

Oracle Code One 2019

G1 and ZGC: A Look into the Progress of Garbage Collection in Java

[DEV4459]

Mikael Vidstedt, Director, Java Virtual Machine
Paul Su, Director, JVM Garbage Collection

Oracle
September 16, 2019

Safe Harbor

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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle's future plans, expectations, beliefs, intentions and prospects are "forward-looking statements" and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle's Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading "Risk Factors." These filings are available on the SEC's website or on Oracle's website at <http://www.oracle.com/investor>. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.

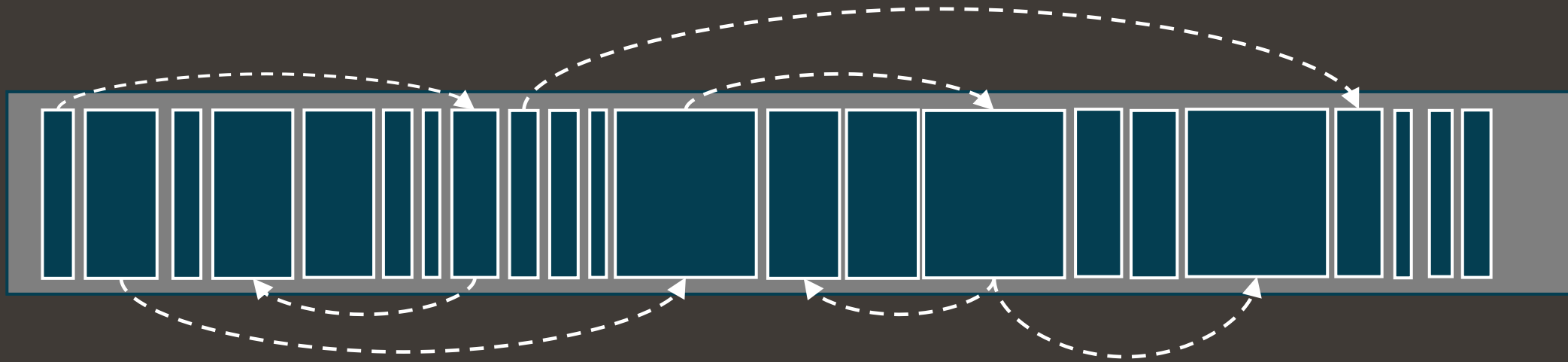
Agenda

- Introduction to garbage collection
- GC tradeoffs
- G1
- ZGC

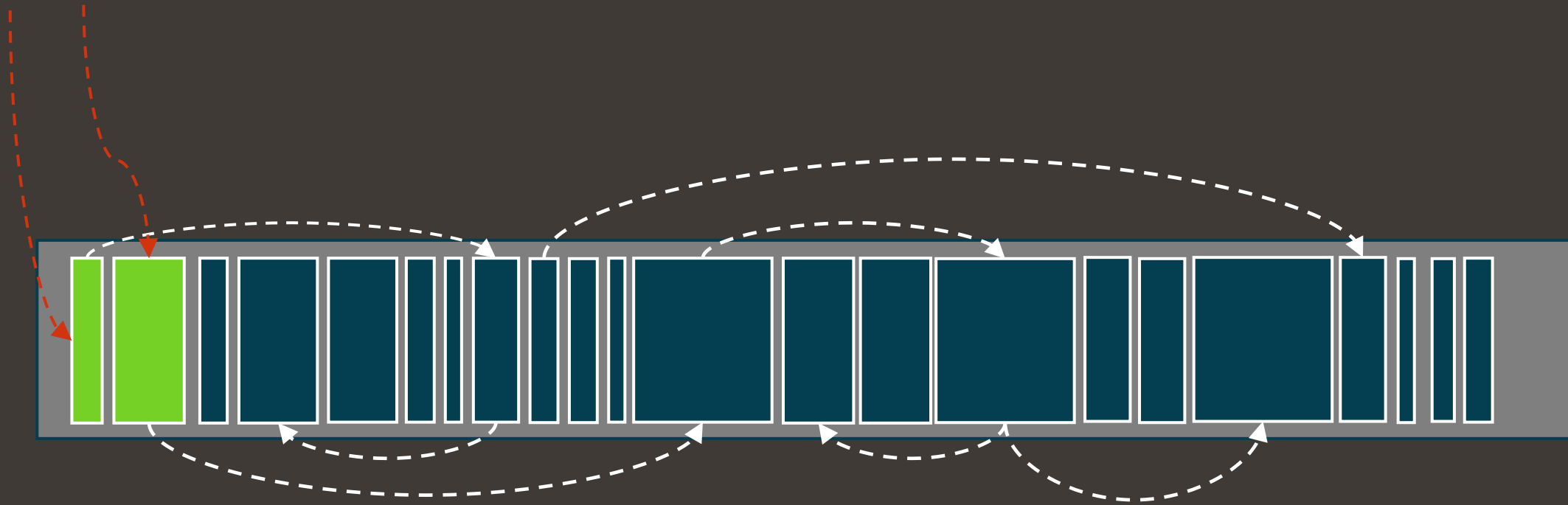
Introduction to Garbage Collection



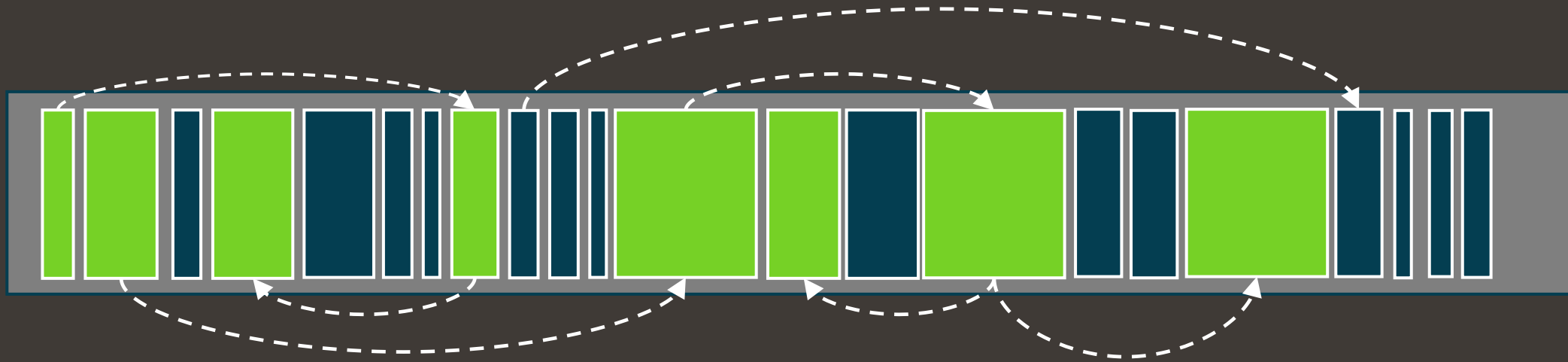
Introduction to Garbage Collection



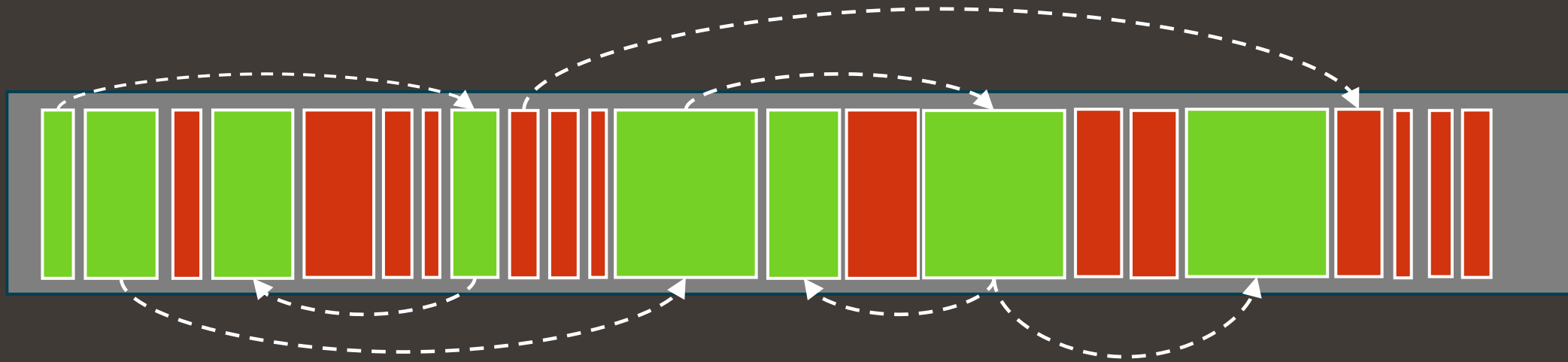
Introduction to Garbage Collection



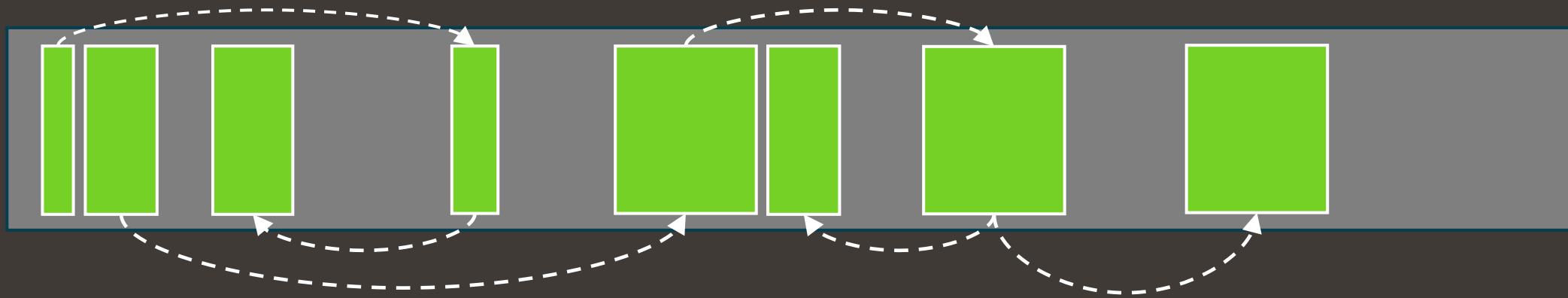
Introduction to Garbage Collection



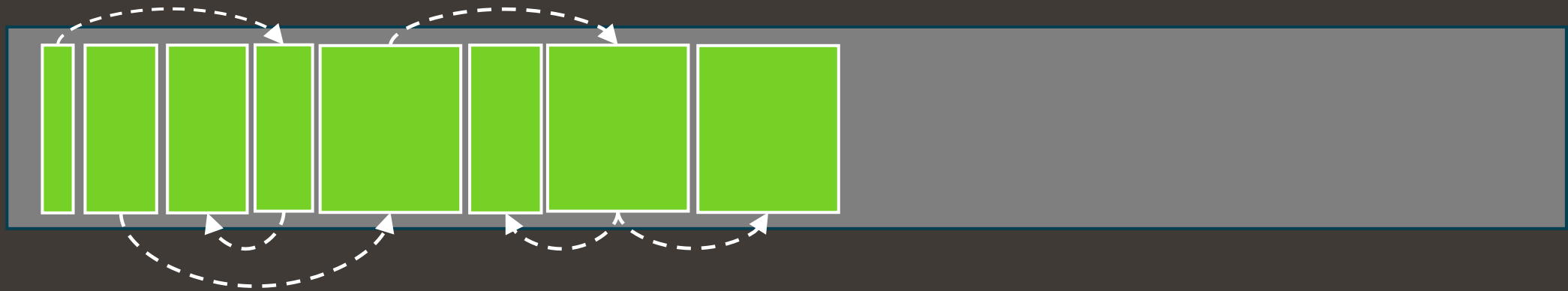
Introduction to Garbage Collection



