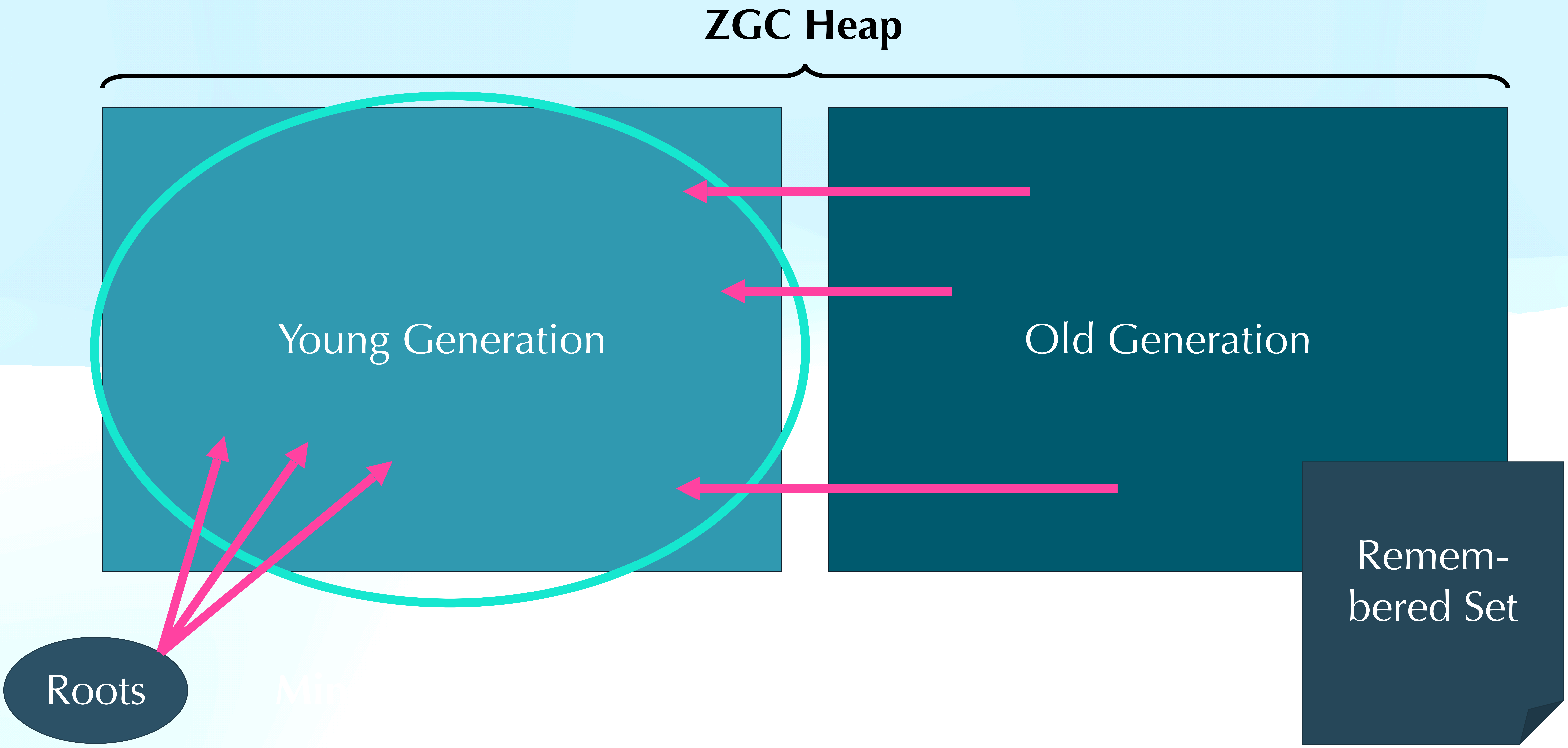
Old Generation



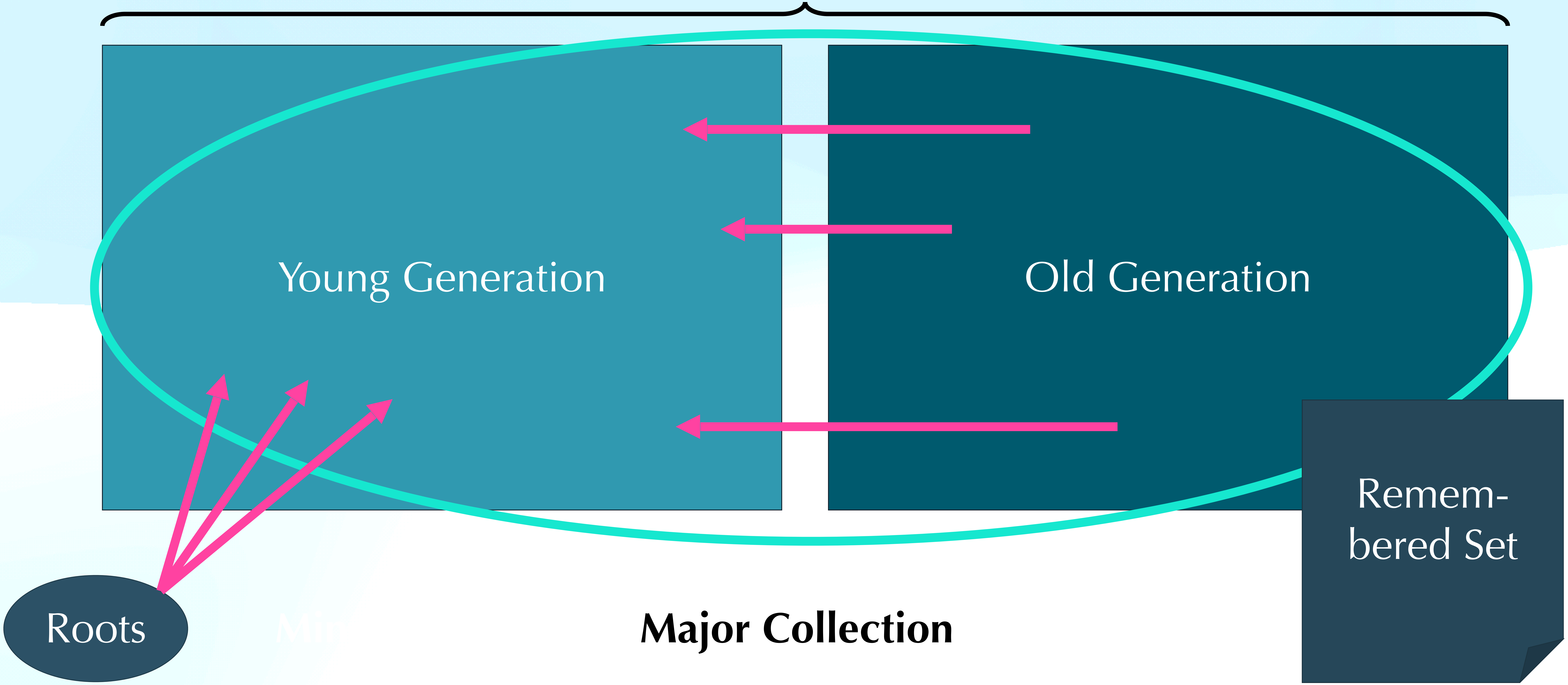
ZGC Heap

Young Generation

Old Generation

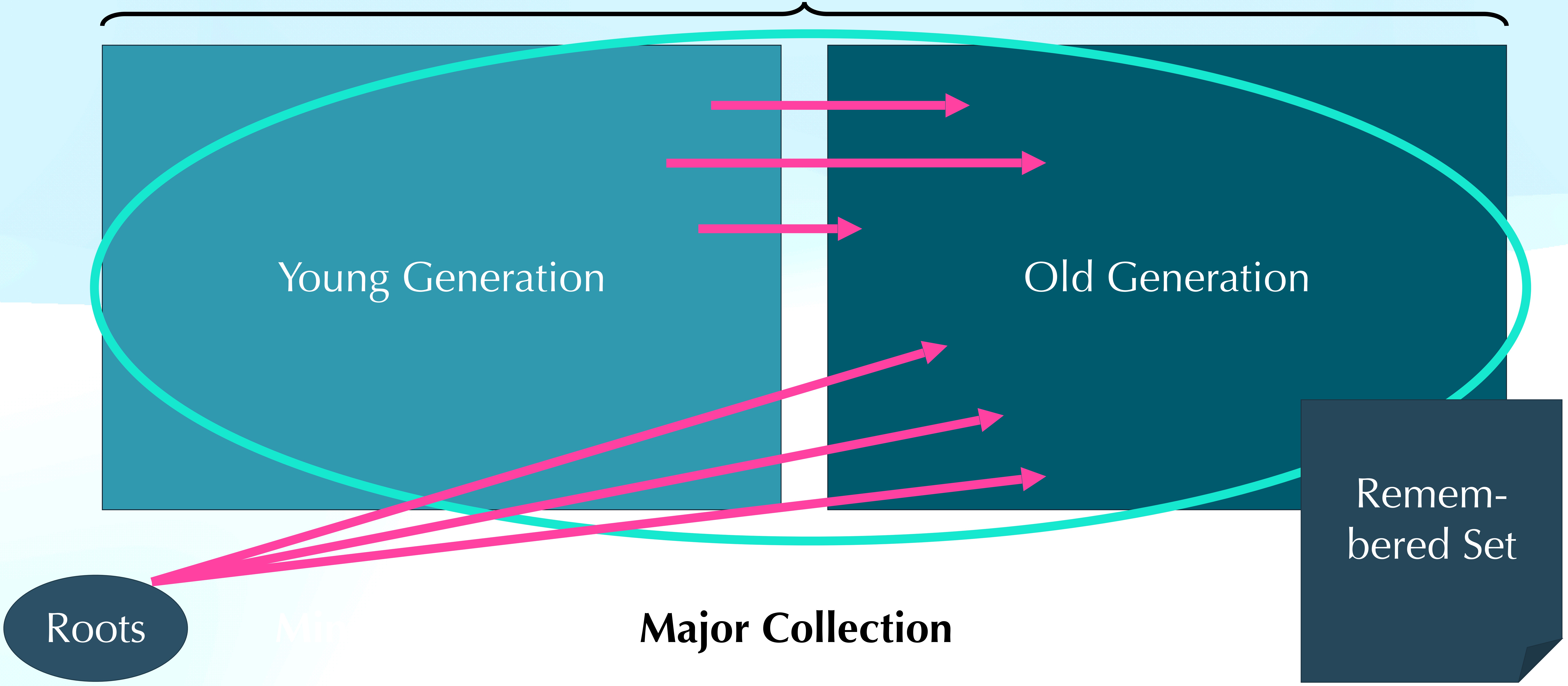

ZGC Heap
Young Generation
Old Generation
Remem-bered Set
Roots
Min

ZGC Heap



Major Collection

ZGC Heap



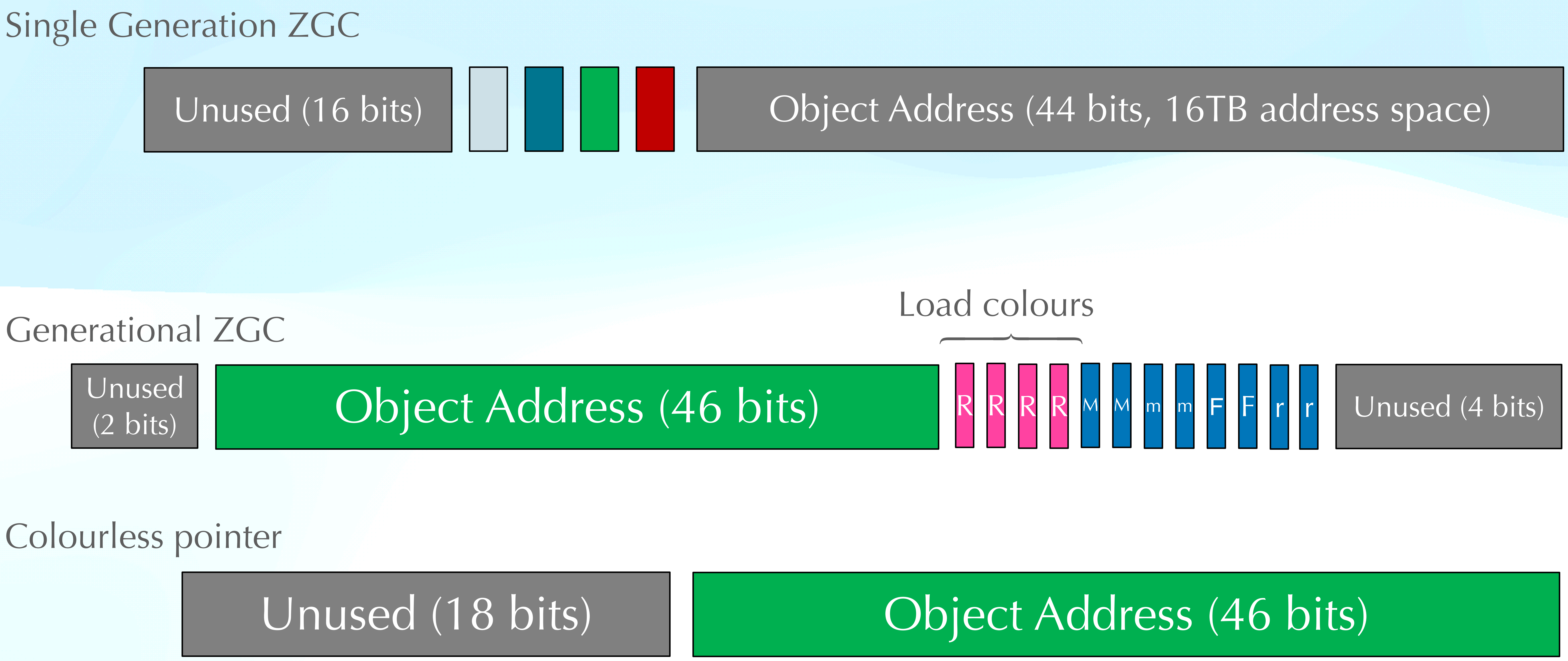
Major Collection

Generational ZGC

- Single-generation ZGC only used load barriers
- Pointer metadata need grows (colour space); multi-mapping no longer feasible
  - But we still want to keep load barrier at two instructions

- Generational ZGC needs store barriers as well
  - Remembered-set maintenance
  - Colourless roots

Comparing Pointer Anatomies (simplified)



Single Generation ZGC

Unused (16 bits)

Object Address (44 bits, 16TB address space)

Generational ZGC

Load colours

Unused (2 bits)

Object Address (46 bits)

R R R R M M m m F F r r

Unused (4 bits)

Colourless pointer

Unused (18 bits)

Object Address (46 bits)

