Microsoft

Java Team

JDK 8 LTS to the latest Performance and Responsiveness Prospective

@RajputAnilK

Anil Kumar

Datacenter Performance

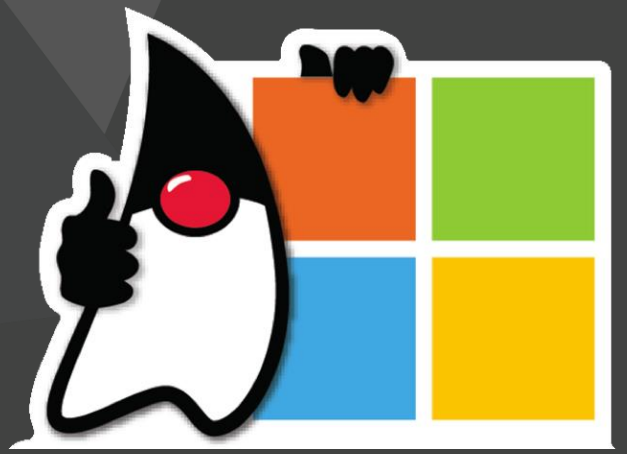
Anil.Kumar@intel.com

@mon_beck

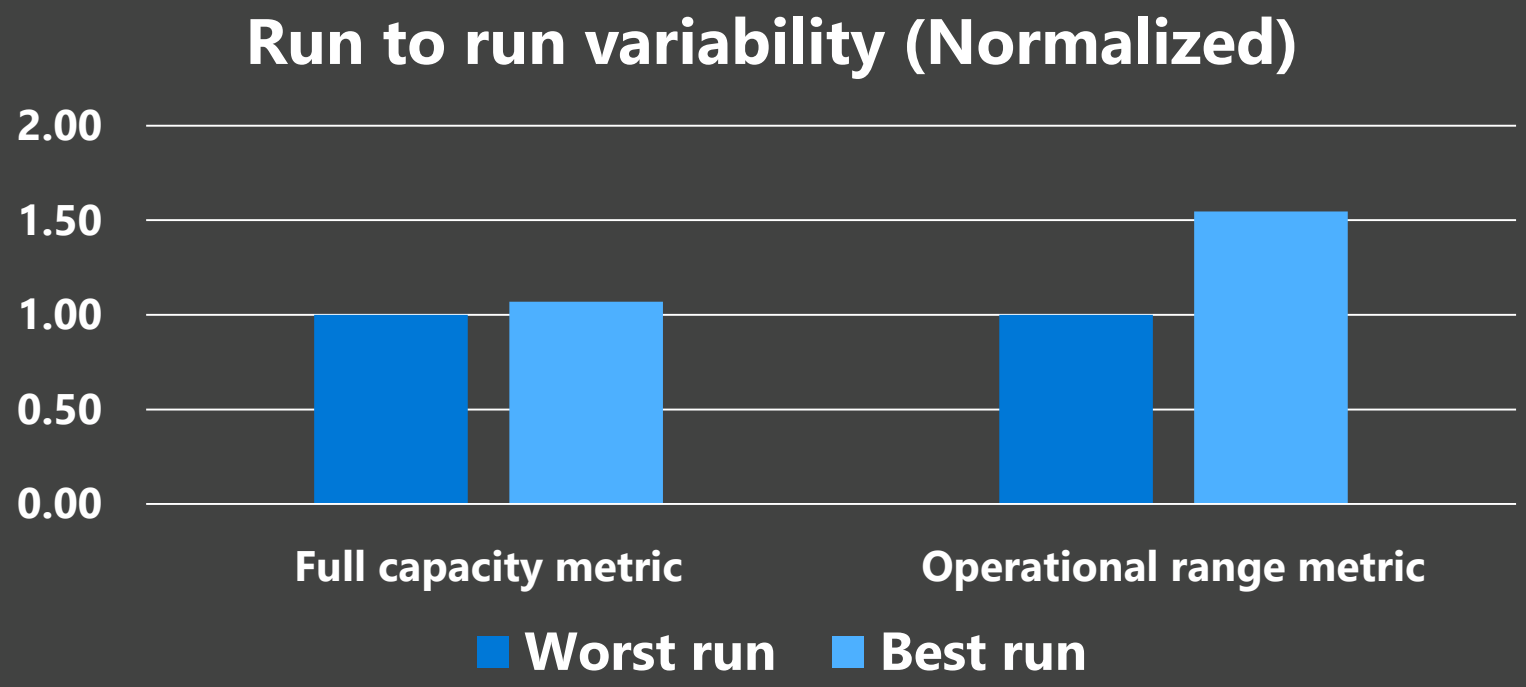