Introduction to Garbage Collection



Introduction to Garbage Collection



Collectors in the JVM

GC	Optimized For
Serial	Memory Footprint
Parallel	Throughput
G1	Throughput/Latency Balance
CMS	Latency
ZGC	Scalability/Low Latency

G1

The G1 Garbage Collector

The default garbage collector since JDK 9

- First introduced in 6u14
- Supported since 7u4

The goal: throughput *and* low latency

The default pause target for G1 is 200 milliseconds

- Higher pause goal → more throughput, higher latency
- Lower pause goal → less throughput, lower latency

G1 - Generational region-based memory management

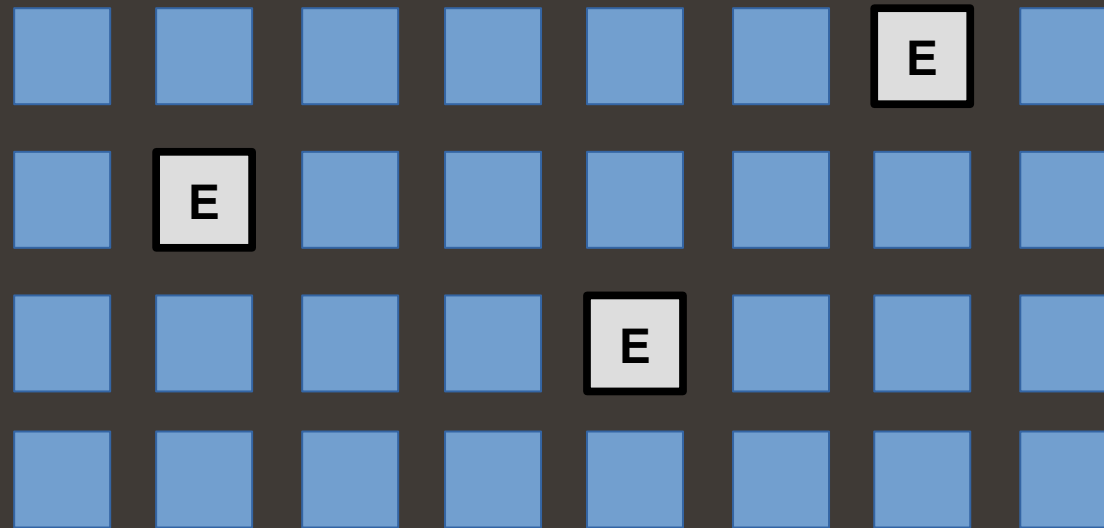
- The heap is split into multiple regions
- Region size depends on heap size, e.g. 2 MB for 4 GB heap



Heap

G1 - Generational region-based memory management

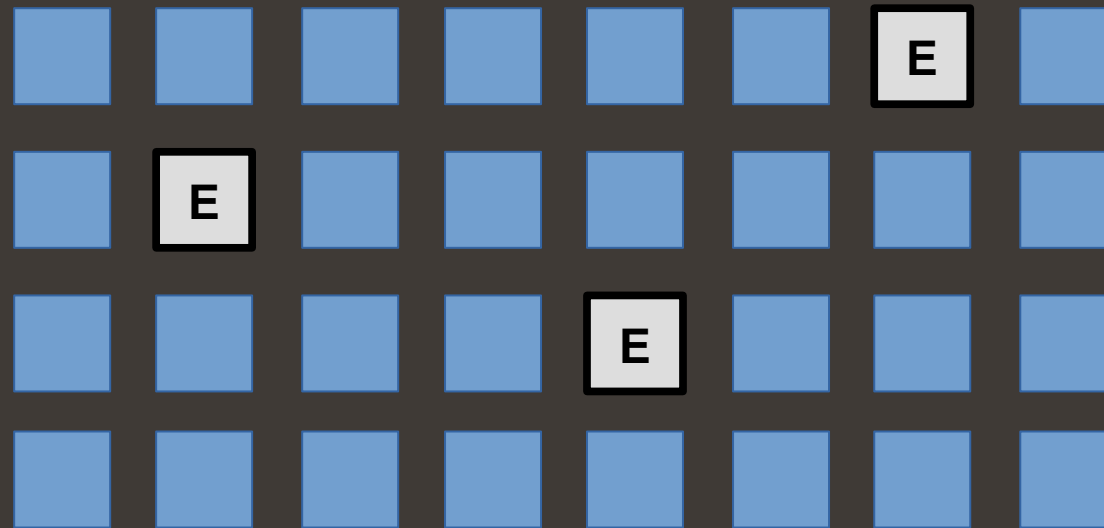
- New objects are allocated into eden (E) regions



Heap

G1 - Generational region-based memory management

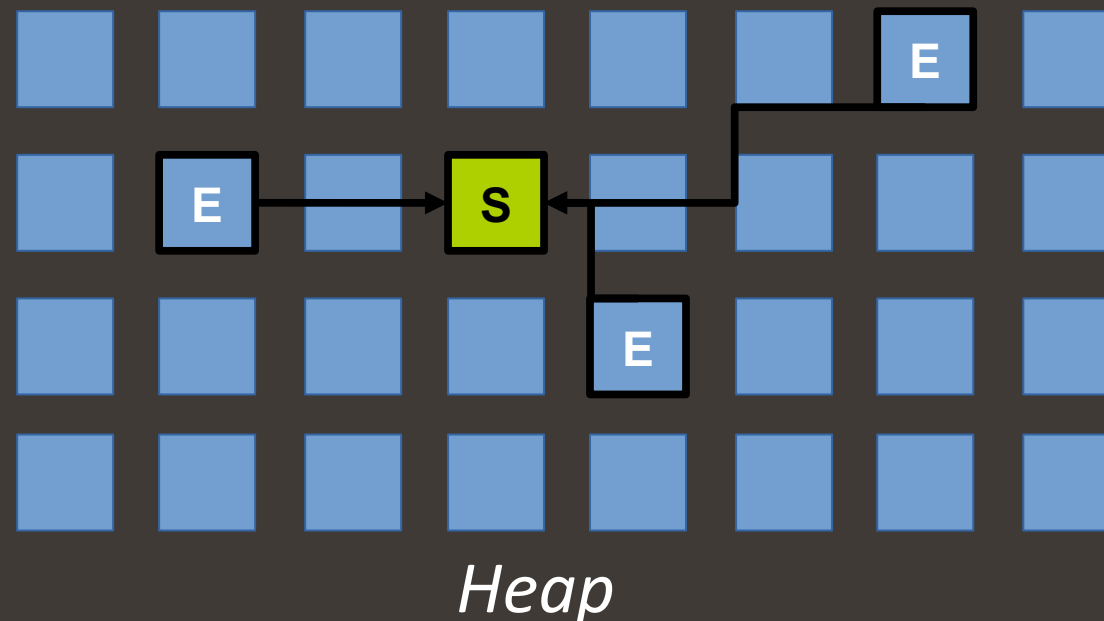
- A *young collection* happens after a number of eden regions have been allocated



Heap

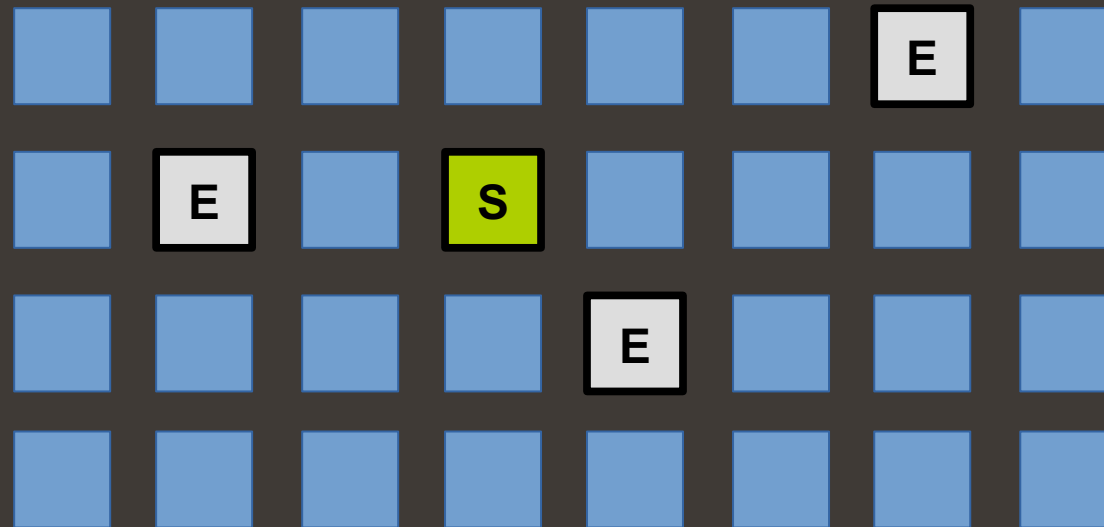
G1 - Generational region-based memory management

- Young collections compactly copy live objects in eden regions to survivor regions (S)