Colour Hierarchy

✋ Mark + Remap
✋ Reference Processing
EC Selection
✋ Relocation
t

Load Good < Finalizable Good < Mark Good < Store Good

OK to load
Marked live
Added to remset

Evolution of ZGC Barriers

Load Barriers

Non-Generational ZGC

```
movq rax, 0x10(rbx)
testq rax, 0x20(r15)
jnz slow_path
```

Generational ZGC

```
movq rax, 0x10(rbx)
shrq rax, $address_shift
ja slow_path
```

Store Barrier

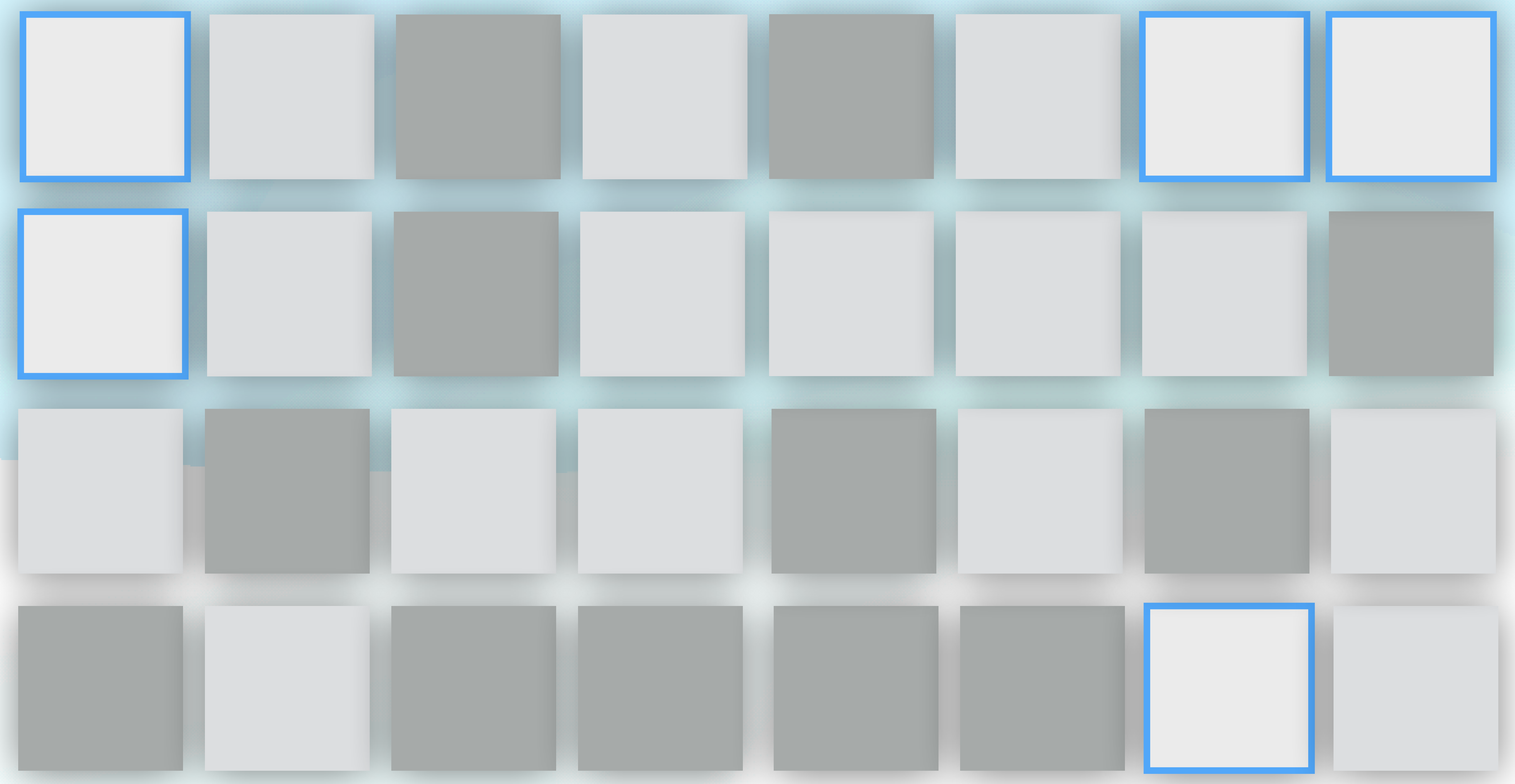
Generational ZGC

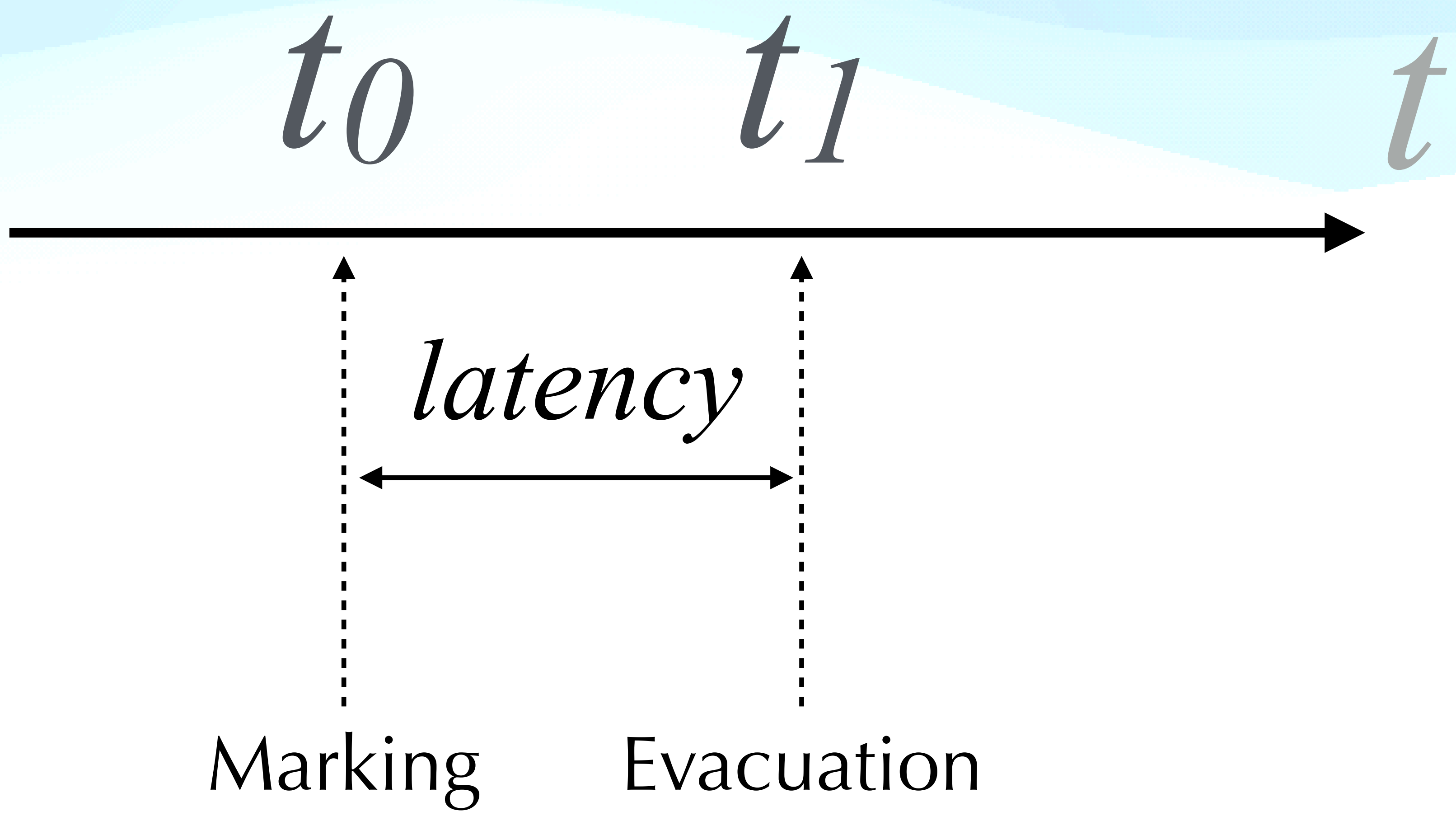
Value in field to be overwritten

```
testl 0x10(rbx), $store_bad_mask
jnz slow_path
shlq rax, $address_shift
orq, rax, $colors
movq 0x10(rbx), rax
```

- `$address_shift`, `$store_bad_mask`, and `$colors` are updated with cross-modifying code
- Carry Flag = last removed bit
- Zero Flag = is it null
- “Jump if above” jumps iff CF == 0 && ZF == 0