Monica Beckwith

JVM Performance

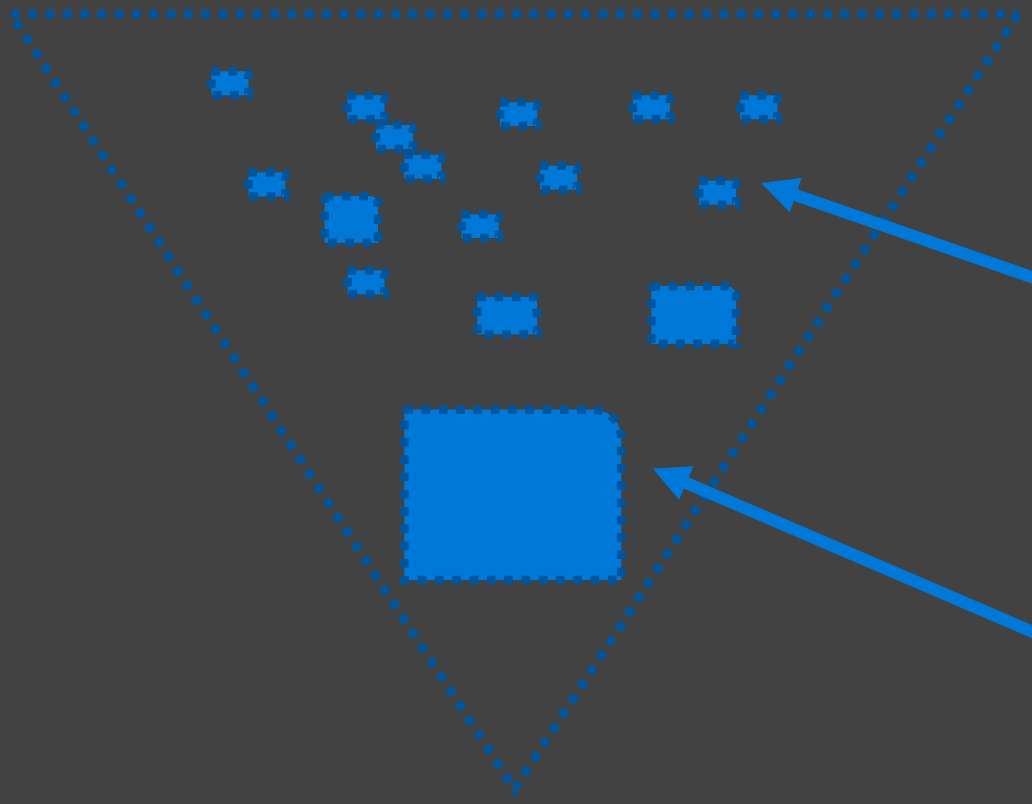
java-performance@Microsoft



Pre-production deployment



What test is running?



Thousands of apps from

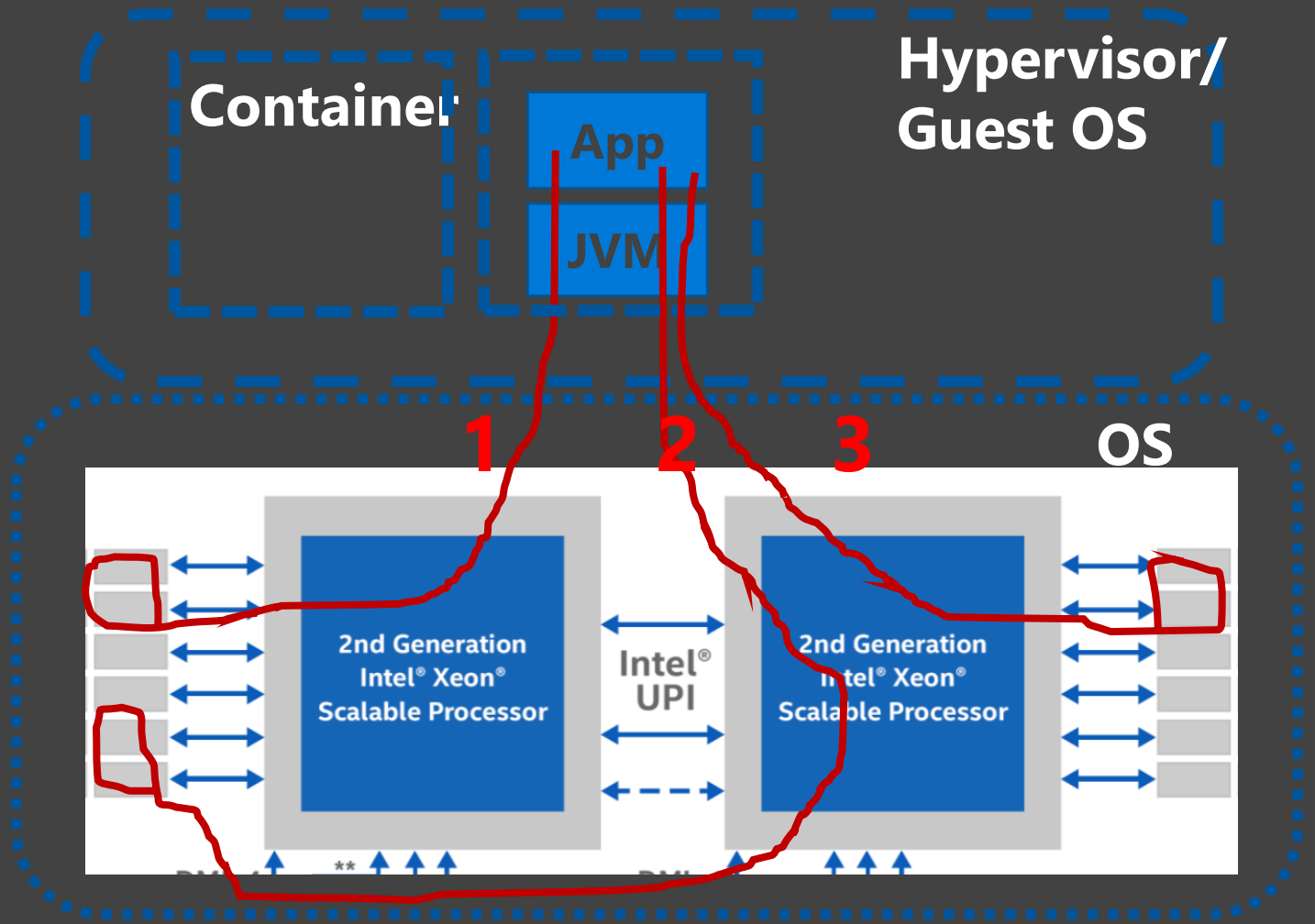
small data footprint (Heap 2GB)