G1 - Generational region-based memory management

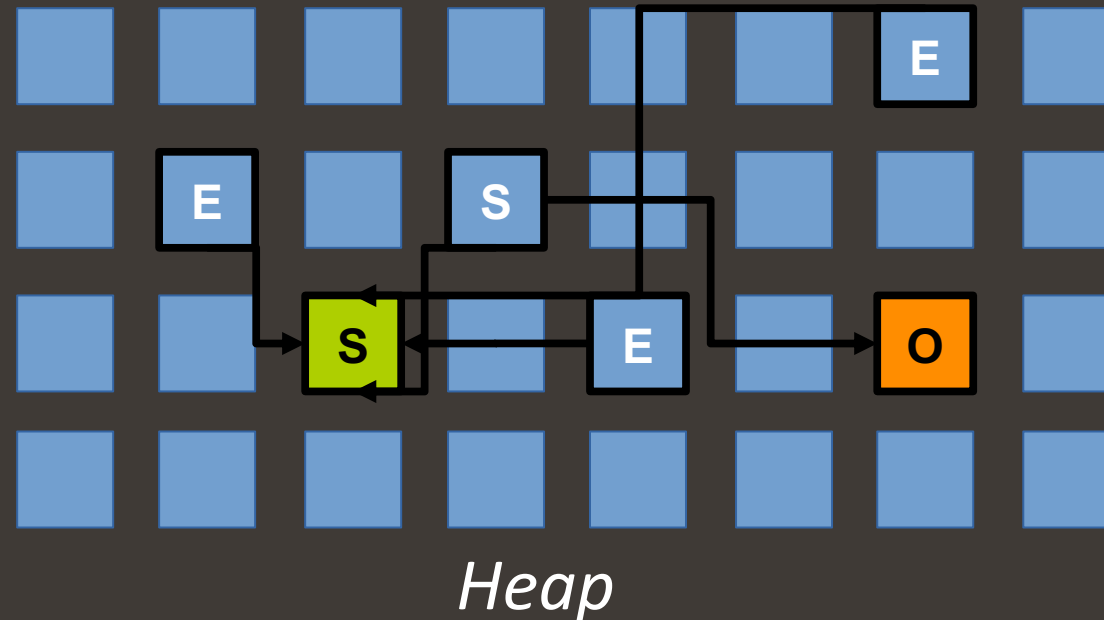
- Objects will then continue to be allocated in eden regions



Heap

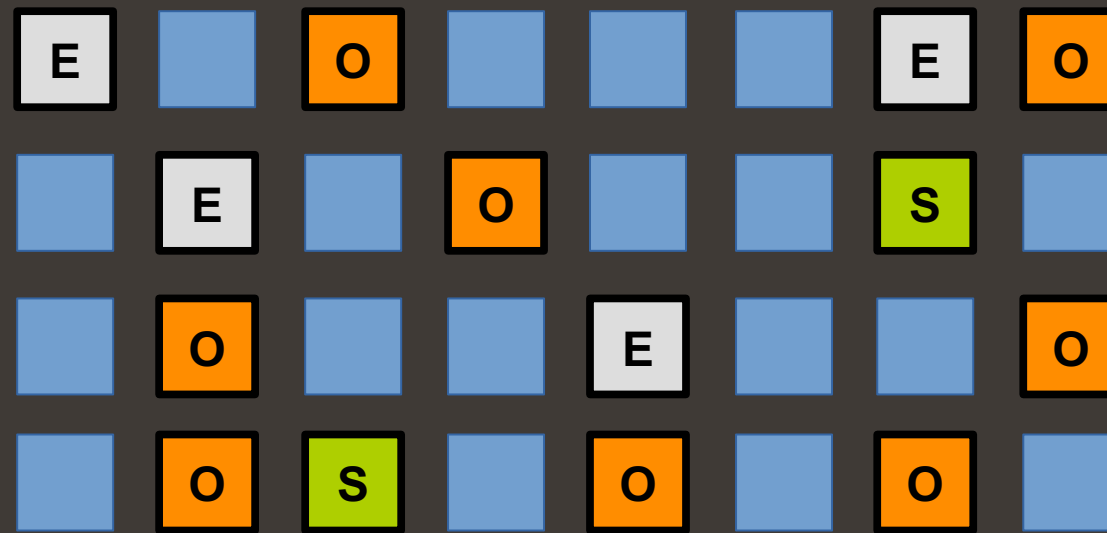
G1 - Generational region-based memory management

- If objects survive multiple young collections, then they are compactly copied into an old region (O)



G1 - Generational region-based memory management

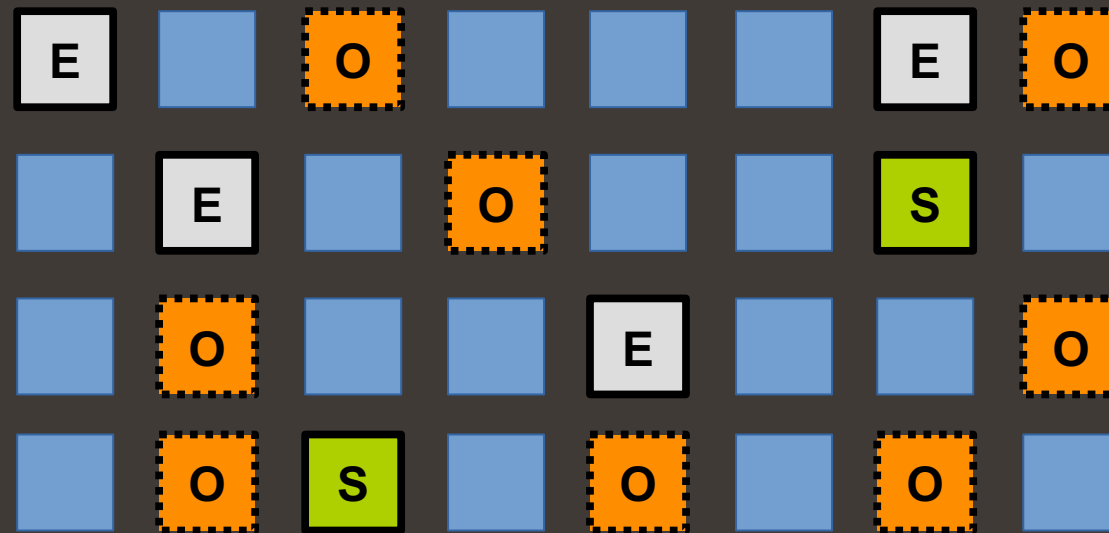
- After a while the heap fills up with eden, survivor and old regions



Heap

G1 - Generational region-based memory management

- All live objects in old regions are then marked *concurrently*
- The Java application is *not* stopped

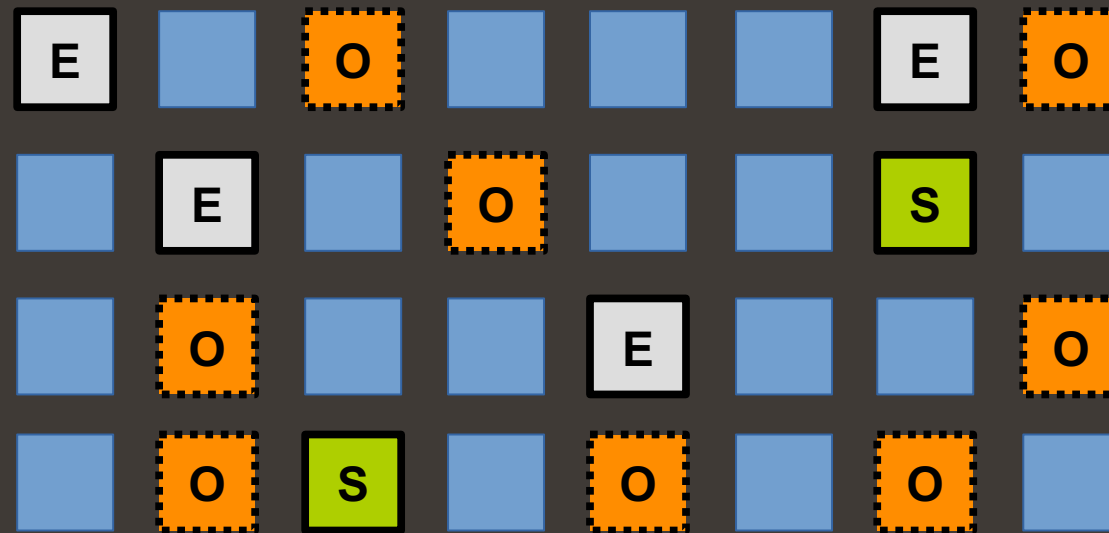


Heap



G1 - Generational region-based memory management

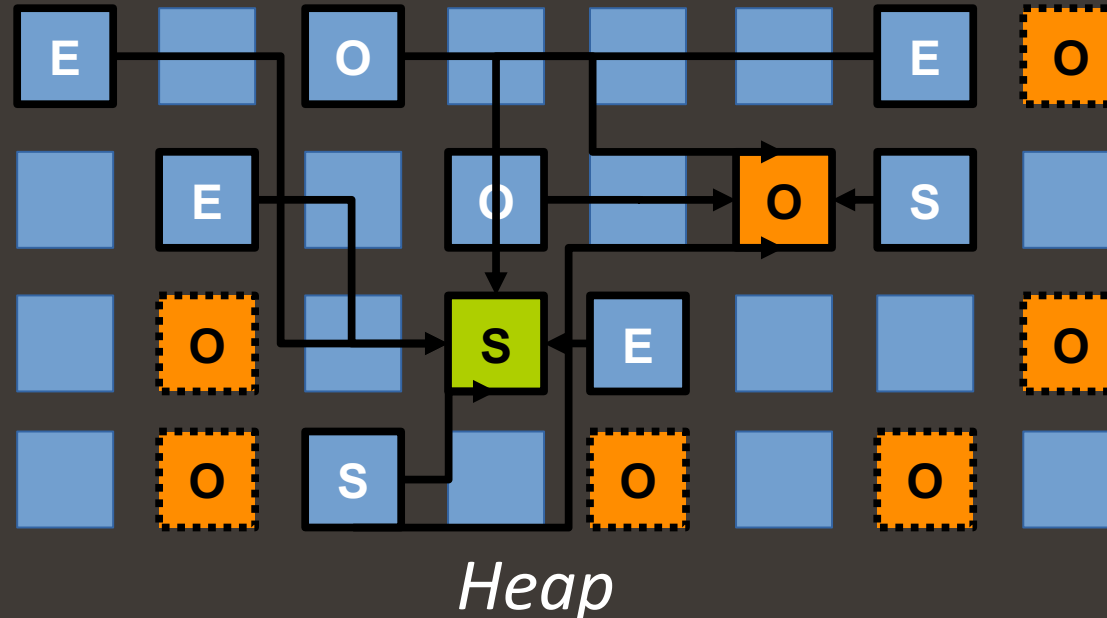
- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.



