Generational Z GC

Low latency GC in OpenJDK

Jean-Philippe Bempel
@jpbempel

DATADOG

Agenda

- Why another GC?
- How Z GC works?
    - non generational
    - generational
- How to size & troubleshoot?

Why another GC?

Trade off

- Throughput => ParallelGC
- Small footprint/Small heap => SerialGC
- Low pause time/Latency => ?

(Alternatives: C4, Shenandoah)

History

- Dev started in 2015
- Open sourced dec 2017
- merged in OpenJDK 11 as experimental
- Production ready with OpenJDK 15
- Generational since OpenJDK 21

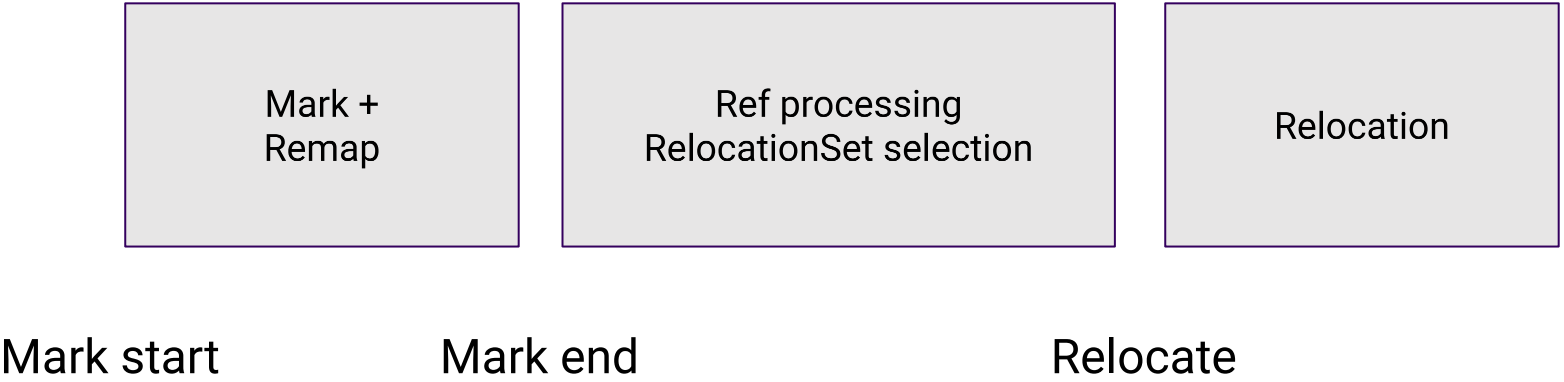
Z GC Characteristics

- Low latency (max pauses <1ms)
- Large heaps (up to 16TB)
- Scalable (not dependant of the size or content of the Heap)

How Z GC works?

non generational

Phases



Mark + Remap

Ref processing
RelocationSet selection

Relocation

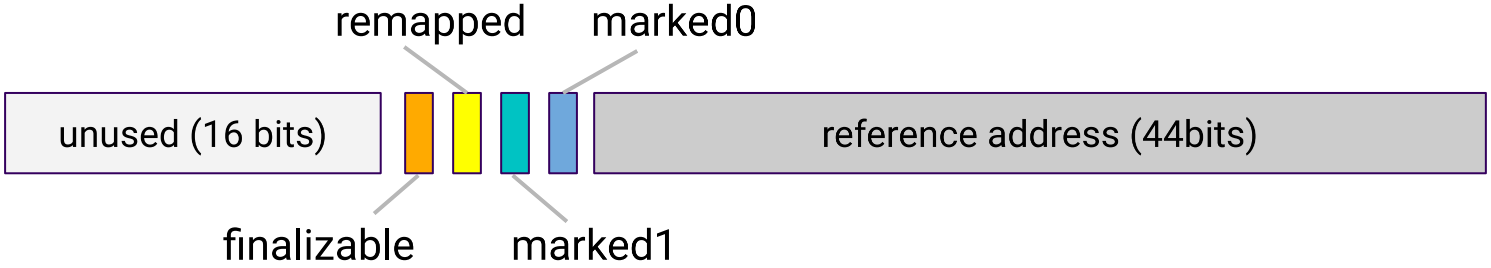
Mark start

Mark end

Relocate

Colored Pointers

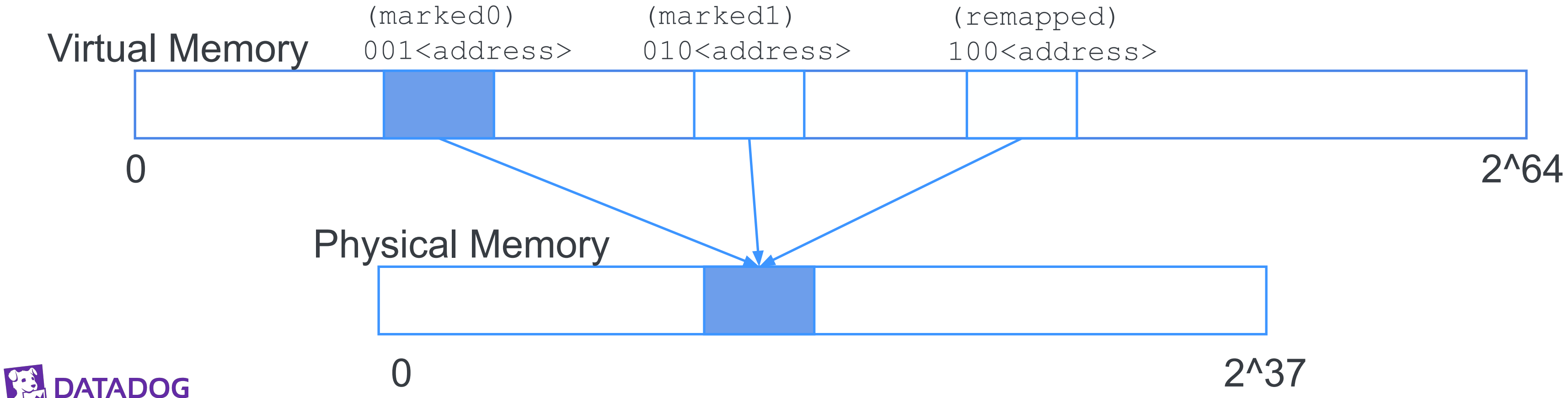
- store metadata in reference addresses
- 44 bits for addressing heap (16TB max)
- compared to global good color in load barriers



DATADOG

Multi mapping

- Same physical page virtual mapped 3 times
- no need to unmask
- reports 3x mem RSS used 🤨



Barriers

- Ensure good invariants before loading ref address
  - Object marked during marking
  - Object relocated/correct new address

- Checking good color (global state) stored in ThreadLocal

- Done before dereferencing (load time)

- Allows JIT optimization (1 load, n deref)

```
mov     rbp,QWORD PTR [r10+0xb8]
test    QWORD PTR [r15+0x28],rbp
jne     slow_path
```

Pauses

Mark start:

- Flip good color (marked0/1)
- reset structures and stats

Mark end:

- Verify Marking ended

Relocation start:

- Flip good color (relocate)
- Update stats

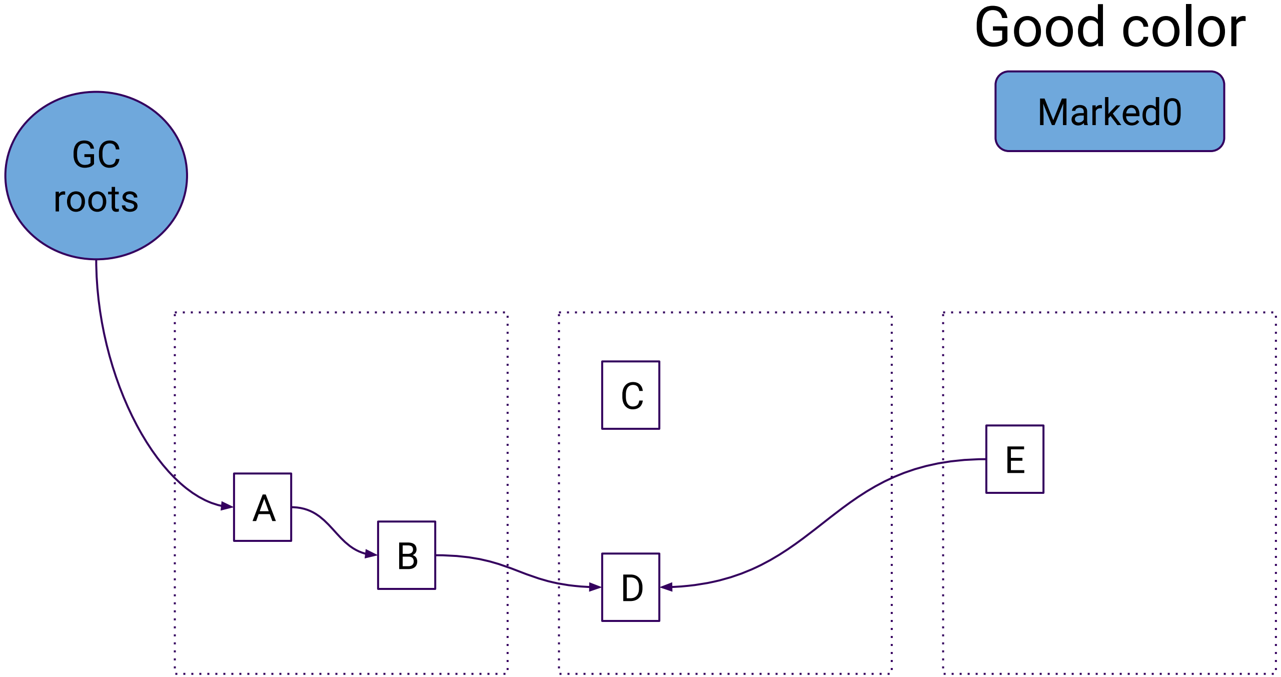
Why pauses are so small?