to

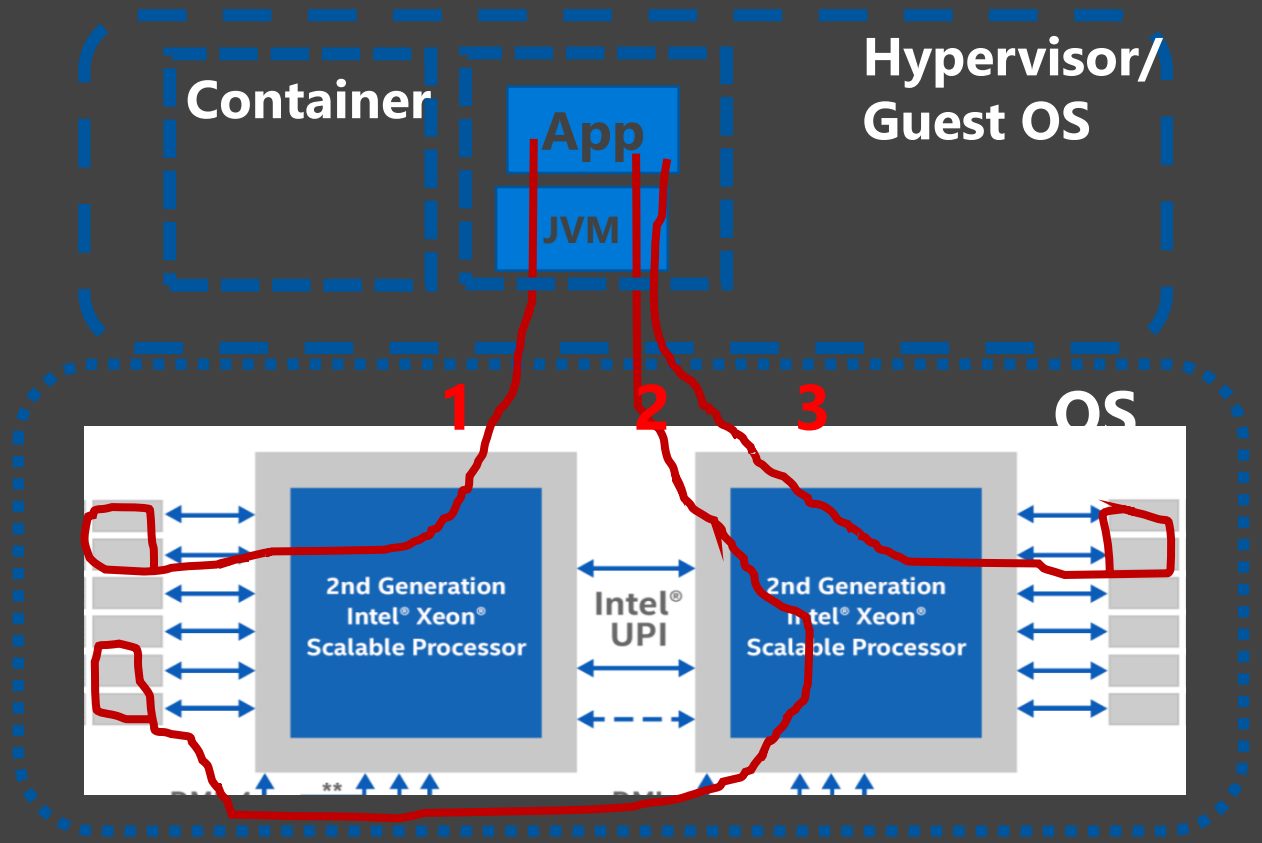
very large data footprint (Heap 100G

Test: a representative *benchmark* !

Deployment environment?



Deployment environment?



CPU threads?

Container,
Guest OS policies,
Pinning

Heap memory?