Heap

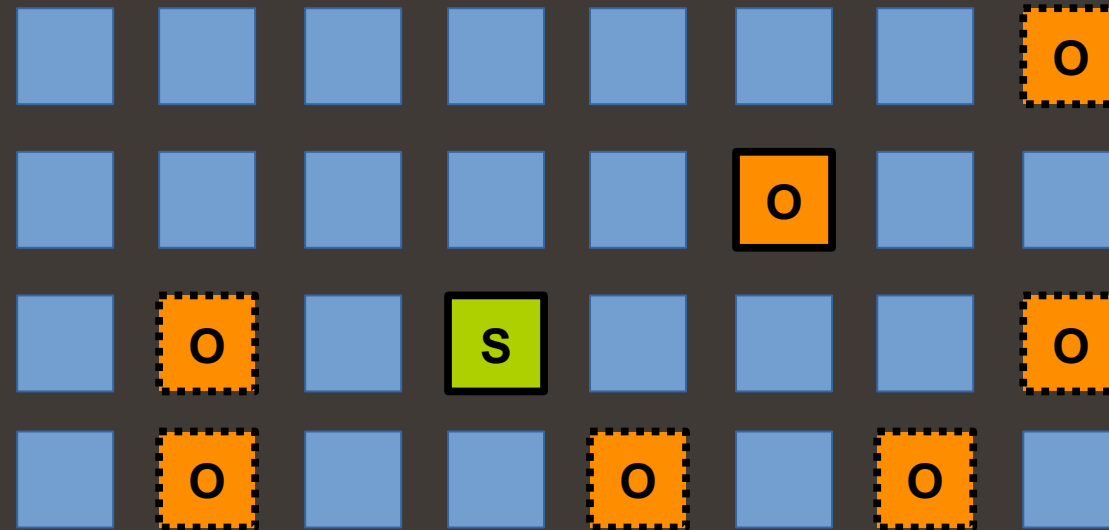
G1 - Generational region-based memory management

- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.



G1 - Generational region-based memory management

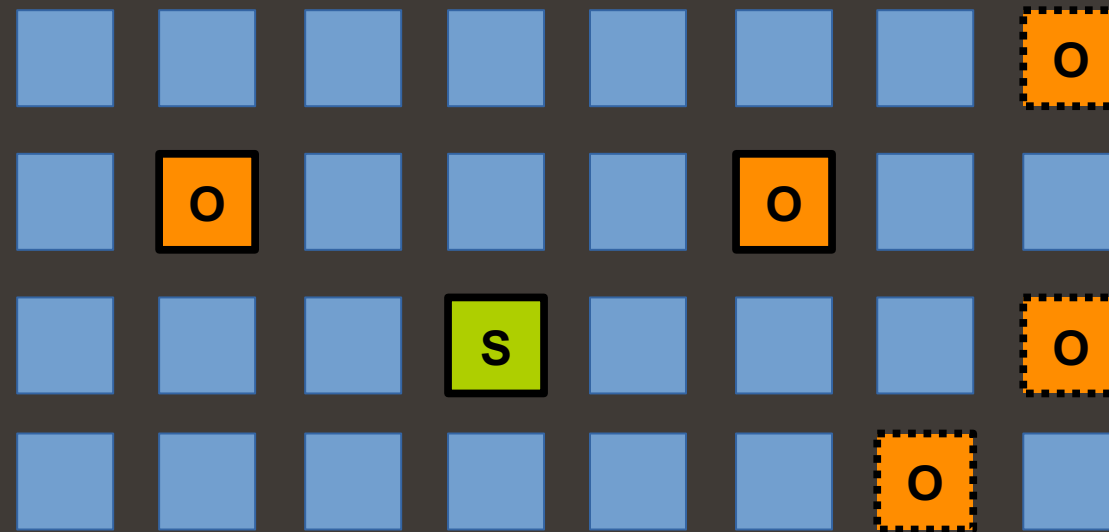
- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.



Heap

G1 - Generational region-based memory management

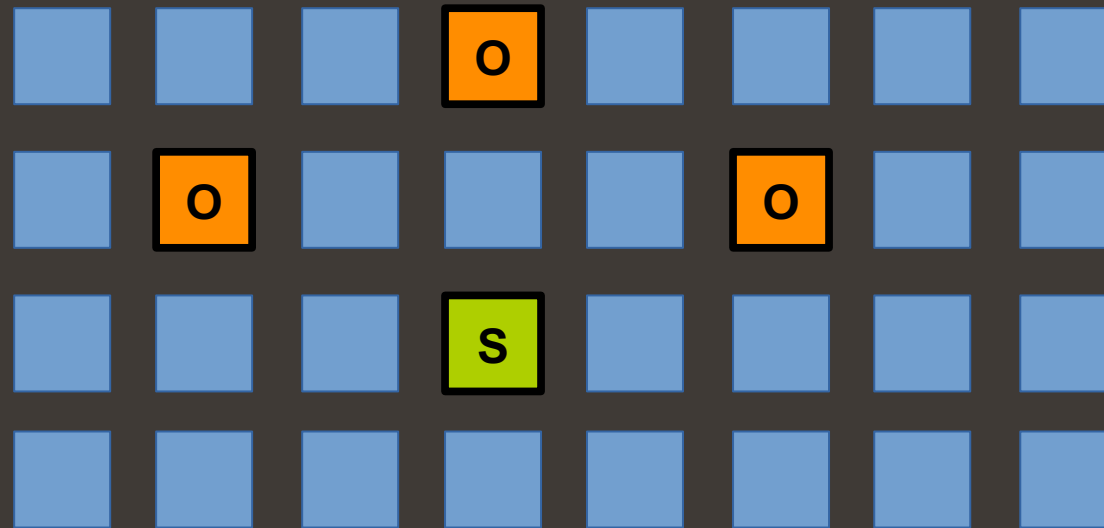
- Eden, survivor and old regions are then collected in *mixed collections*.
- Live objects are compactly copied into survivor and old regions.



Heap

G1 - Generational region-based memory management

- When no more old regions are suitable for collection, then G1 will resume doing young collections



G1 enhancements since JDK 8

13 GC-related JEPs since JDK 8

- 5 related to G1

G1 enhancements since JDK 8

13 GC-related JEPs since JDK 8

- 5 related to G1

~1450 GC enhancements

- ~699 related to G1

The JEPs represent only a small portion of the work going into the GC area.