- Everything else is concurrent:
  - regular GC phases (Mark, reloc, remap)
  - Refs processing (Weak, Soft, Phantom)
  - Class unloading

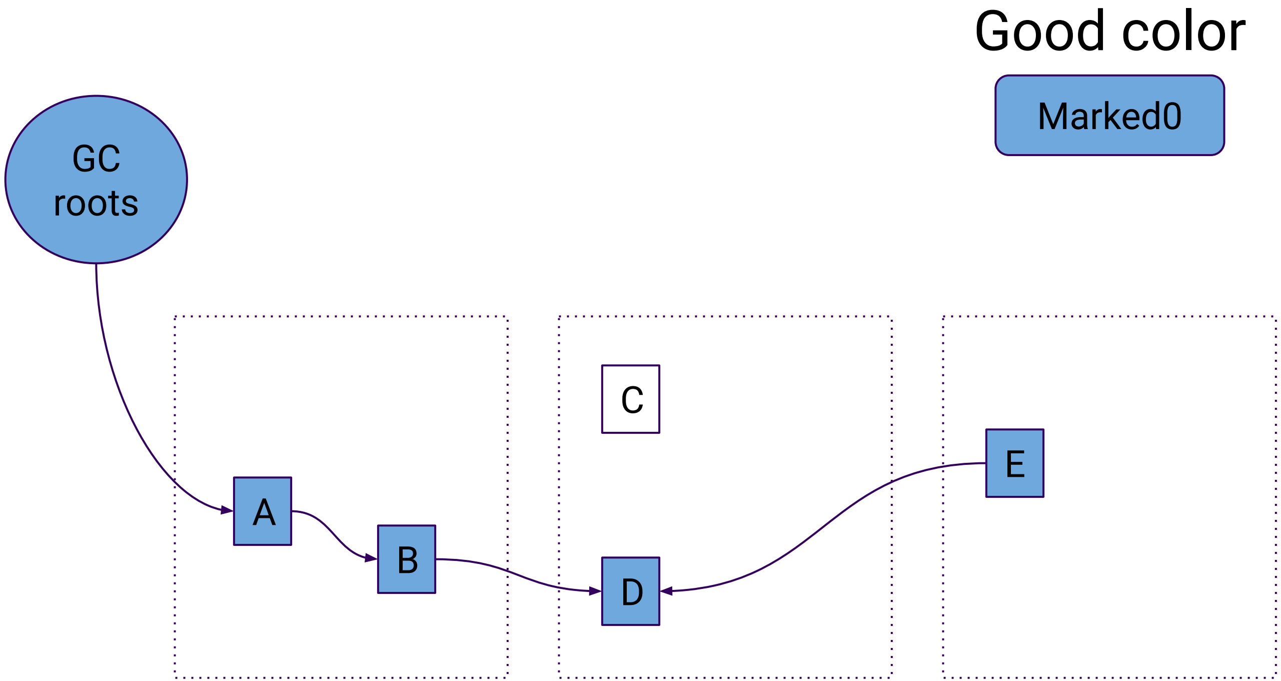
- ThreadLocal Handsakes (JEP 312, JDK 10)

- Concurrent Thread-Stack Processing (JEP 376, JDK16)