Guest OS policies,
Memory fragmentation,
Transparent large pages

Agenda

JDK 8 LTS to latest high level changes

Data using various benchmarks

Explanation for expected and/or strange behaviors

Summary

JDK 8 LTS to the latest …

Monitoring, code readability and debugging

New usages

Containers

Microservices

FaaS (Function as a Service)

Polyglot programming

Performance

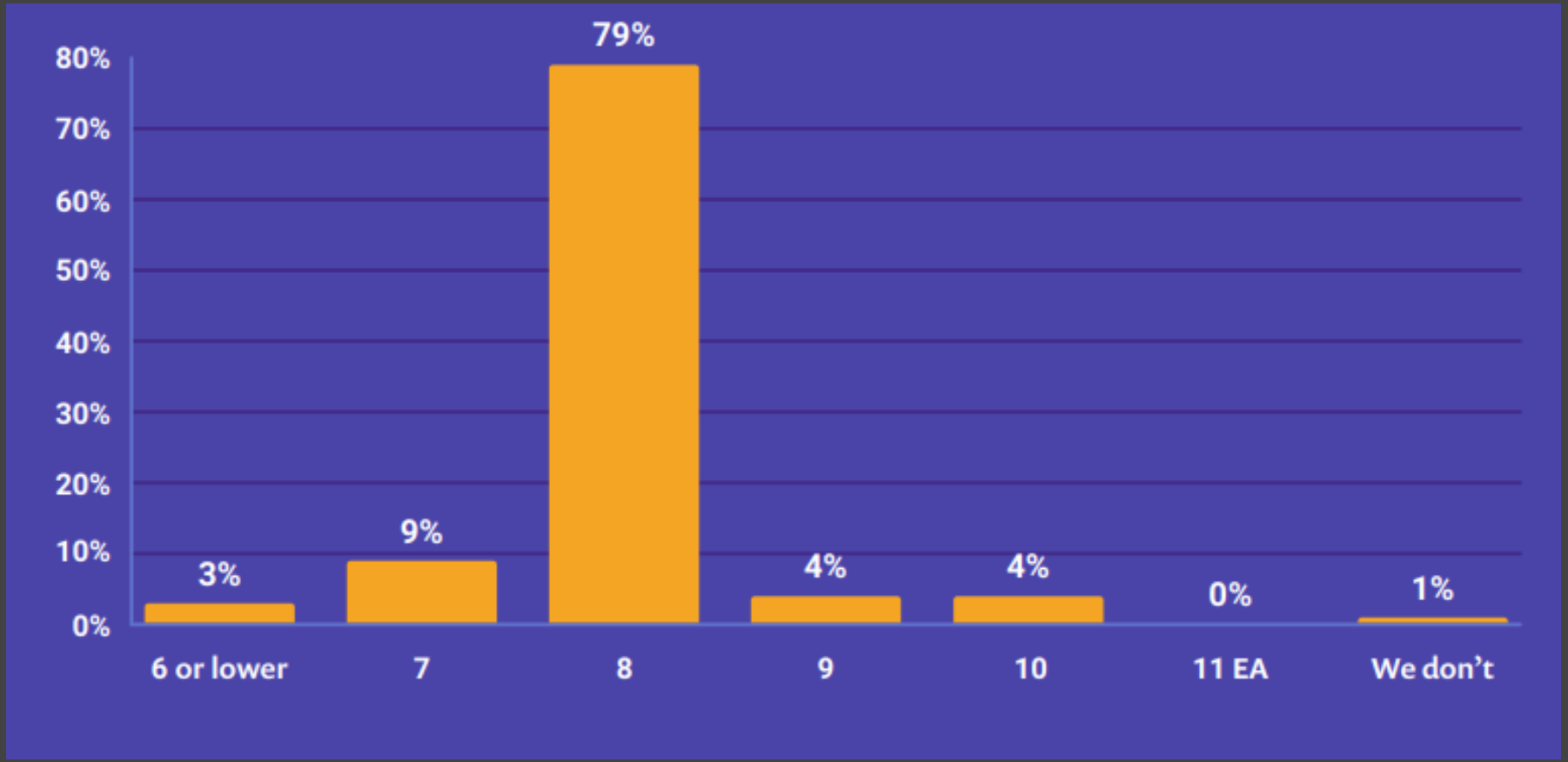
Concurrency → Fork/Join → Parallel Streams → Project Loom

Value Types

Networking: Java I/O → NIO → Netty

Java SE JDK 8 as base for normalization

Currently most use Java SE ?



Avoid workloads with high variability

Java* JMH* Benchmark

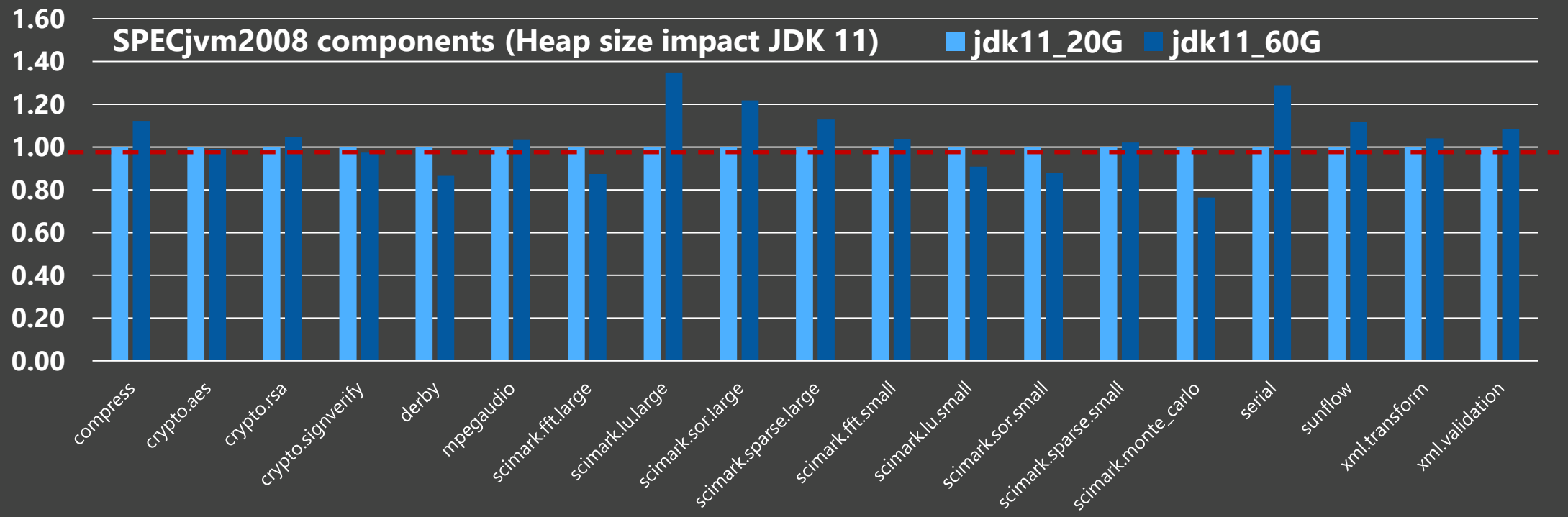
Configuration setting "-f 1 -wi 3 -w 5s -i 2 -r 15s -t 1"