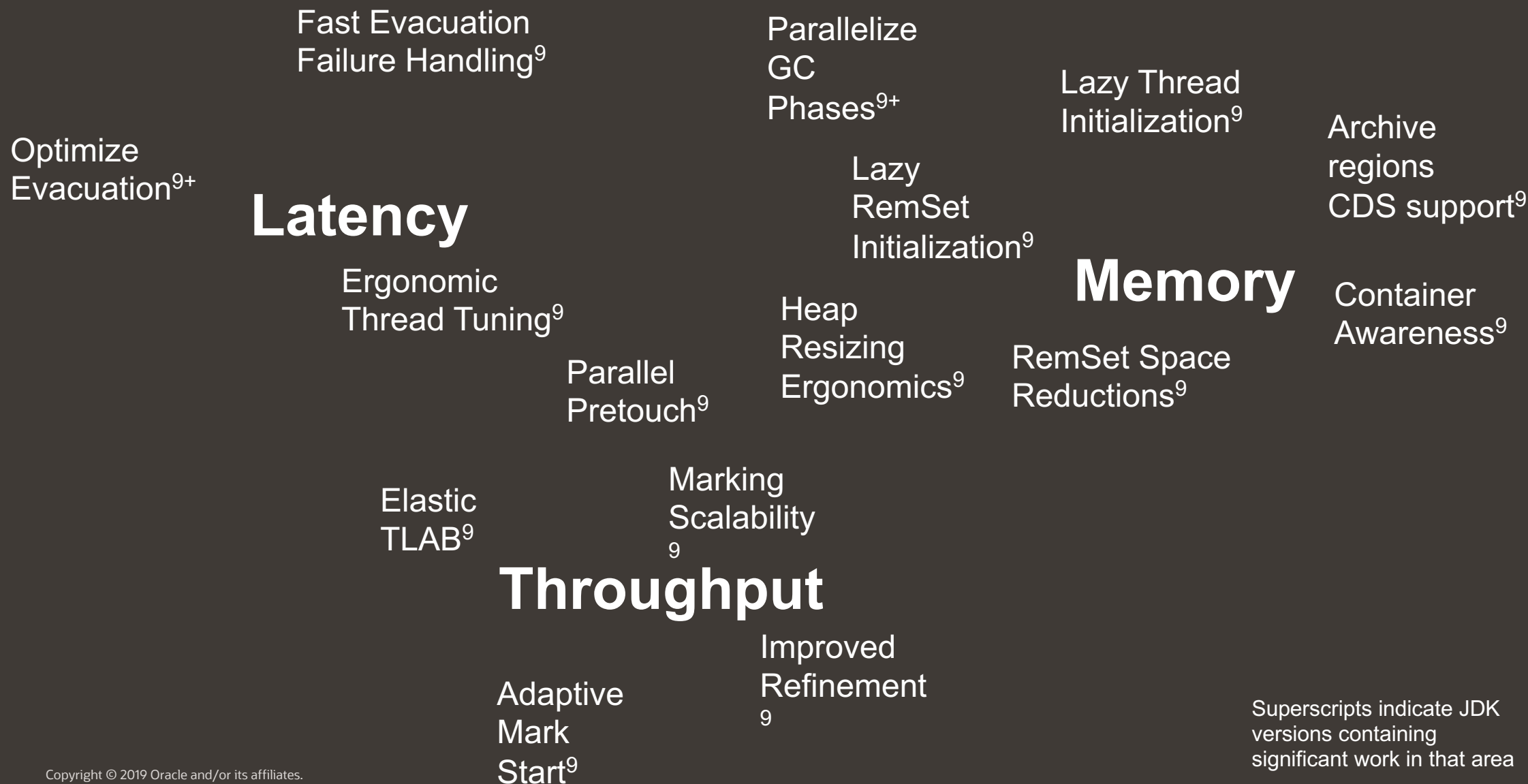
G1 enhancements between JDK 8 and JDK 9

Latency

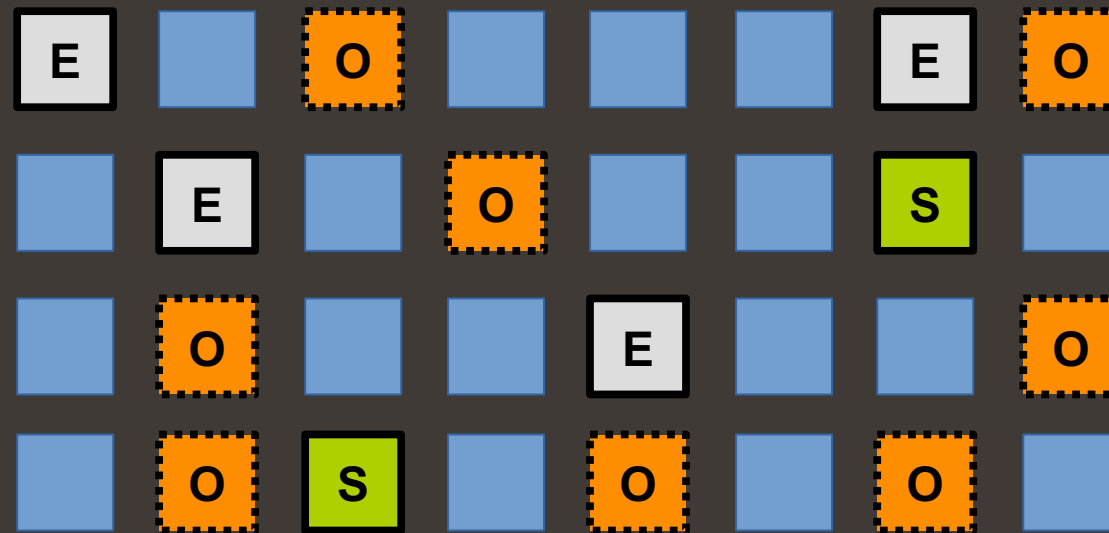
Memory

Throughput

G1 enhancements between JDK 8 and JDK 9



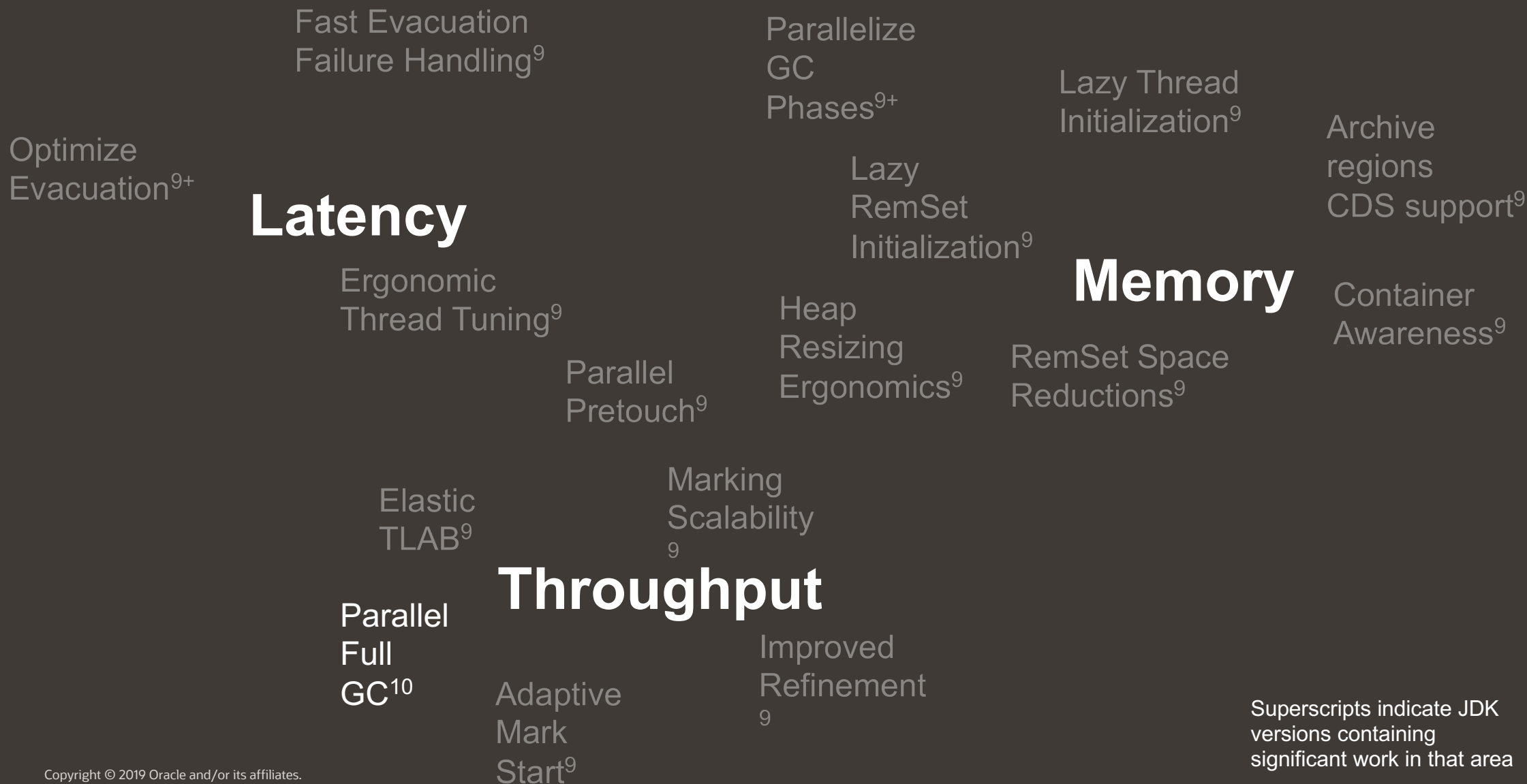
Adaptive Mark Start/Initiating Heap Occupancy Percentage (IHOP)



Heap

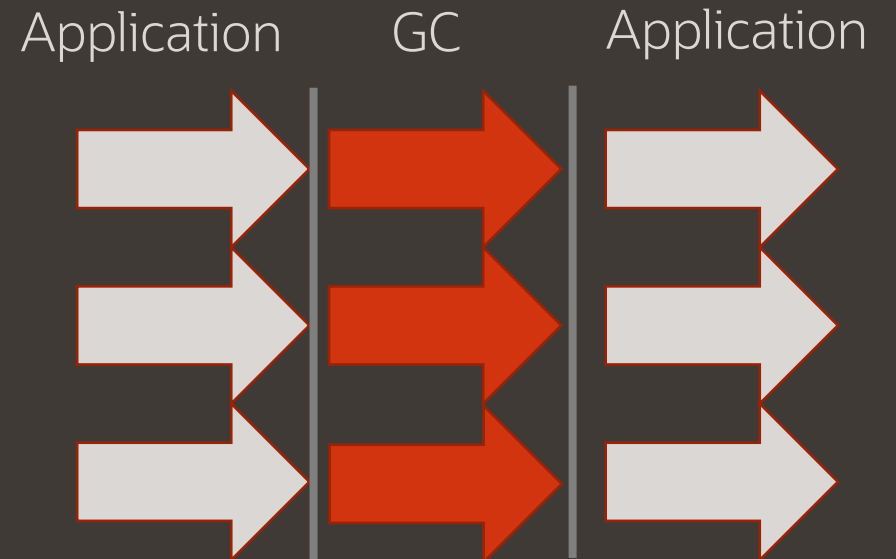
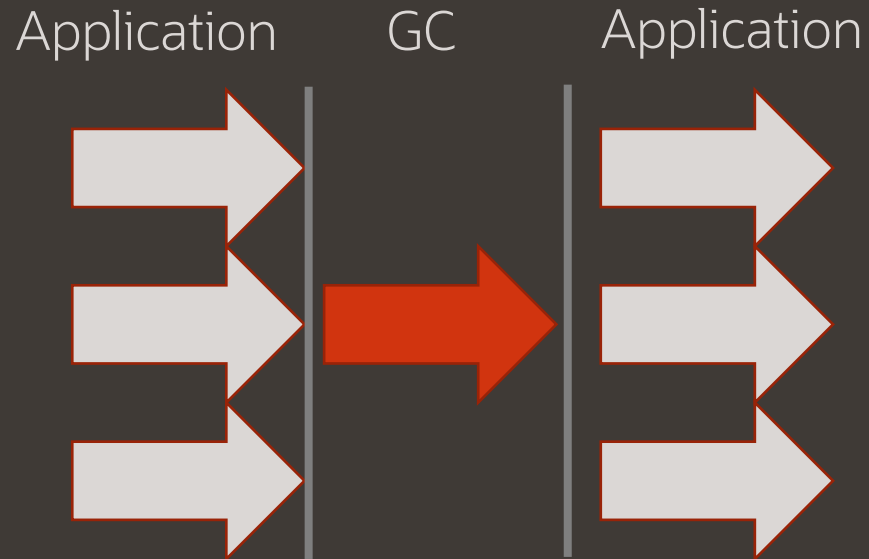