Z GC cycle example



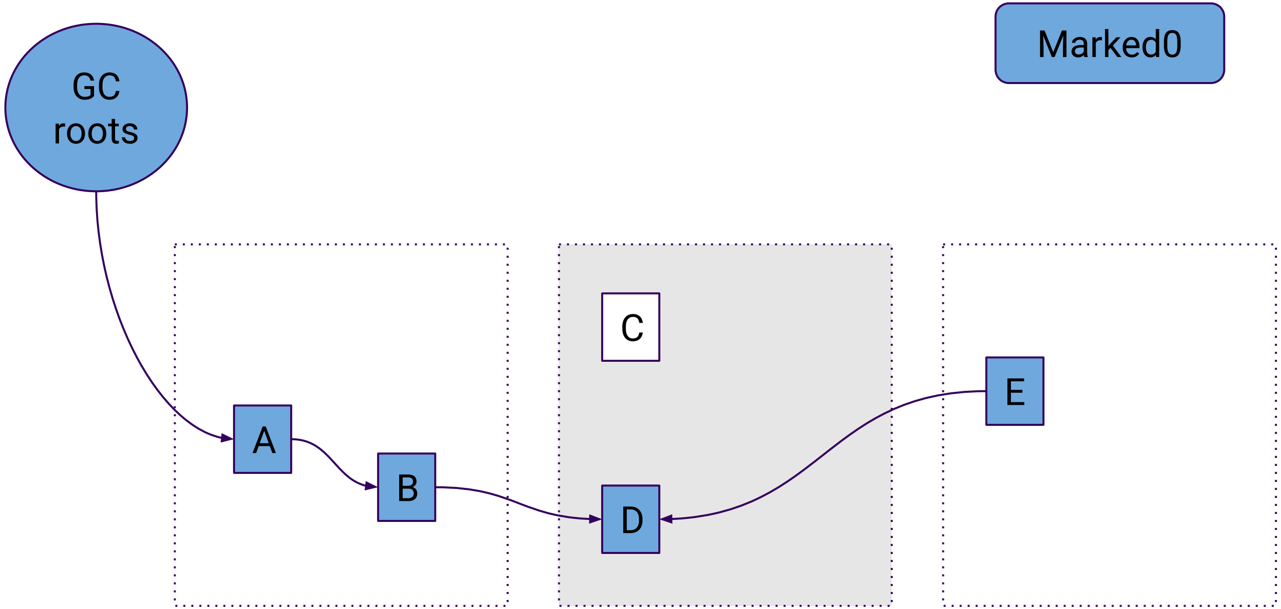
Marking



Page evacuation selection

Good color

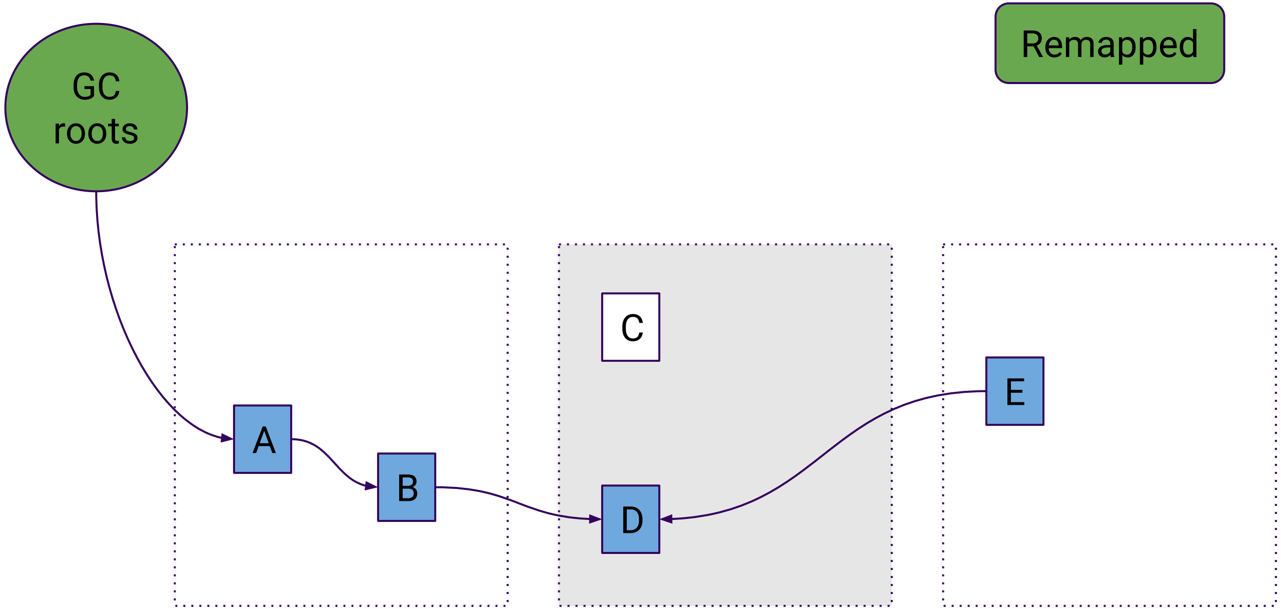
Marked0



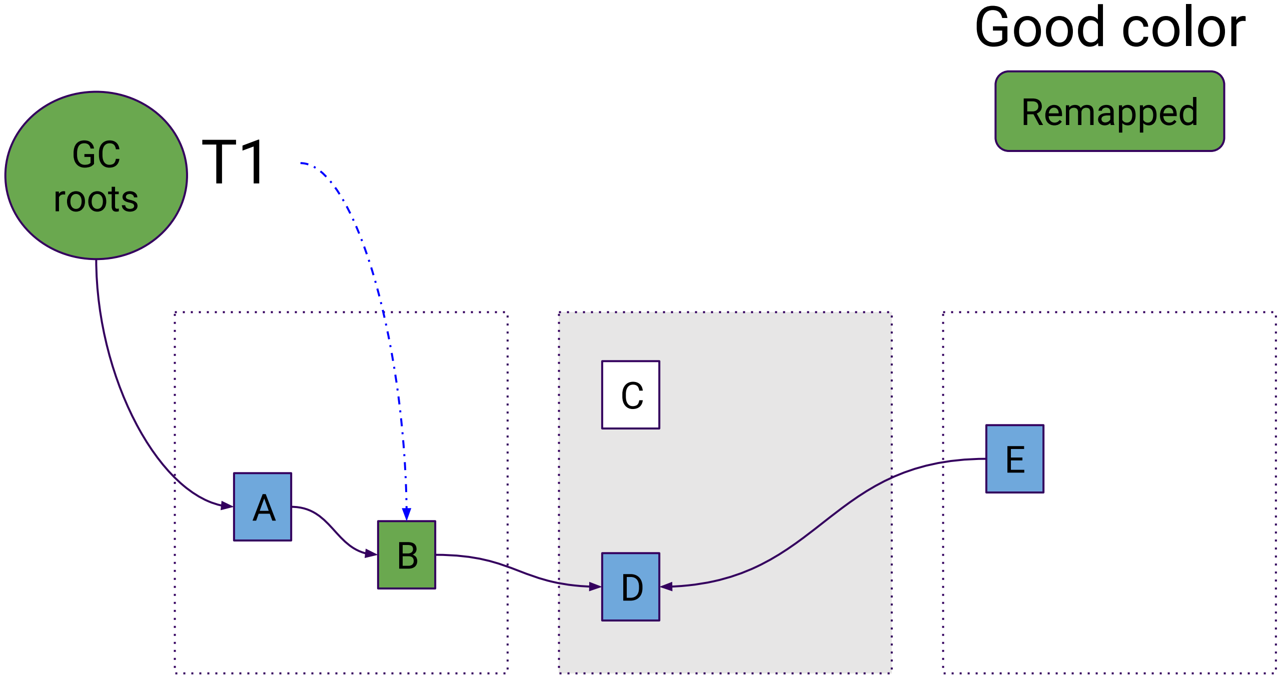
Relocation: flipping good color

Good color

Remapped



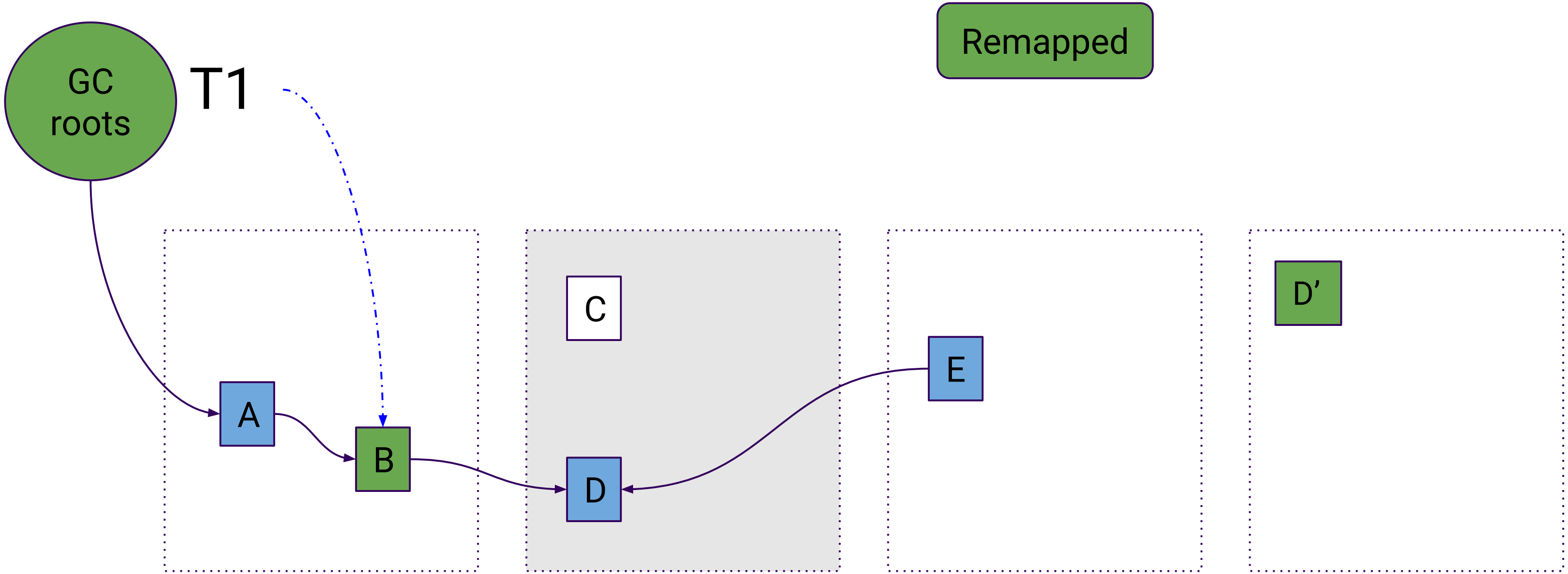