org.openjdk.bench.java.util.stream.AllMatcher.seq_filter_findFirst	1.0x	Run 1
org.openjdk.bench.java.util.stream.AllMatcher.seq_filter_findFirst	1.5x	Run 2
org.openjdk.bench.vm.lambda.invoke.Function1.mref_bndLL_IL	1.0x	Run 1
org.openjdk.bench.vm.lambda.invoke.Function1.mref_bndLL_IL	1.5x	Run 2

Avoid heap allocation variability

Significant impact can result from variable heap allocation

For larger than 20GB heaps
System in use for long time
Transparent large pages in use



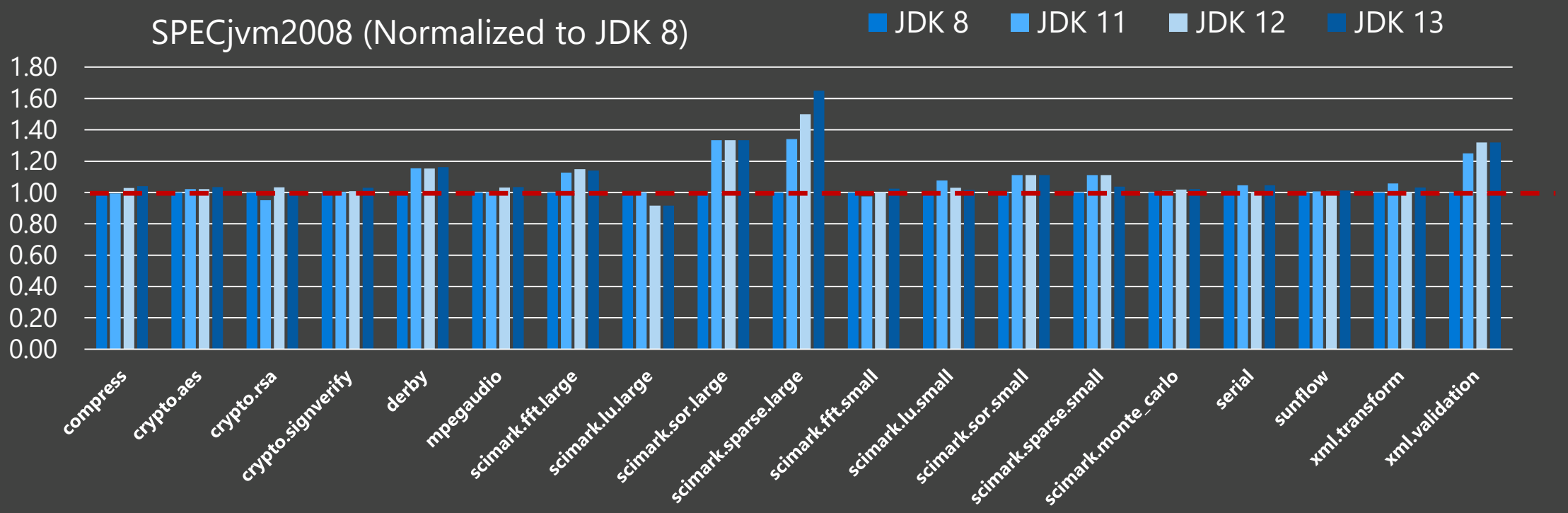
JDK 8 LTS vs. 11 LTS vs. 12 vs. 13 compare

Performance: Throughput

SPECjvm2008: compute + memory

JDK 11 LTS, 12 and 13 > JDK 8 LTS

6 minutes execution time for each worklet (sufficient optimization time)