Snapshot-at-the-beginning marking

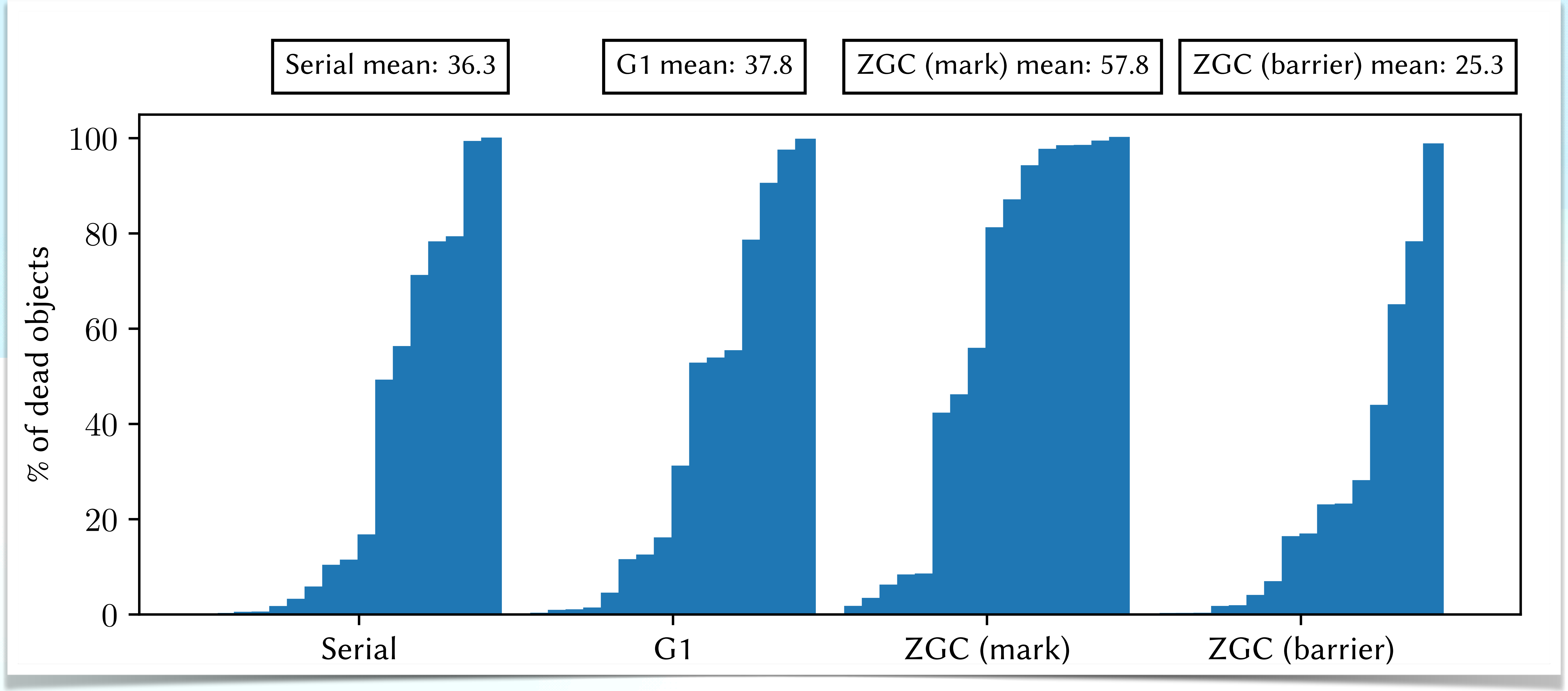


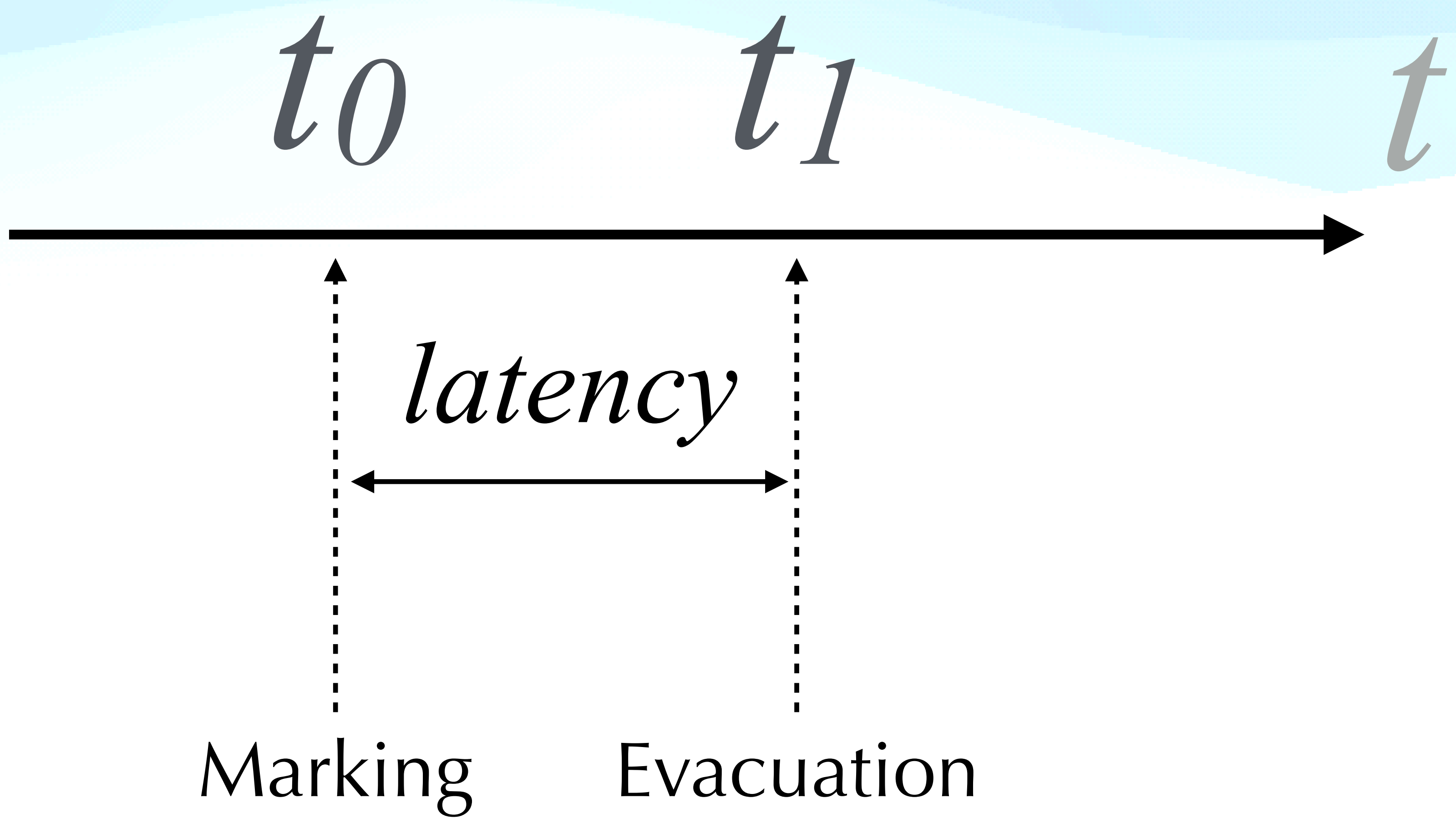


How good is information from t_0 at t_1?

Consequences of acting on stale liveness

Investigation using DaCapo (version at Git hash 6e411f33)