Relocation: User thread access



Relocation: evacuating objects

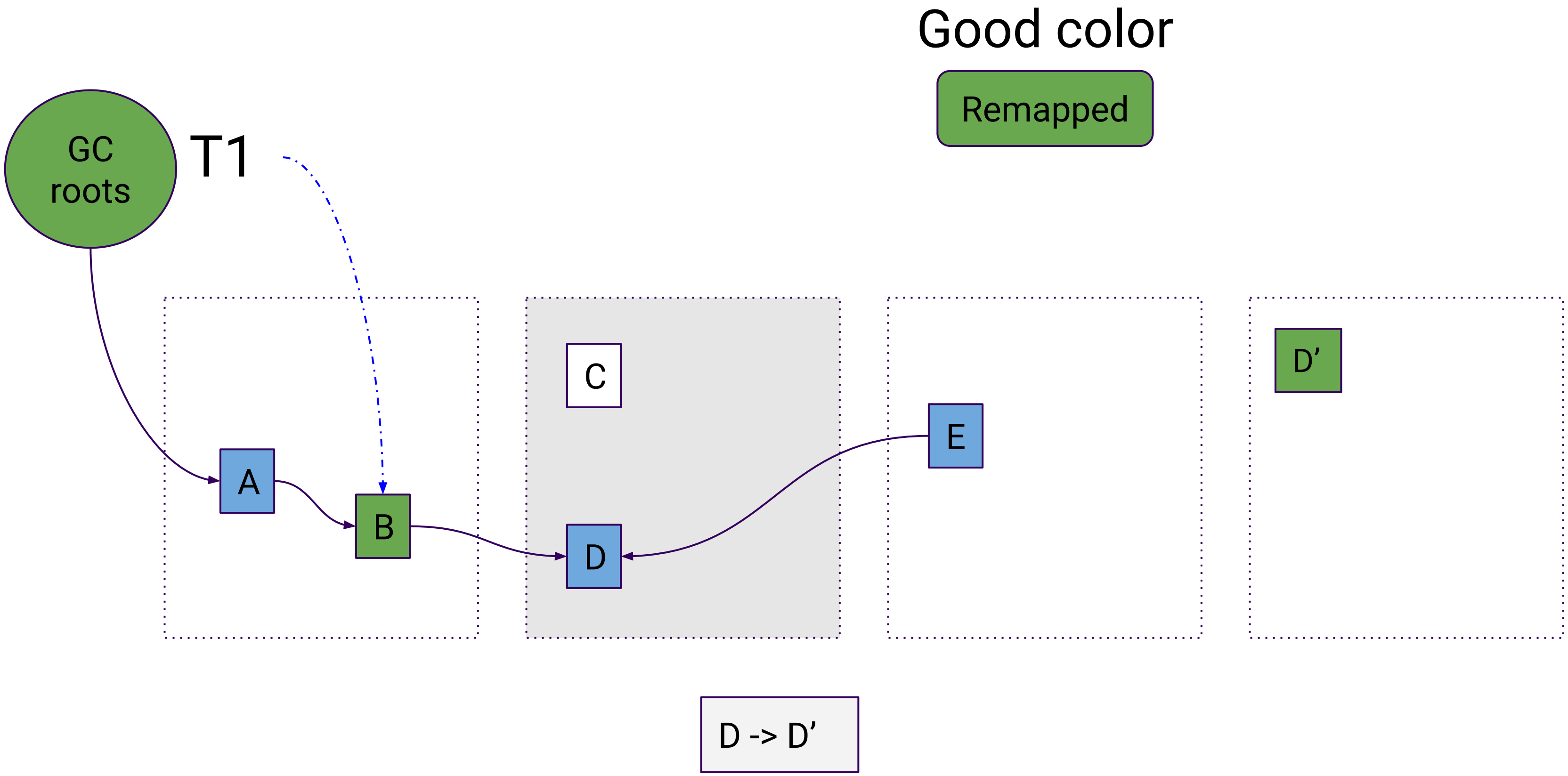
Good color

Remapped



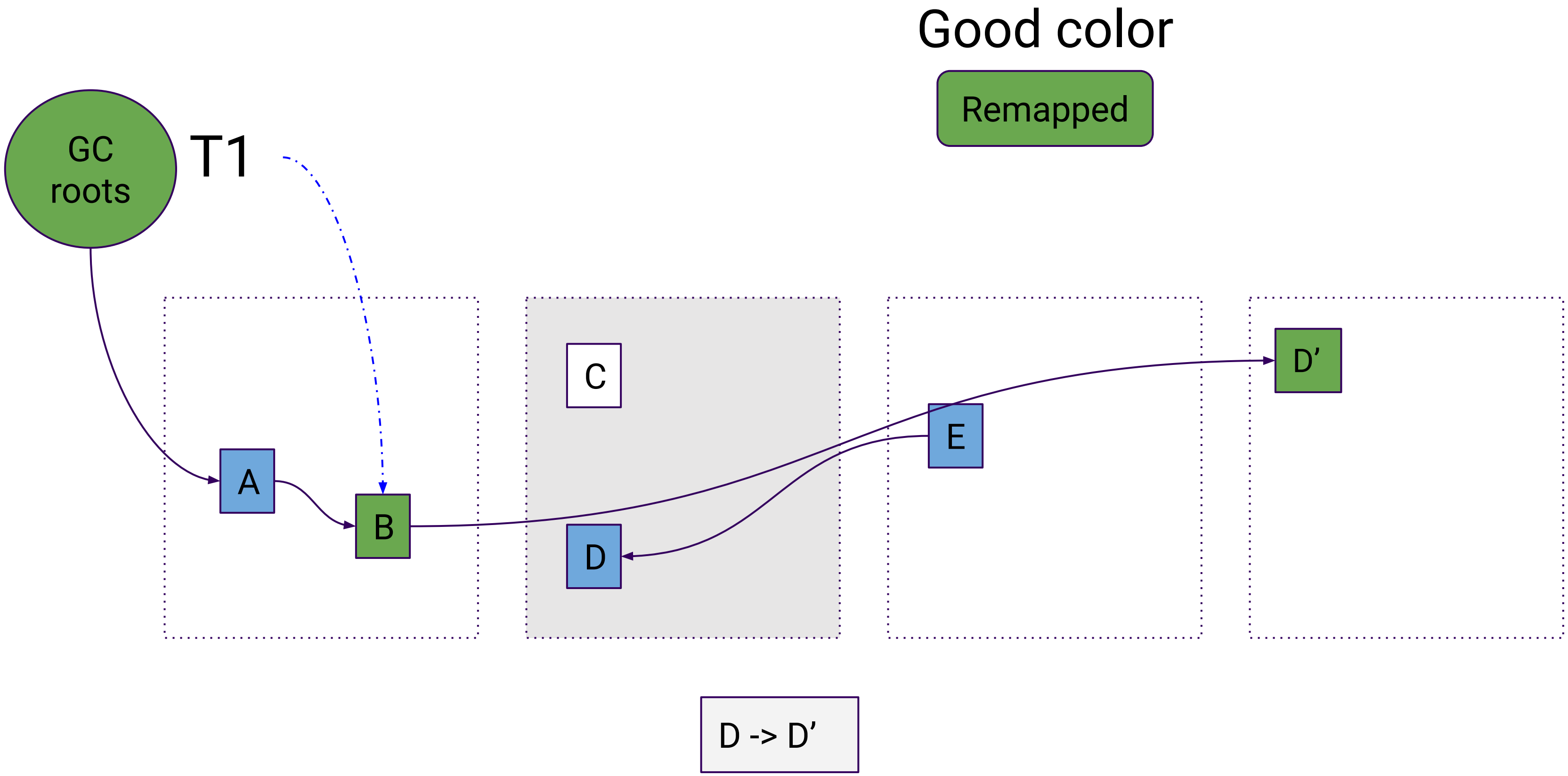
DATADOG

Relocation: forward table



DATADOG

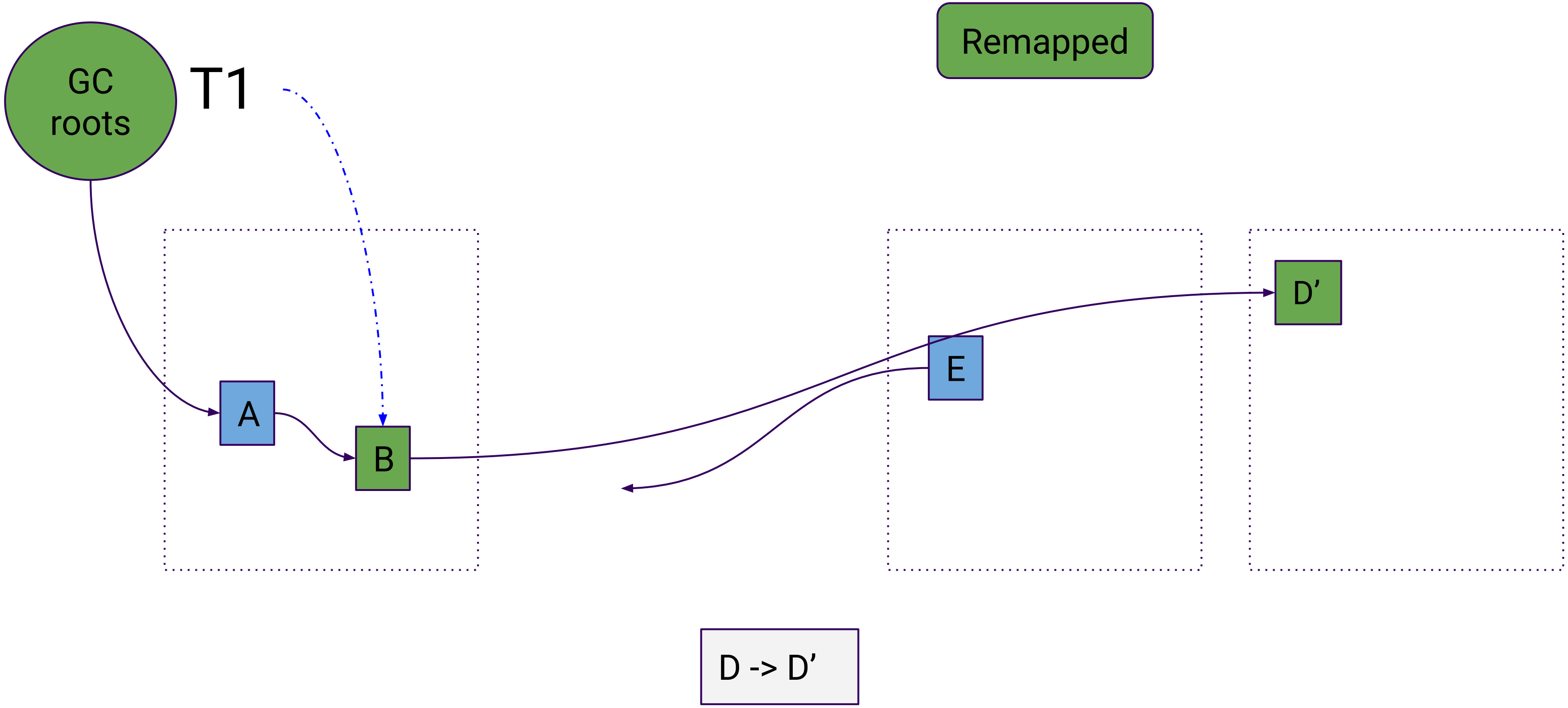
Relocation: fix address



Relocation: Page reclamation