G1 enhancements between JDK 9 and JDK 10

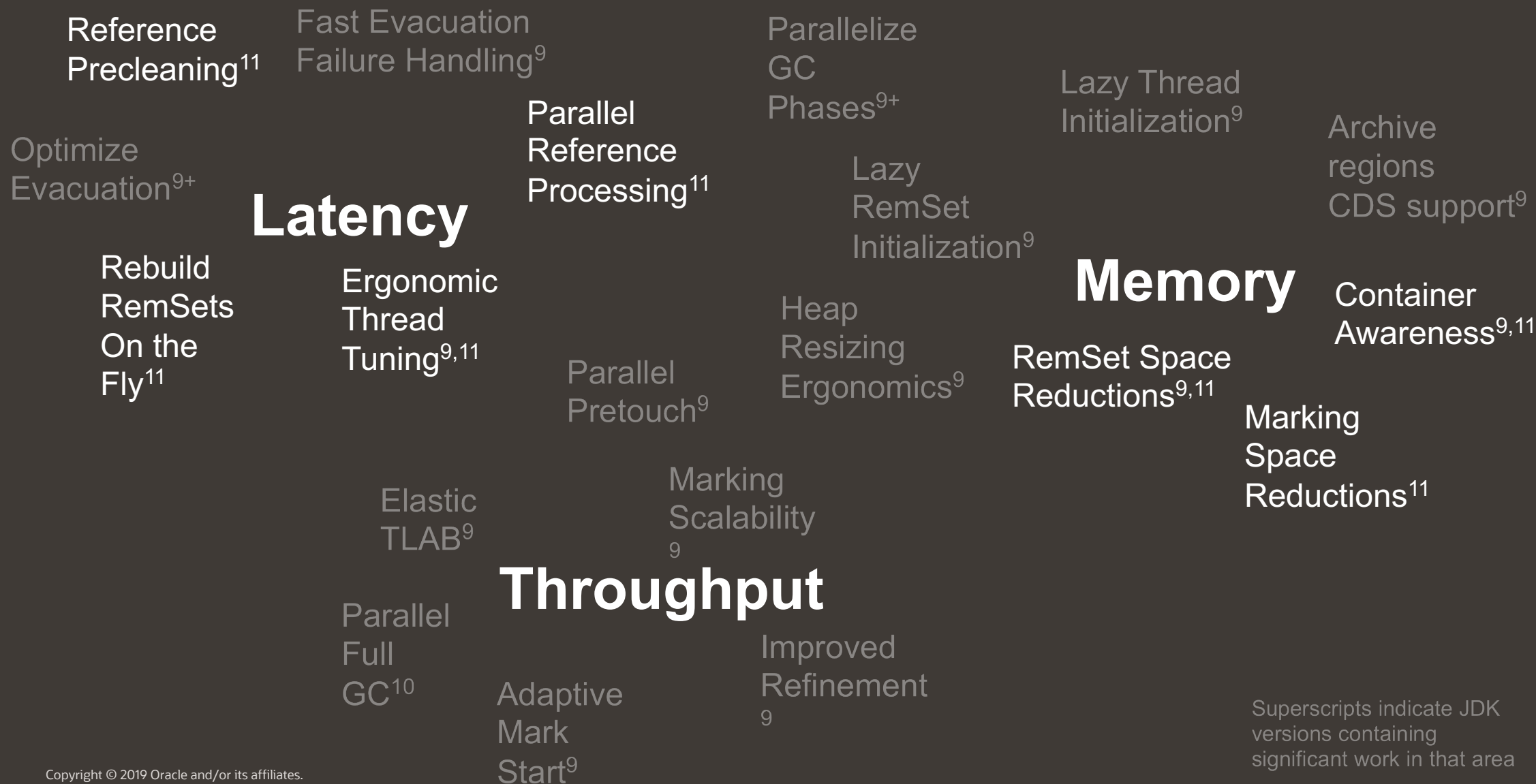


Parallel Full GC

- Now uses the same number of parallel threads as young and mixed GCs



G1 enhancements between JDK 10 and JDK 11



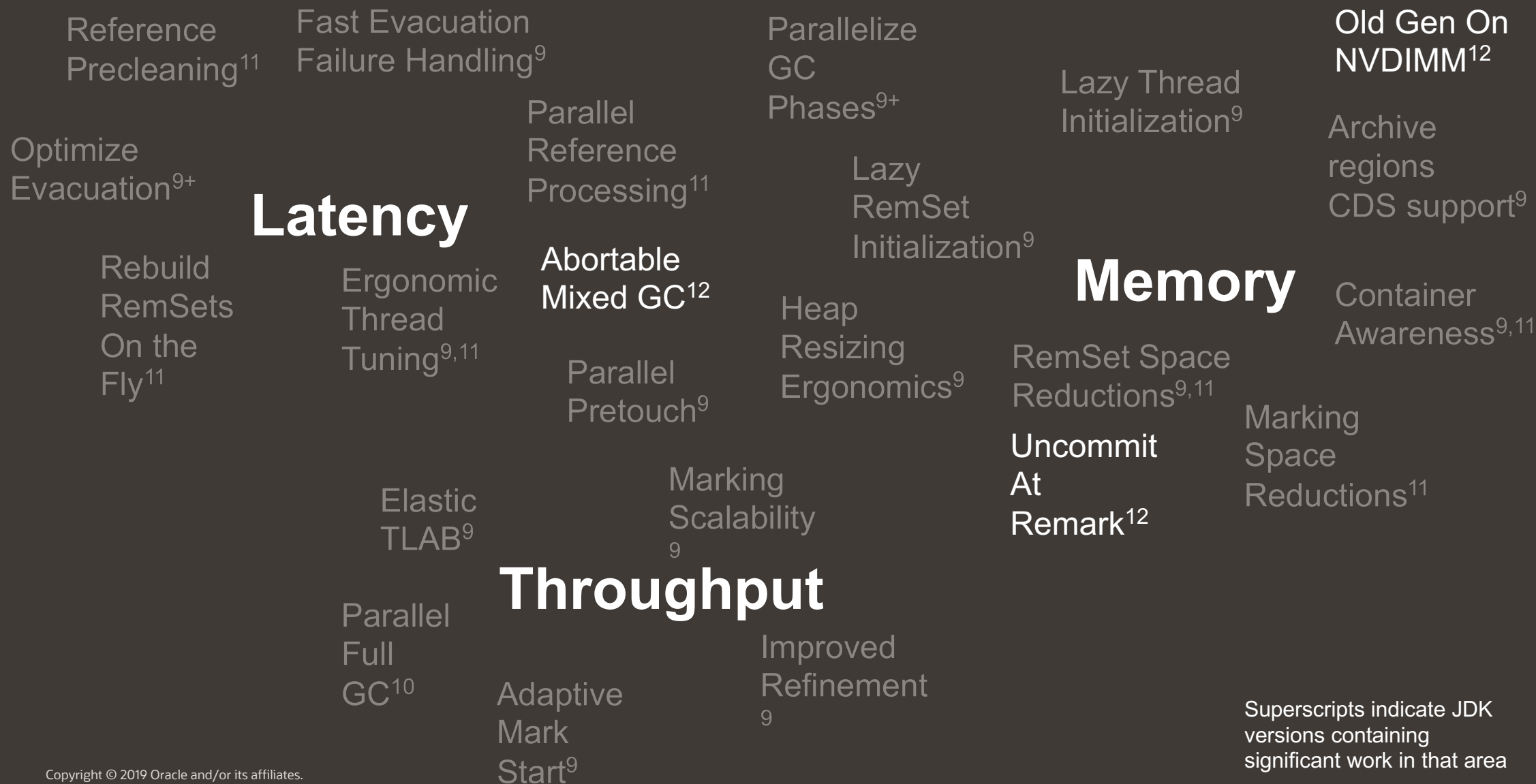
Rebuild Remembered Sets on the Fly

- Remembered sets are data structures that track references into a region
- Can occupy a significant amount of memory i.e. 20% of the total heap
 - Remembered sets for old regions are particularly large
- G1 maintains remembered sets for all regions
 - But only need remembered sets for old regions during Mixed GCs

Rebuild Remembered Sets on the Fly

- Dynamically rebuild remembered sets after the concurrent mark phase
- Only build remembered sets for regions in the collection set
- Improves both throughput and latency

G1 enhancements between JDK 11 and JDK 12



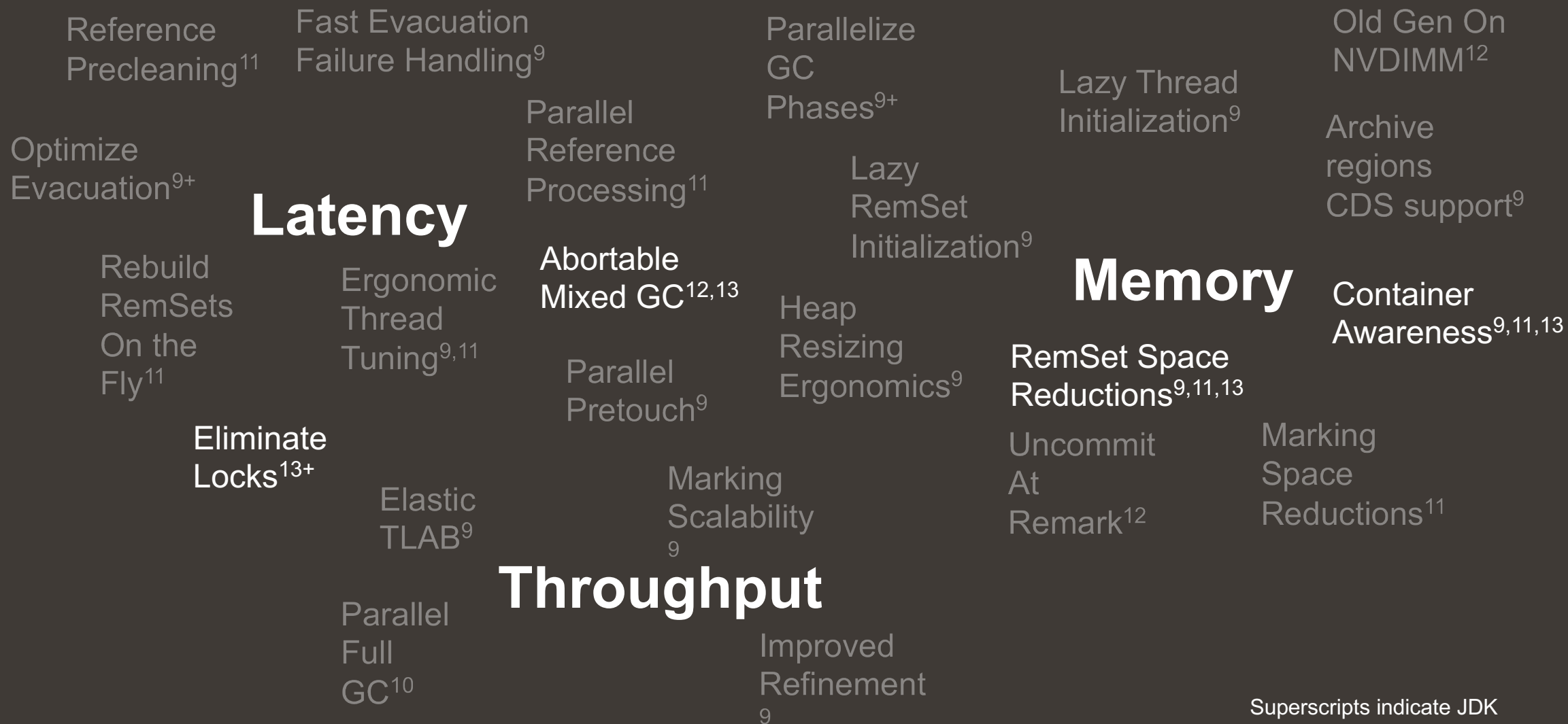
Abortable Mixed GCs

- G1 attempts to avoid exceeding the pause time target by using heuristics to select a collection of regions that can be collected within the given time
- Once started, all the selected regions must be collected
- Can exceed the pause target if the collection set is too large