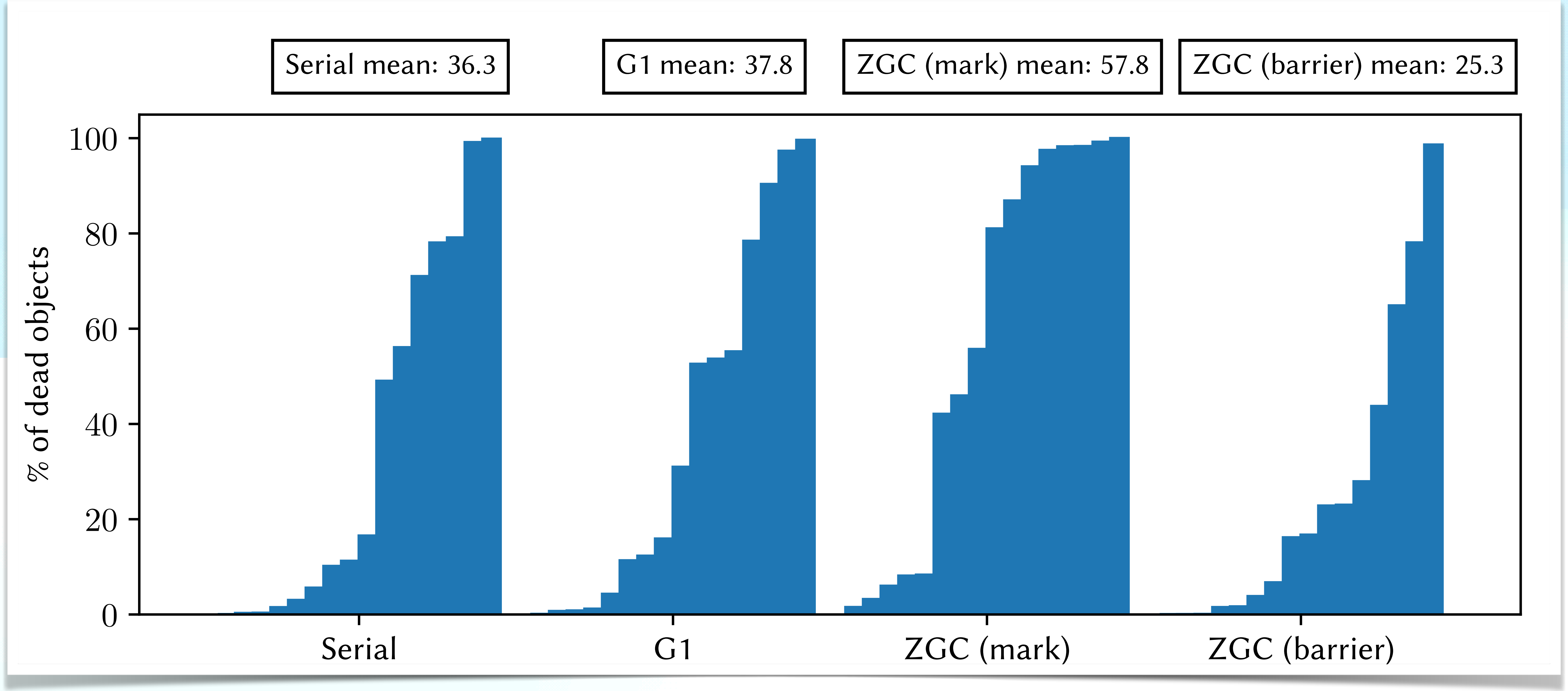




25% of all objects that we copy were effectively dead at the time

Consequences of acting on stale liveness

Investigation using DaCapo (version at Git hash 6e411f33)