Good color

Remapped



GC roots

T1

A

B

E

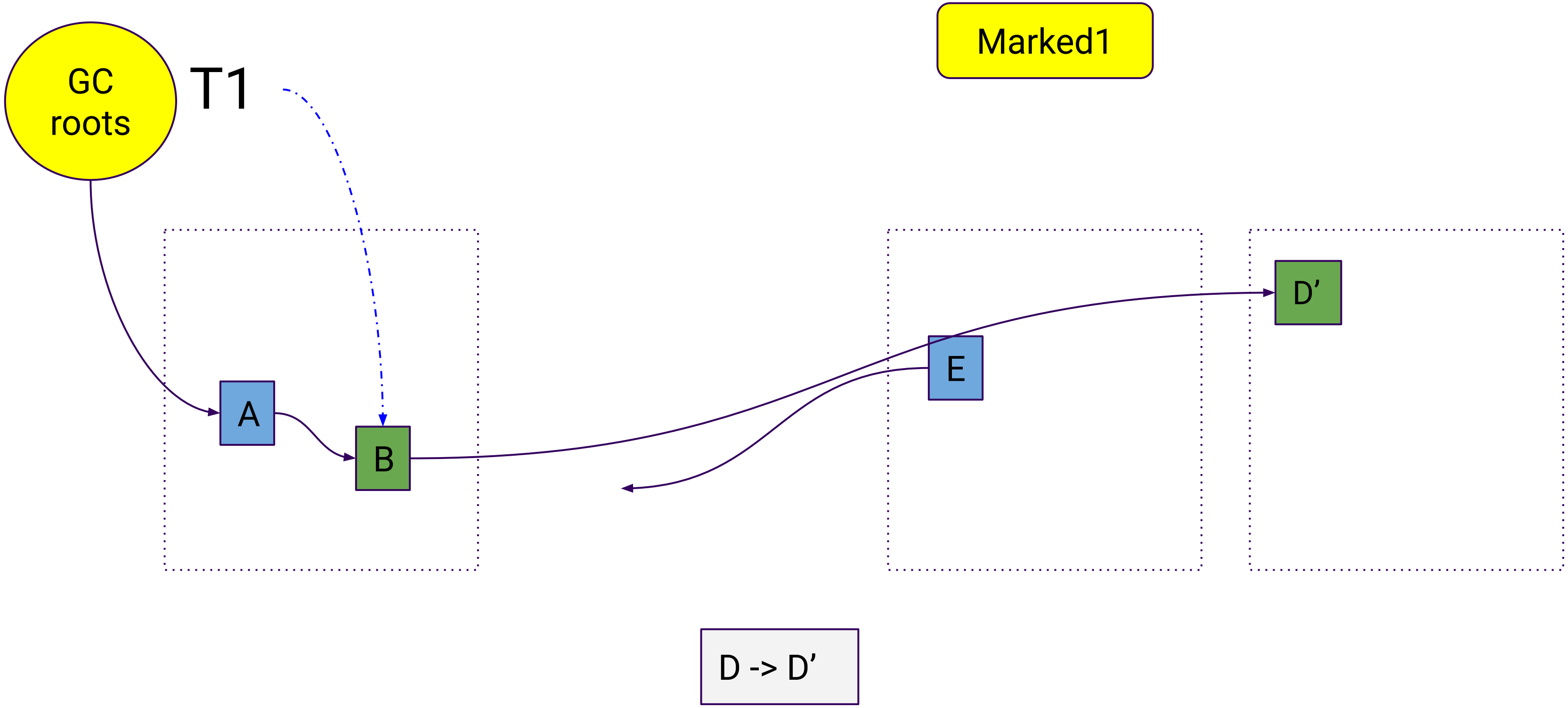
D’

D -> D’

Remap + Mark: flipping good color

Good color

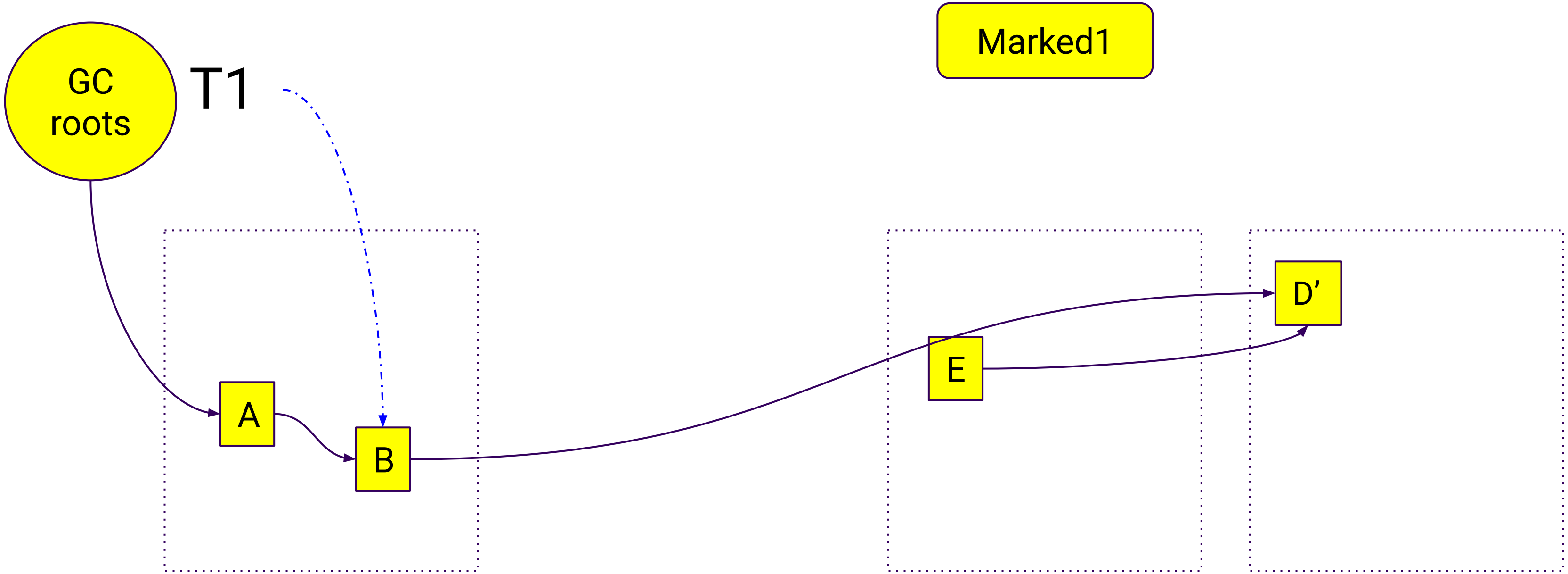
Marked1



End cycle/Begin next cycle: Remap + Mark

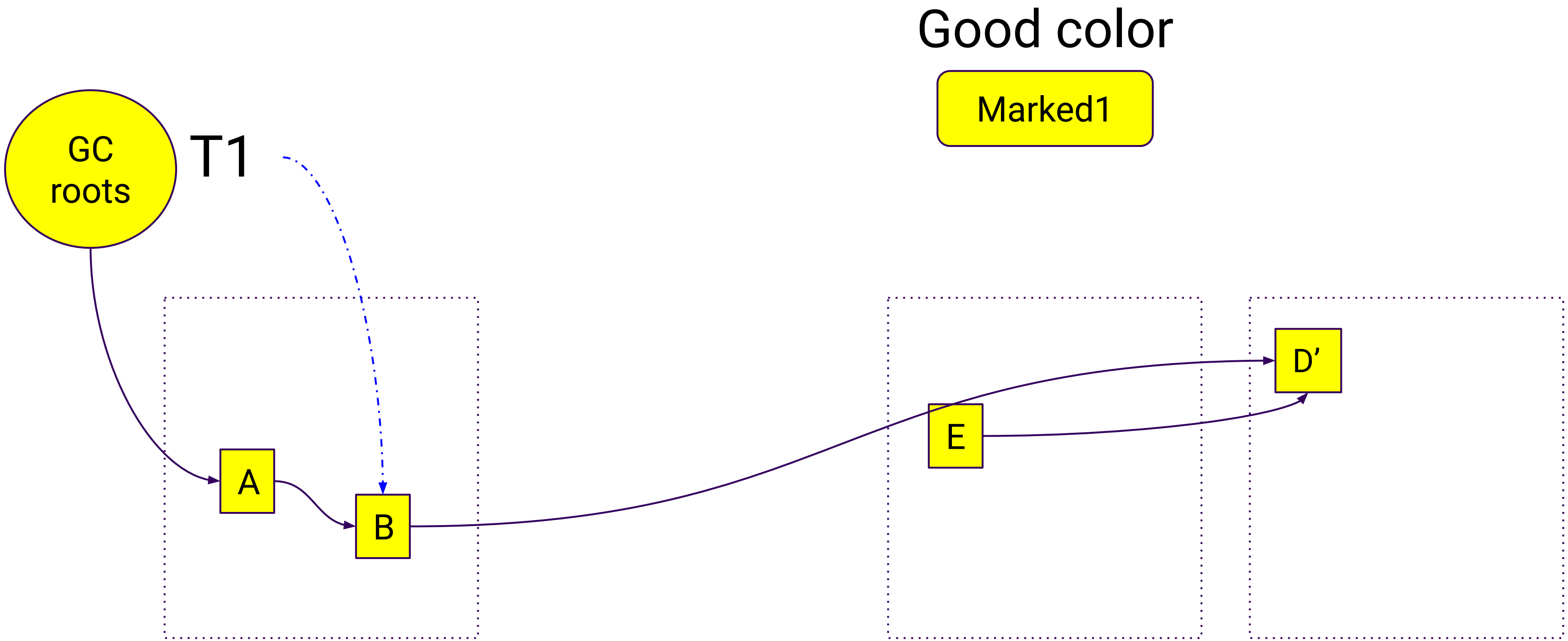
Good color

Marked1



D -> D’