Abortable Mixed GCs

- Split the collection set into mandatory and optional regions
- Mandatory regions are always collected
- Optional regions can be processed incrementally until there is no time left

G1 enhancements between JDK 12 and JDK 13



G1 performance improvements between JDK 8 and JDK 14

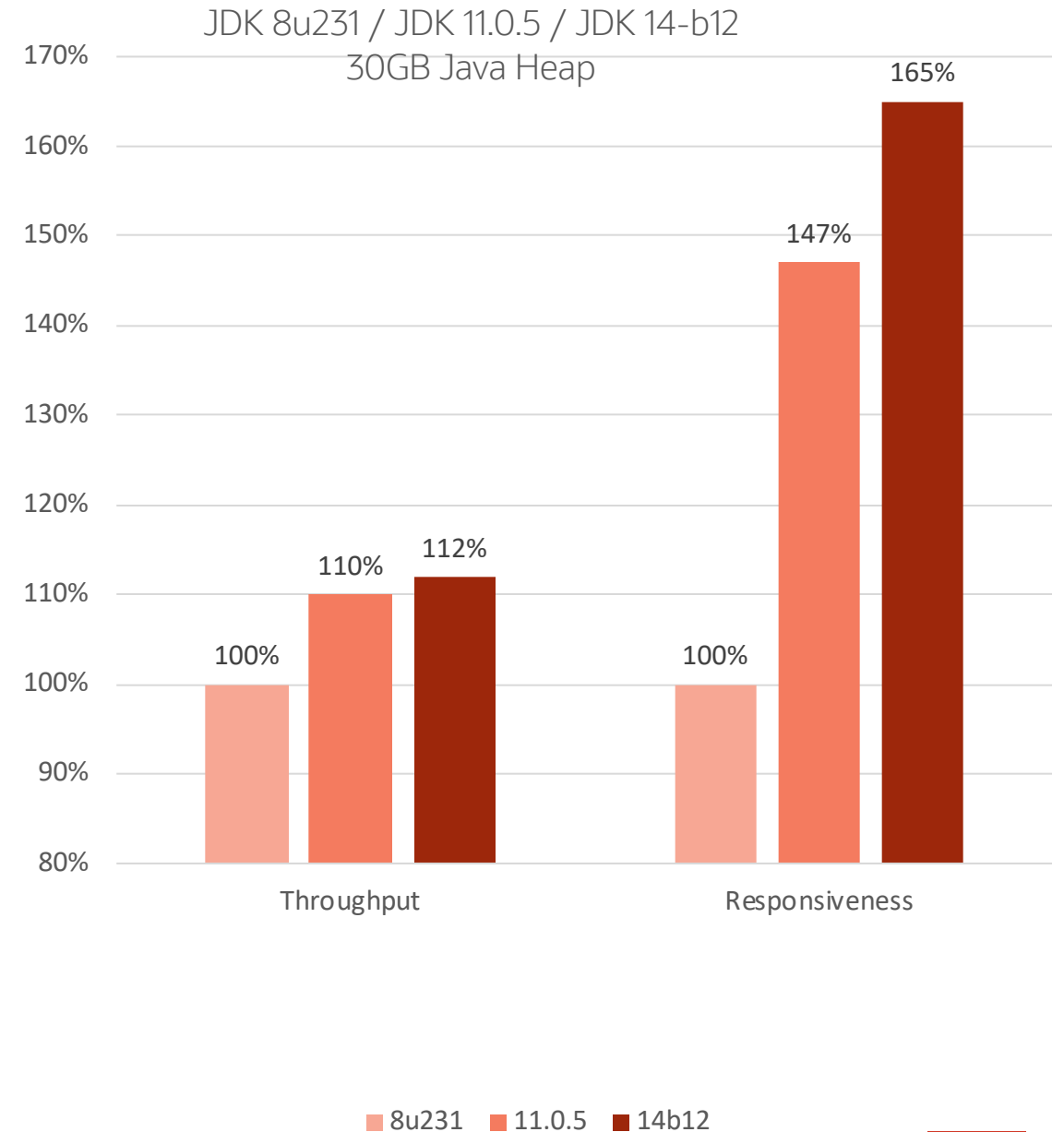
Heap Size: 30GB

OS: Oracle Linux 7.4

HW: Intel Xeon E5-2690 2.9GHz

2 sockets, 8 cores

- 12% improvement in maximum throughput between JDK 8 and JDK 14
- 65% improvement in responsiveness



G1 enhancements JDK 14 and beyond

Predictions

Improved
RemSet
Scan

RemSet
Storage

Latency

Memory

Reduce
Barrier
overhead

Throughput

Improved NUMA
support

ZGC

ZGC - A Scalable Low-Latency Garbage Collector

TB

Multi-terabyte heaps

10_{ms}

Max GC pause time



Easy to tune

15%

Max application throughput reduction

ZGC at a Glance

Concurrent
Tracing
Compacting
Single generation

Region-based
NUMA-aware
Load barriers
Colored pointers

ZGC pause times do not increase
with the heap or live-set size

ZGC pause times do increase
with the root-set size

(Number of Java Threads)

Auto-tuning

Design: No knobs until proven differently!

Application Threads (aka. "Mutator Threads")

Allocate memory (`new`)
Generate garbage

GC Threads

Collect garbage
Free up memory for allocation

Allocation Rate > Collection Rate → Allocation Stall

Avoiding Allocation Stalls

Generate less garbage

Collect garbage faster

Avoiding Allocation Stalls

Generate less garbage

Collect garbage faster

Avoid allocating objects

Run application more slowly

JIT compiler optimizations

Avoiding Allocation Stalls

Generate less garbage

Collect garbage faster

~~Avoid allocating objects~~

Run application more slowly

JIT compiler optimizations

Avoiding Allocation Stalls

Generate less garbage

Collect garbage faster

~~Avoid allocating objects~~

~~Run application more slowly~~

JIT compiler optimizations

Avoiding Allocation Stalls

Generate less garbage

Collect garbage faster

~~Avoid allocating objects~~

Speed up GC implementation

~~Run application more slowly~~

Use more GC threads

JIT compiler optimizations

Have application threads help out

Use more memory

“Barriers” – GC Callbacks

Barriers

Small piece of code injected by the JVM

Executed when accessing/updating an object

Store / Load Barriers (aka. Write / Read)

Store: Executed when storing a reference to the Java heap

Load: Executed when reading a reference from the Java heap

Use of Barriers in GCs

Existing GCs make use of store (write) barriers

- True for G1, Parallel, Serial, ~~CMS~~

- Helps speed up generational support

ZGC uses load barriers

- Mutator threads take on some additional work

ZGC Load barrier

```
String name = person.name;
```

<load barrier>

```
String copy = name;           // No barrier
```

```
name.isEmpty();              // No barrier
```

```
int age = person.age;        // No barrier
```

Person

```
String name;  
int    age;  
double height;
```

String

...

ZGC Load barrier

```
String name = person.name;  
<load barrier>
```

ZGC Load barrier

```
String name = person.name;  
<load barrier>
```

ZGC Load barrier

```
String name = person.name;  
if (!good(name)) {  
    name = slow_case(name);  
}
```

ZGC Load barrier

```
String name = person.name;  
if (!good(name)) {  
    name = slow_case(name);  
}
```

Where to place the good/bad information?
In object? GC side structure?

Colored Pointers

Modern machines use 64-bit addresses/pointers

Exception: Compressed oops

Object pointers stored as 32-bit “indexes”

Colored Pointers

Modern machines use 64-bit addresses/pointers

Exception: Compressed oops

Object pointers stored as 32-bit “indexes”

For ZGC

References (object pointers) are always 64 bit



Colored Pointers

Modern machines use 64-bit addresses/pointers

Exception: Compressed oops

Object pointers stored as 32-bit “indexes”

For ZGC

References (object pointers) are always 64 bit

Only 64-bit platforms

No compressed oops



Load Barrier Using Colored Pointers

```
mov 0x10(%rax), %rbx      // String n = person.name;
test %rbx, 0x20(%r15)    // Bad color?
jnz  slow_path          // Yes -> Enter slow path and
                           // mark/relocate/remap, adjust
                           // 0x10(%rax) and %rbx
```

Load Barrier Using Colored Pointers

```
mov 0x10(%rax), %rbx      // String n = person.name;  
test %rbx, 0x20(%r15)    // Bad color?  
jnz  slow_path          // Yes -> Enter slow path and  
                           // mark/relocate/remap, adjust  
                           // 0x10(%rax) and %rbx
```