Spring	GB	%
Relocated	398	100 %
Accessed	31	8 %
Dead	367	92 %

25% of all objects that we copy were effectively dead at the time

Copy fewer dead objects & reduce barrier storms

Disclaimer: work-in-progress

✋ Mark + Remap
✋ Relocation (Old)
Relocation (Young)
✋ Mark + Remap
✋ Relocation (Old)
Relocation (Young)
t
✋ Mark + Remap
✋ Relocation (Old)
Relocation (Young)
✋ Mark + Remap
✋ Relocation (Old)
Relocation (Young)
t

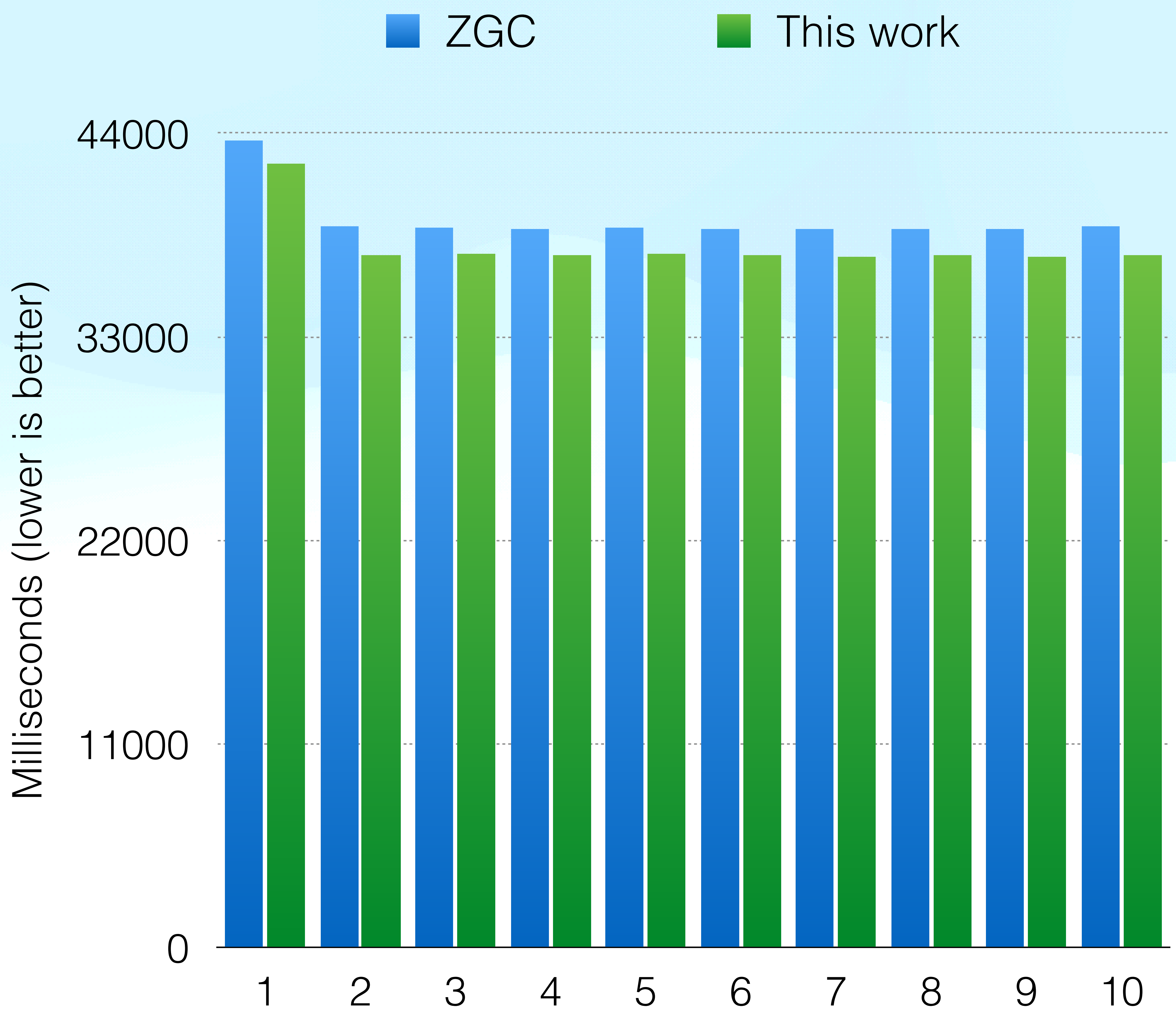
Identify sparsely populated pages

Evacuate sparsely populated pages

Delaying relocation means more objects will be dead at relocation time

Insanely preliminary results

- Scavenging but only on sparsely populated pages
- Young relocation in some graph order
- Reduce CPU usage of ZGC
- Flatten the curve of mem. reclamation
- Challenging to work with the 12 colour bits
- Unclear what the good heuristics are



10 iterations of Spring — ≈3.6% speedup

Some Concluding Remarks

- Load barrier-based designs are everywhere and they can work well
- Generational ZGC complexity is higher than non-generational
- Load barriers are very useful to have in your language
    - Concurrent relocation
    - Locality optimisations
    - Various forms of telemetry
- Would be very nice to close the performance gap between concurrent collectors and throughput collectors because the design is (conceptually) simpler