Dropping forwarding table



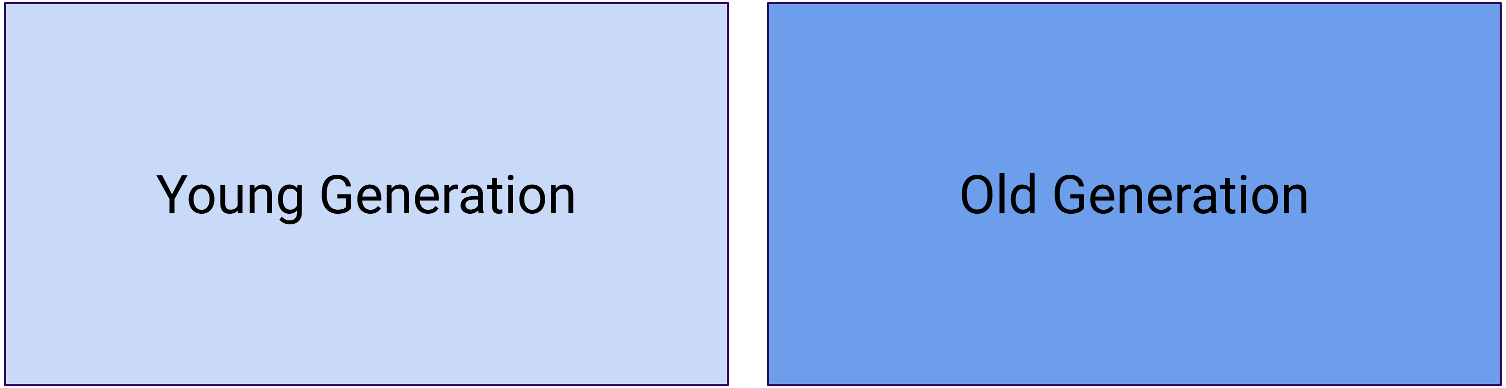
How Z GC works?

generational (JDK 21 -XX:+ZGenerational)

Generational

- Generational GCs are still a good filter for time and CPU

- Heap divided in 2 logical spaces Young & Old

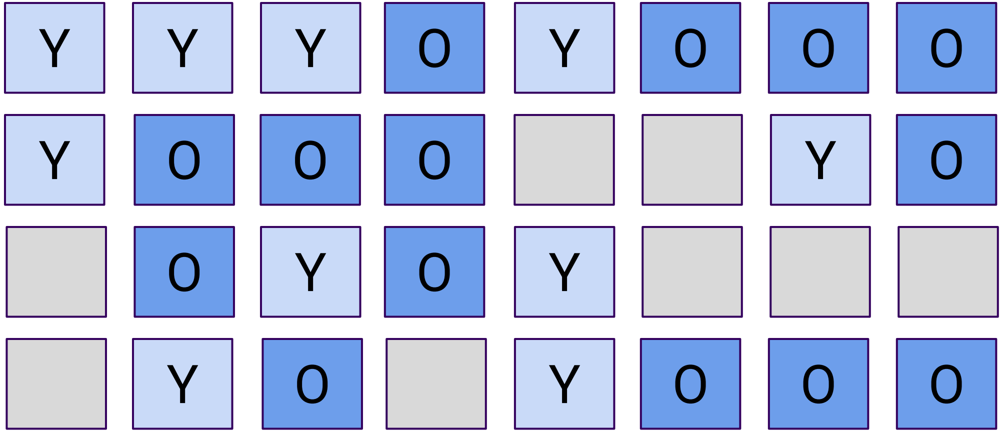


Young Generation

Old Generation

Z GC Generational

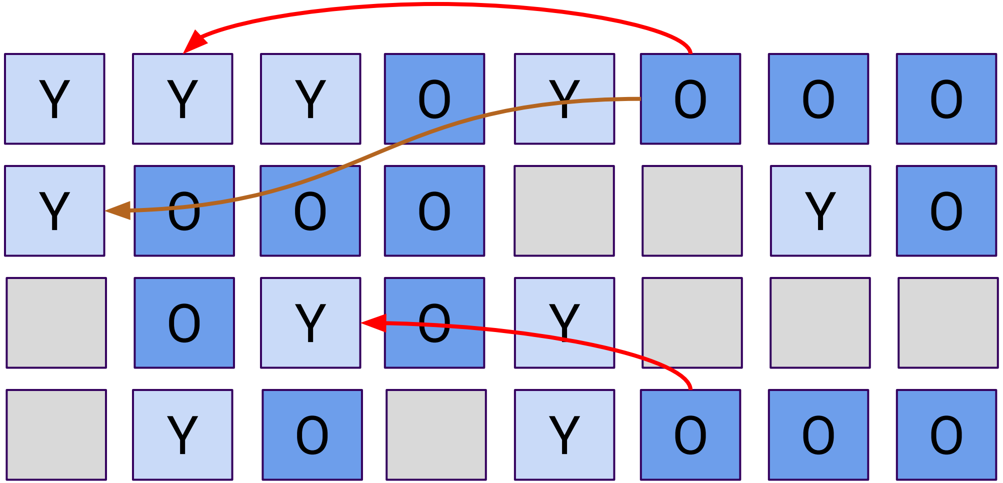
- Each page is assigned to a generation



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

inter generations

- References exist inter-generation
- Store barriers required to track them



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

Store Barriers

- Happen when a reference is written to a field

- If bad color:
  - change color to good one
  - mark object
  - update RememberedSet

- Color the stored reference

Load Barriers

- Happens at load time
- if bad color:
    - change for good one
    - check if relocated/relocate/remap
- uncolor loaded reference

Barrier tricks

- Split responsibility between load and store barriers
- 2 instructions load barriers

```
mov     rbp,QWORD PTR [r10+0xb8]
shr     rbp,0xd
ja      slow_path
```

- uncolor + check good color
- shift value depends on the current color
  and patched on-the-fly

Barrier tricks

- load bits: remapped state (Young & old)
  load=0001 shr=13
  load=1000 shr=16
  `ja` jumps only if CF=0 && ZF=0

```
mov     rbp,QWORD PTR [r10+0xb8]
shr     rbp,0xd
ja      slow_path
```

- store bits:
  - Marked (Young & old)
  - Finalizable
  - RememberedSet

```
test    DWORD PTR [rsi+0x10],0xeae0
jne     slow_path
mov     rdx,rax
shl     rdx,0xd
or      rdx,0x1510
mov     QWORD PTR [rsi+0x10],rdx
```

unused (2 bits)	reference address (46 bits)	load (4 bits)	store (8 bits)	unused (4 bits)

Other Changes

- No more Multi-mapping
- Aging in place (no evac for Young region -> Survivor)
- Relocation in-place (same region)
- Large Objects reclaimed in minor GC

How to size & troubleshoot?

Heap Sizing

- Like for any GC, the more the better
- More true for Concurrent GC (race with allocation rate)
- cores/threads help also significantly
- Generational GC helps to reduce the need for more memory amd or more CPU

SoftMaxHeapSize

- JVM option introduced in JDK 13

- Allow to reduce Heap footprint:
  - Most of the time 2GB
  - Occasionally spikes to 5GB
  - => `-XX:SoftMaxHeapSize=2G`
  - Above the limit, triggers GC more frequently
  - Uncommits OS pages once usage below the limit

Allocation Stalls

- What happens if Allocation rate > GC reclamation?