Command line option for SPECjvm2008

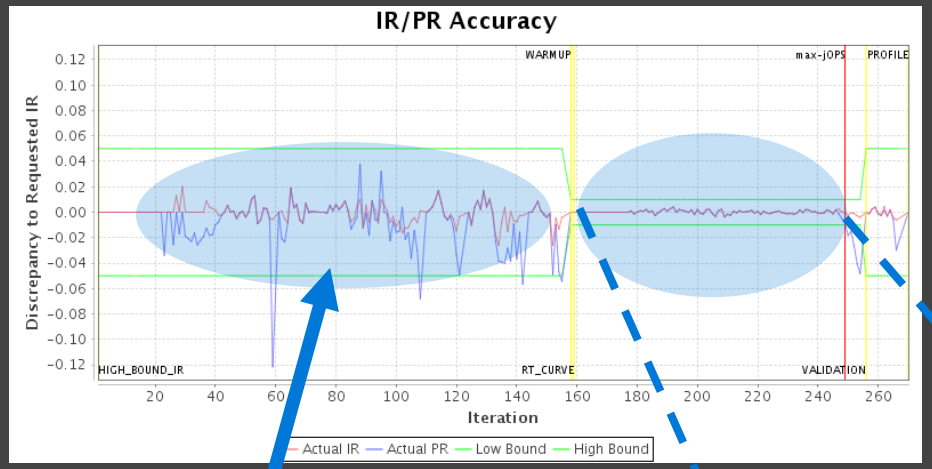
```
RUN_OPTS="-showversion"

${JAVA} ${RUN_OPTS} -jar SPECjvm2008.jar -ict -coe \
  startup.helloworld startup.compress startup.crypto.aes startup.crypto.rsa startup.crypto.signverify \
  startup.mpegaudio startup.scimark.fft startup.scimark.lu startup.scimark.monte_carlo startup.scimark.sor \
  startup.scimark.sparse startup.serial startup.sunflow startup.xml.transform startup.xml.validation \
 compress crypto.aes crypto.rsa crypto.signverify derby mpegaudio scimark.fft.large scimark.lu.large \
  scimark.sor.large scimark.sparse.large scimark.fft.small scimark.lu.small scimark.sor.small \
  scimark.sparse.small scimark.monte_carlo serial sunflow xml.transform xml.validation
```

JDK 8 LTS vs. 11 LTS vs. 12 vs. 13 compare

Performance, Responsiveness, and Variability

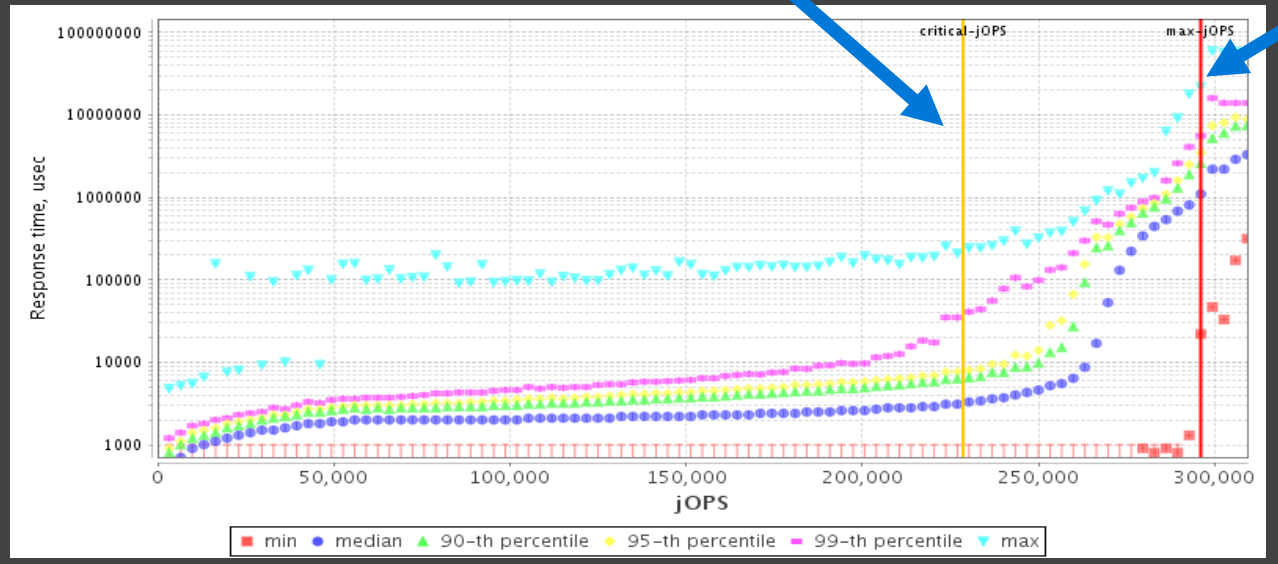
SPECjbb*2015: quick summary



Rough estimation of high bound settled performance

critical-jOPS (Responsiveness)