~4% execution overhead

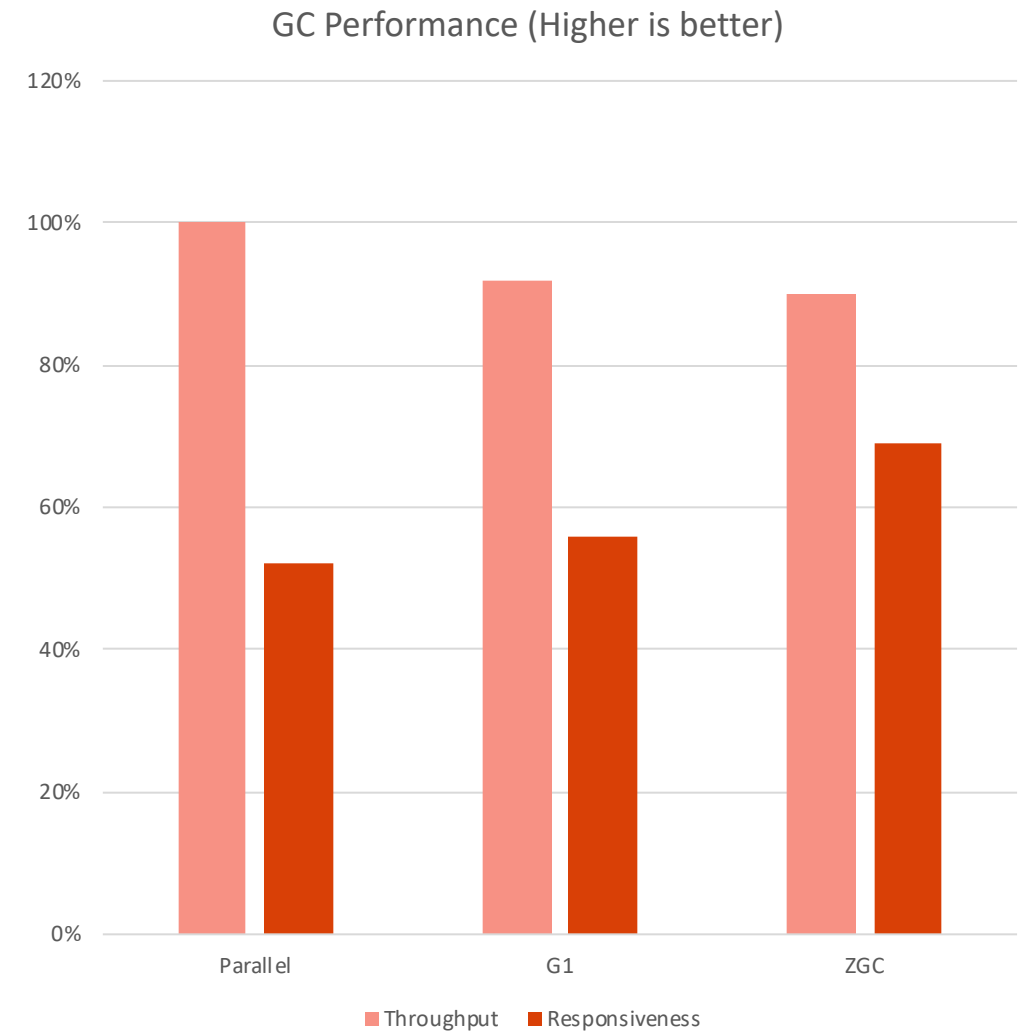
GC Performance

Heap Size: 128GB

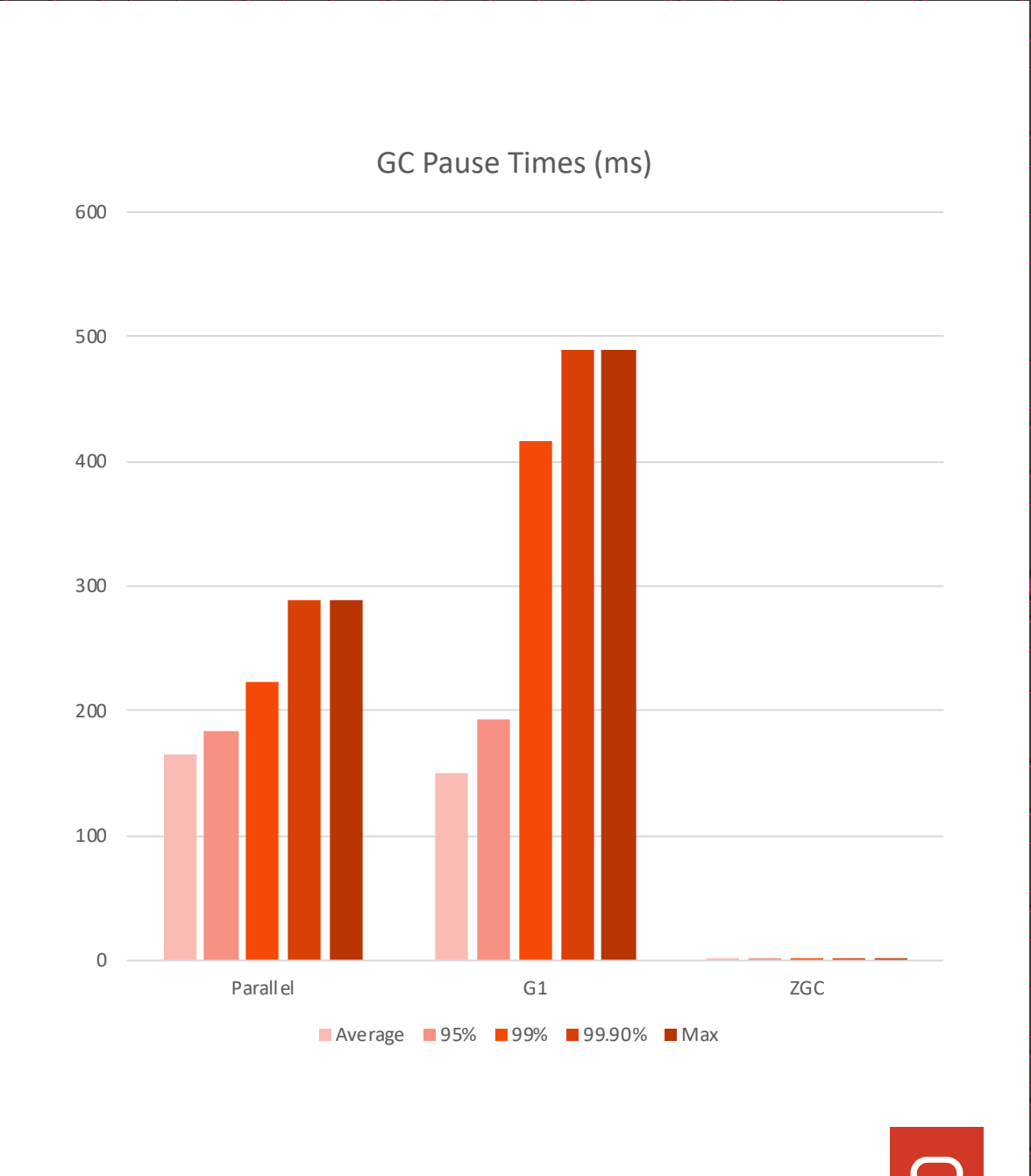
OS: Oracle Linux 7.5

HW: Intel Xeon E5-2690 2.9GHz

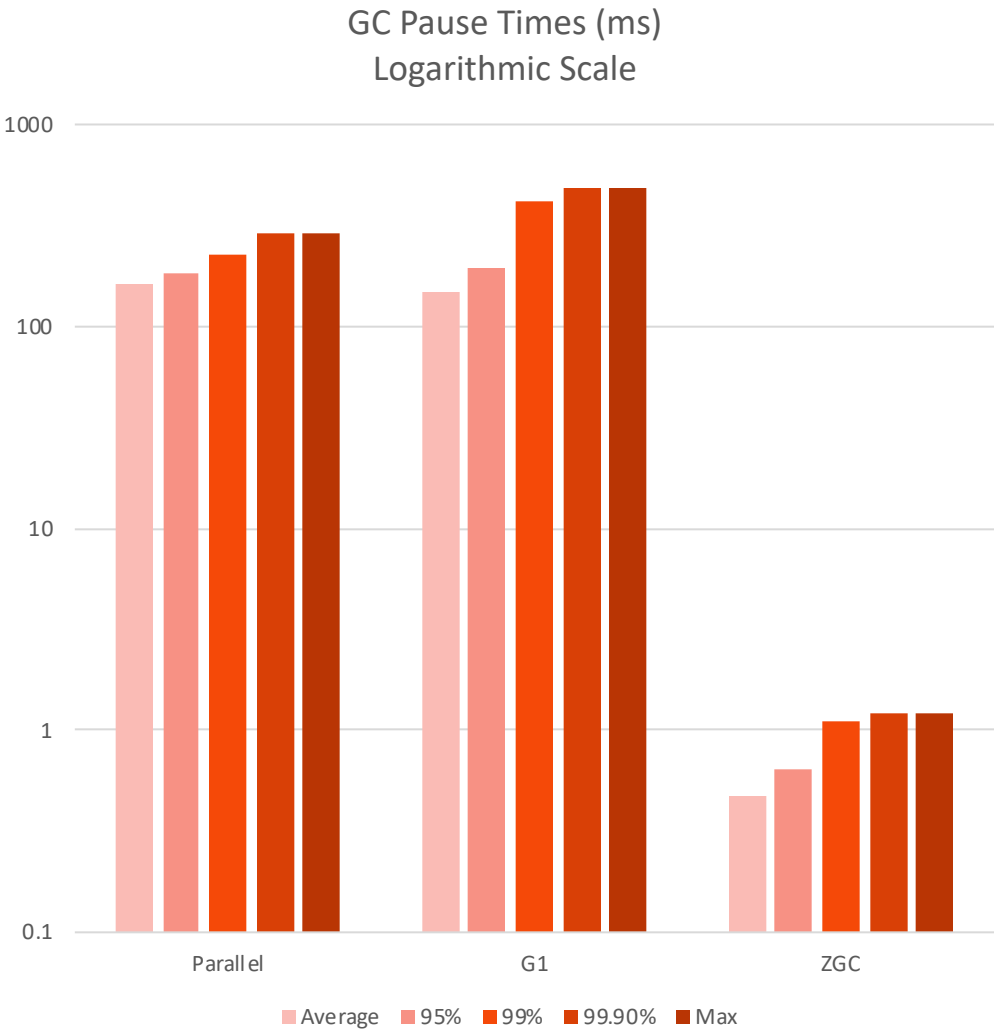
2 sockets, 16 cores (32 hw-threads)



GC Pause Times



GC Pause Times (Logarithmic Scale)



Using ZGC (JDK 11+)

`-XX:+UnlockExperimentalVMOptions`

`-XX:+UseZGC`

Tuning options

Look out for Allocation Stalls

Option #1: Increase the max heap size

-Xmx<size>

Trade memory for better latency

Option #2: Increase number of GC threads

-XX:ConcGCThreads=<number>

Trade CPU-time for better latency

Status/Recent Improvements

JDK 11

First JDK to include open sourced ZGC (Experimental)

Status/Recent Improvements

JDK 11

First JDK to include open sourced ZGC (Experimental)

JDK 12

Concurrent class unloading

Thread-local handshakes

Status/Recent Improvements

JDK 11

- First JDK to include open sourced ZGC (Experimental)

JDK 12

- Concurrent class unloading

- Thread-local handshakes

JDK 13 – Released **hours** ago!

- Max heap size increased to 16 TB (was: 4 TB)

- Uncommit unused memory

- Linux/aarch64 port

Next up: Productization

Stability

Super late barrier expansion

Support additional platforms

macOS, Windows, ...

Potential Future Work

Generational support

- Leverage “Weak generational hypothesis”

- Manage higher allocation rates

- Reduce CPU utilization

Segmented Array Clearing

Chasing Sub-millisecond *max* pause times

- Concurrent thread stack scanning

- Additional latency improvements

- Low latency VM

Further Reading

G1 links

- GC Tuning Guide
 - <https://docs.oracle.com/en/java/javase/12/gctuning/garbage-first-garbage-collector.html>

ZGC links

- ZGC wiki
 - <https://wiki.openjdk.java.net/display/zgc/Main>
- GC Tuning Guide
 - <https://docs.oracle.com/en/java/javase/12/gctuning/z-garbage-collector1.html>

Thank You