- Allocating thread will be stalled:
  - Allocation fails
  - Triggers GC
  - Wait for page to be reclaimed to resume allocation

- Any threads trying to allocate can be stalled

Allocation Stalls Monitoring

- GC logs:

```
[254.528s][info][gc] Allocation Stall (http-nio-8080-exec-4) 36.329ms

[254.528s][info][gc] Allocation Stall (StatsD-Sender-1) 28.825ms

[254.528s][info][gc] Relocation Stall (http-nio-8080-exec-9) 0.423ms
```

```
[254.531s][info][gc,alloc] GC(191) y:                        Mark Start        Mark End       Relocate Start    Relocate End

[254.531s][info][gc,alloc] GC(191) y: Allocation Stalls:         0                10                10                0
```

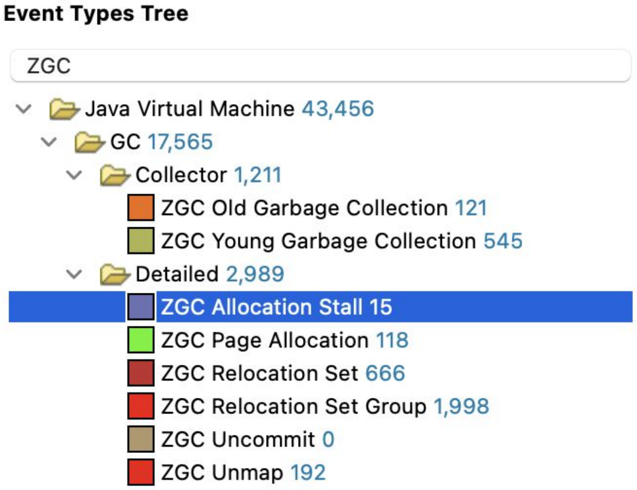
Allocation Stalls Monitoring

- JFR
  jdk.ZAllocationStall event (enabled by default)

```
$ jfr print --events jdk.ZAllocationStall  petclinic-benchmark-profile.jfr
jdk.ZAllocationStall {
  startTime = 10:58:38.982 (2024-05-28)
  duration = 55.1 ms
  type = "Small"
  size = 2.0 MB
  eventThread = "http-nio-8080-exec-8" (javaThreadId = 392)
}
```

Allocation Stalls Monitoring

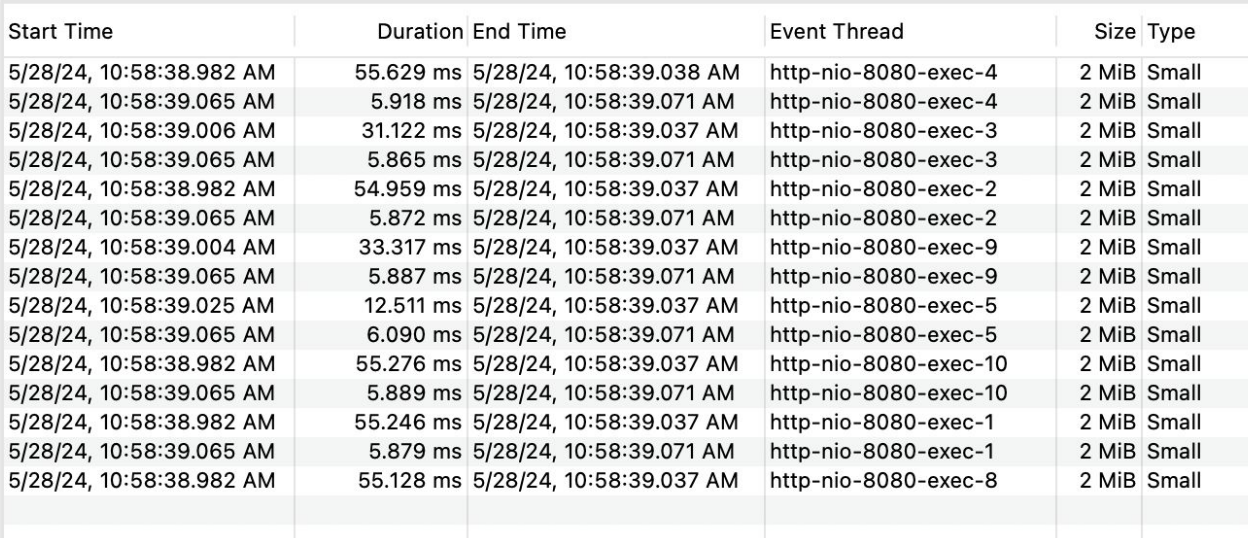
- JMC



Event Types Tree

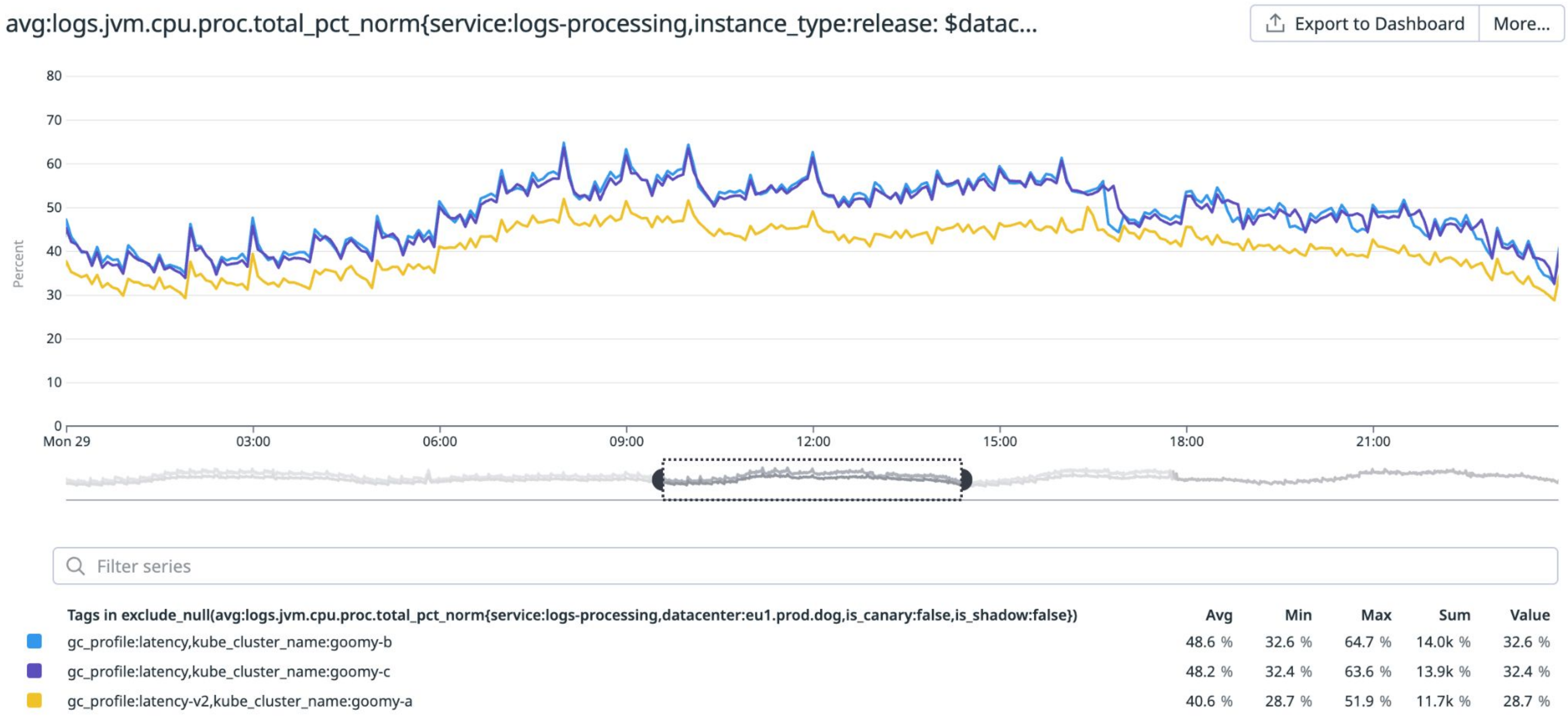
ZGC

- Java Virtual Machine 43,456
  - GC 17,565
    - Collector 1,211
      - 🟧 ZGC Old Garbage Collection 121
      - 🟩 ZGC Young Garbage Collection 545
    - Detailed 2,989
      - 🟪 ZGC Allocation Stall 15
      - 🟩 ZGC Page Allocation 118
      - 🟥 ZGC Relocation Set 666
      - 🟥 ZGC Relocation Set Group 1,998
      - 🟫 ZGC Uncommit 0
      - 🟥 ZGC Unmap 192