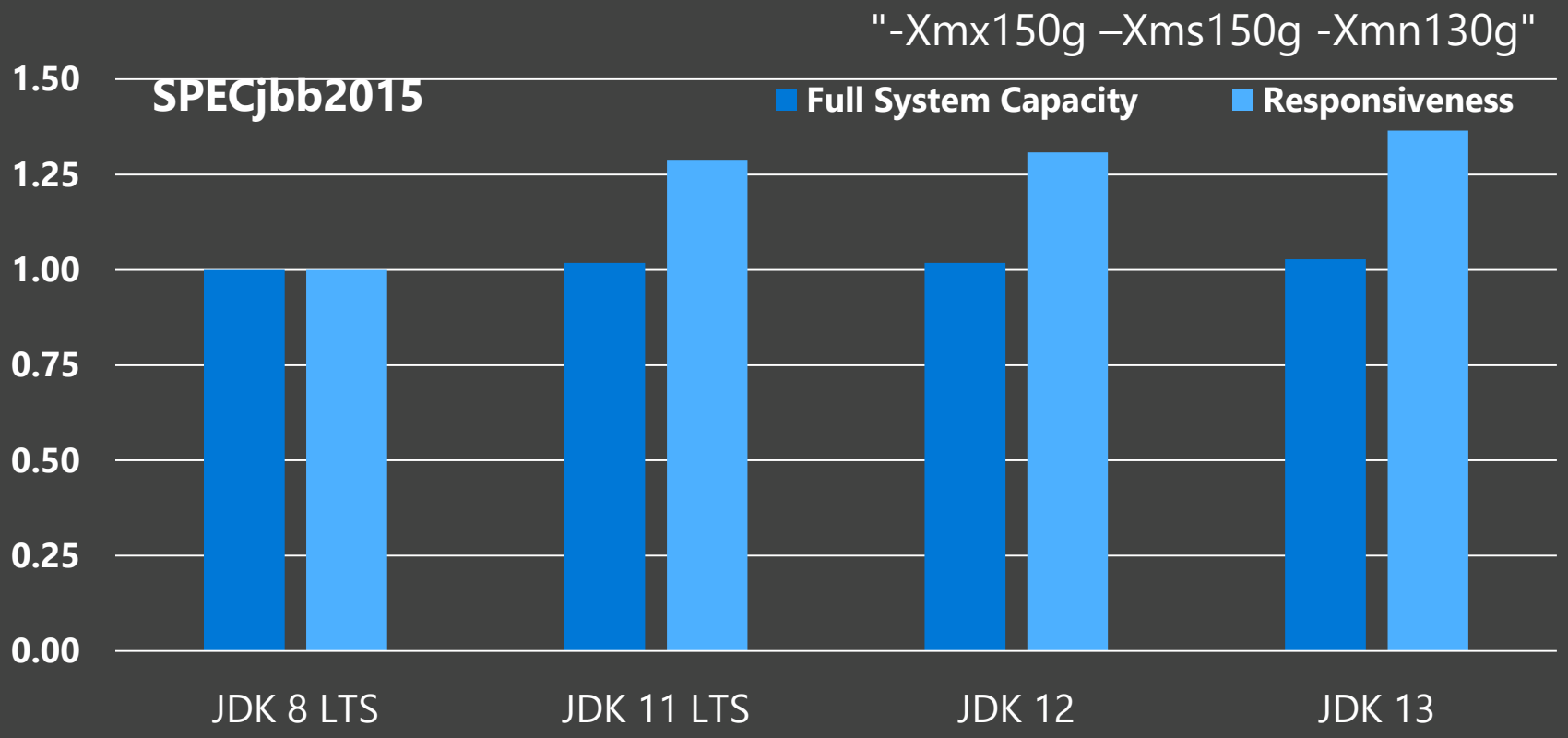
max-jOPS (Full system capacity)