Start Time	Duration	End Time	Event Thread	Size	Type
5/28/24, 10:58:38.982 AM	55.629 ms	5/28/24, 10:58:39.038 AM	http-nio-8080-exec-4	2 MiB	Small
5/28/24, 10:58:39.065 AM	5.918 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-4	2 MiB	Small
5/28/24, 10:58:39.006 AM	31.122 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-3	2 MiB	Small
5/28/24, 10:58:39.065 AM	5.865 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-3	2 MiB	Small
5/28/24, 10:58:38.982 AM	54.959 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-2	2 MiB	Small
5/28/24, 10:58:39.065 AM	5.872 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-2	2 MiB	Small
5/28/24, 10:58:39.004 AM	33.317 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-9	2 MiB	Small
5/28/24, 10:58:39.065 AM	5.887 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-9	2 MiB	Small
5/28/24, 10:58:39.025 AM	12.511 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-5	2 MiB	Small
5/28/24, 10:58:39.065 AM	6.090 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-5	2 MiB	Small
5/28/24, 10:58:38.982 AM	55.276 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-10	2 MiB	Small
5/28/24, 10:58:39.065 AM	5.889 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-10	2 MiB	Small
5/28/24, 10:58:38.982 AM	55.246 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-1	2 MiB	Small
5/28/24, 10:58:39.065 AM	5.879 ms	5/28/24, 10:58:39.071 AM	http-nio-8080-exec-1	2 MiB	Small
5/28/24, 10:58:38.982 AM	55.128 ms	5/28/24, 10:58:39.037 AM	http-nio-8080-exec-8	2 MiB	Small

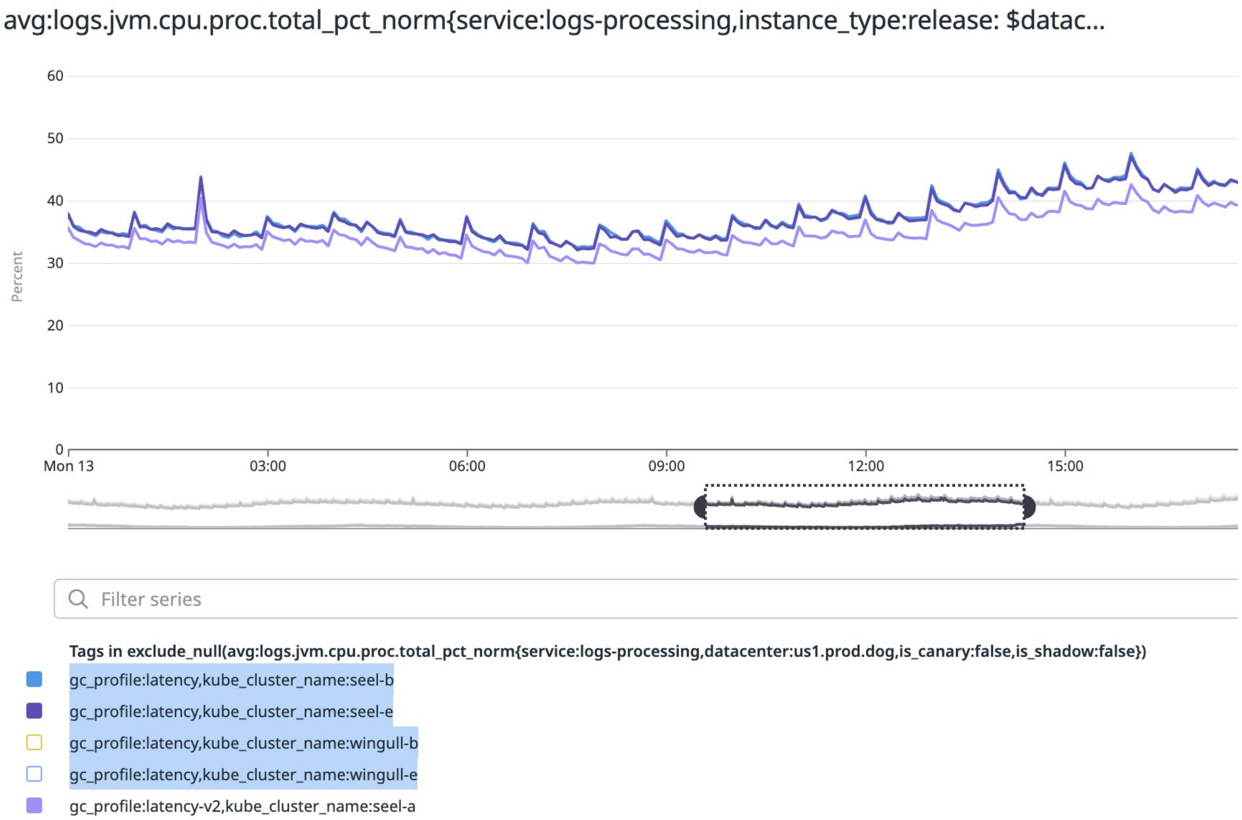
RetEx: @ Datadog



avg:logs.jvm.cpu.proc.total_pct_norm{service:logs-processing,instance_type:release: $datac...

	Tags in exclude_null(avg:logs.jvm.cpu.proc.total_pct_norm{service:logs-processing,datacenter:eu1.prod.dog,is_canary:false,is_shadow:false})	Avg	Min	Max	Sum	Value
🟦	gc_profile:latency,kube_cluster_name:goomy-b	48.6 %	32.6 %	64.7 %	14.0k %	32.6 %
🟪	gc_profile:latency,kube_cluster_name:goomy-c	48.2 %	32.4 %	63.6 %	13.9k %	32.4 %
🟨	gc_profile:latency-v2,kube_cluster_name:goomy-a	40.6 %	28.7 %	51.9 %	11.7k %	28.7 %

RetEx: @ Datadog



avg:logs.jvm.cpu.proc.total_pct_norm{service:logs-processing,instance_type:release: $datac...

Filter series

Tags in exclude_null(avg:logs.jvm.cpu.proc.total_pct_norm{service:logs-processing,datacenter:us1.prod.dog,is_canary:false,is_shadow:false})

🟦 gc_profile:latency,kube_cluster_name:seel-b

🟪 gc_profile:latency,kube_cluster_name:seel-e

□ gc_profile:latency,kube_cluster_name:wingull-b

□ gc_profile:latency,kube_cluster_name:wingull-e

🟪 gc_profile:latency-v2,kube_cluster_name:seel-a

References

- Z GC OpenJDK wiki
- Java’s Highly Scalable Low-Latency Garbage Collector : ZGC
- JEP 333: Z GC: A Scalable Low Latency Garbage Collector
- JEP 439: Generational Z GC
- Adventures in Concurrent Garbage Collector
- Throughput Analysis of Safepoint-attached Barriers in a Low Latency GC
- JVMLS: Generational GC and Beyond
- Introducing Z GC
- JEP 312: Thread-Local Handshakes
- JEP 376: Z GC: Concurrent Thread-Stack Processing

Q&A

@jpbempel

Bonus: Late Barrier Expansion

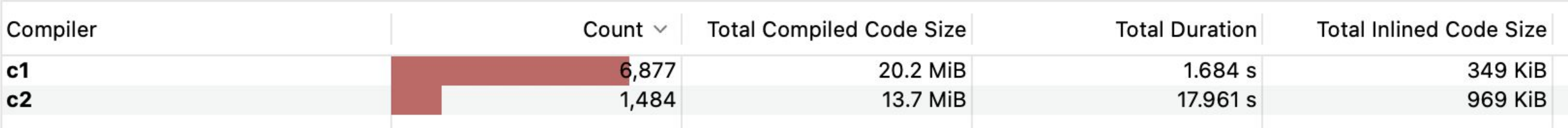
- Barrier code was historically inserted directly into JIT’s IR
- Benefits from JIT’s optimizations
- But consume significant time in CPU
- Hard to maintain

Bonus: Late Barrier Expansion

- Hard coded barriers by cpu arch (JDK-8230565)

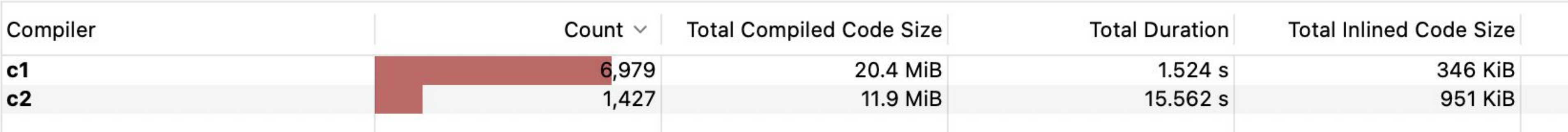
- WIP for G1 (JEP 475)

  petclinic startup with G1 on JDK 17:



Compiler	Count	Total Compiled Code Size	Total Duration	Total Inlined Code Size
c1	6,877	20.2 MiB	1.684 s	349 KiB
c2	1,484	13.7 MiB	17.961 s	969 KiB

  ptclinic startup with Z GC on JDK 17:



Compiler	Count	Total Compiled Code Size	Total Duration	Total Inlined Code Size
c1	6,979	20.4 MiB	1.524 s	346 KiB
c2	1,427	11.9 MiB	15.562 s	951 KiB