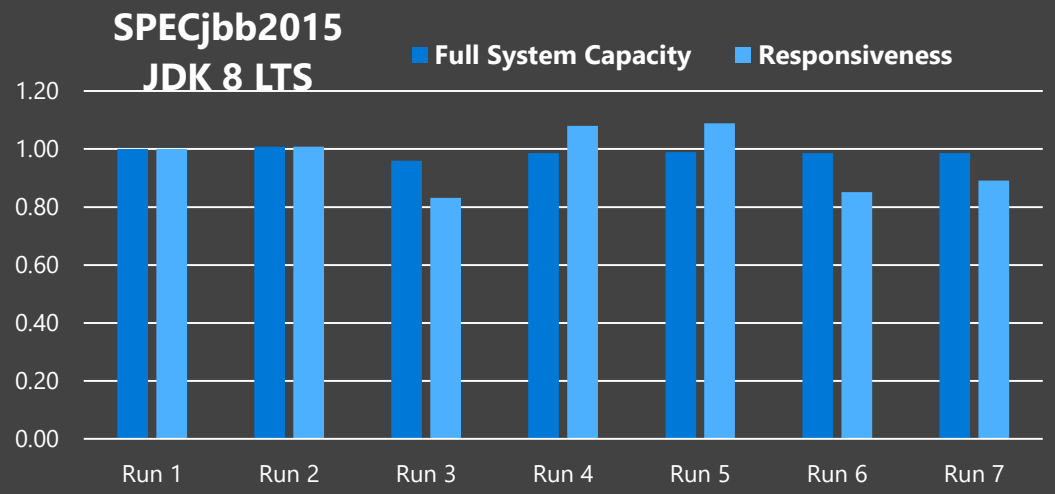
SPECjbb2015: JDK 8 LTS → JDK 11 LTS

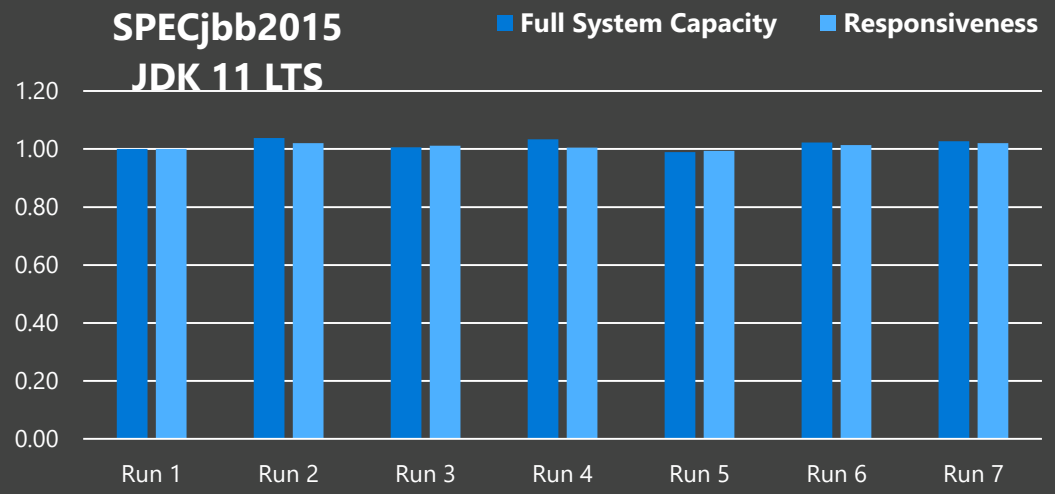
Full system capacity improved up to ~5%

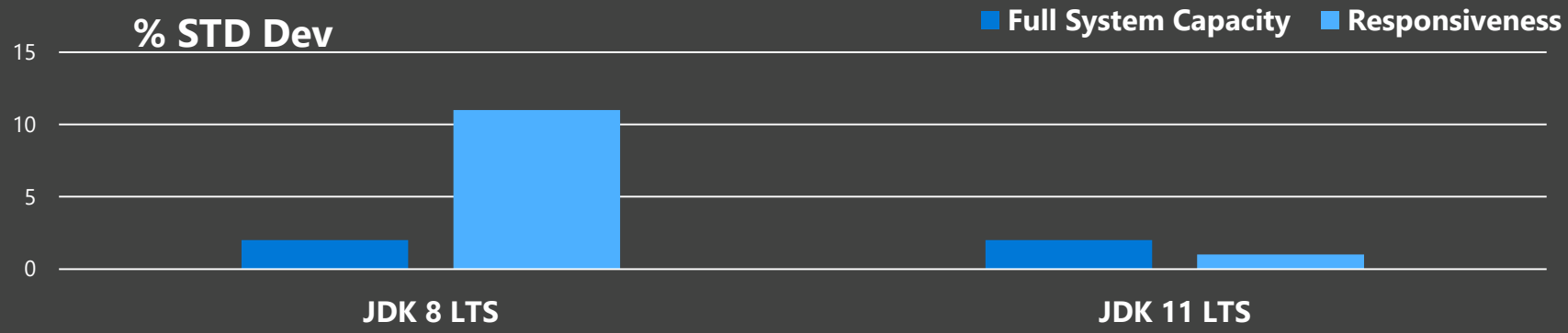
Responsiveness improved up to ~35%



Variability: JDK 8 LTS → JDK 11 LTS

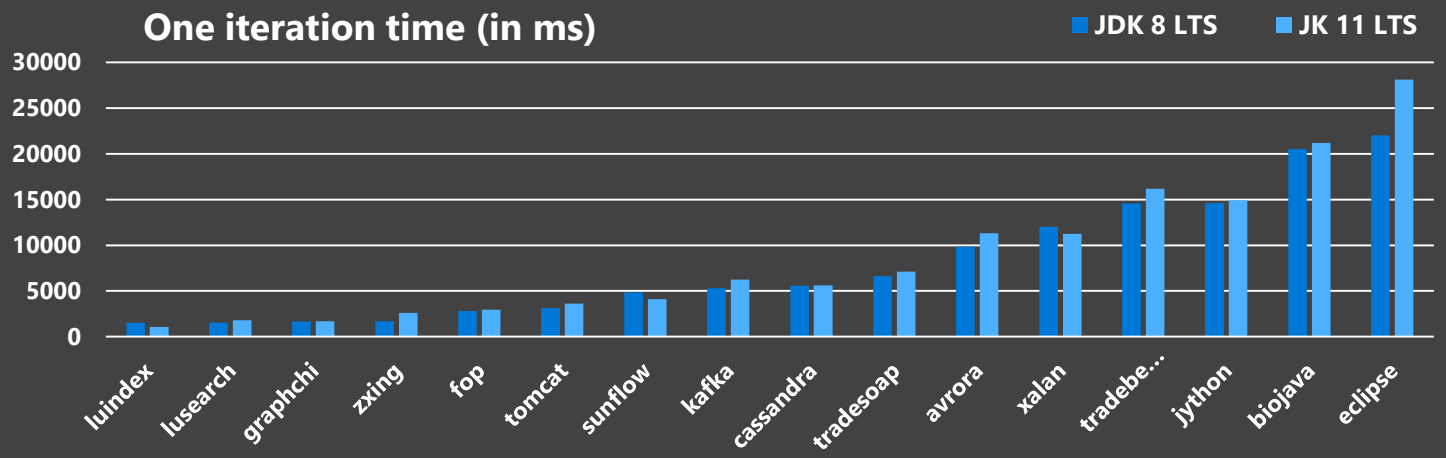


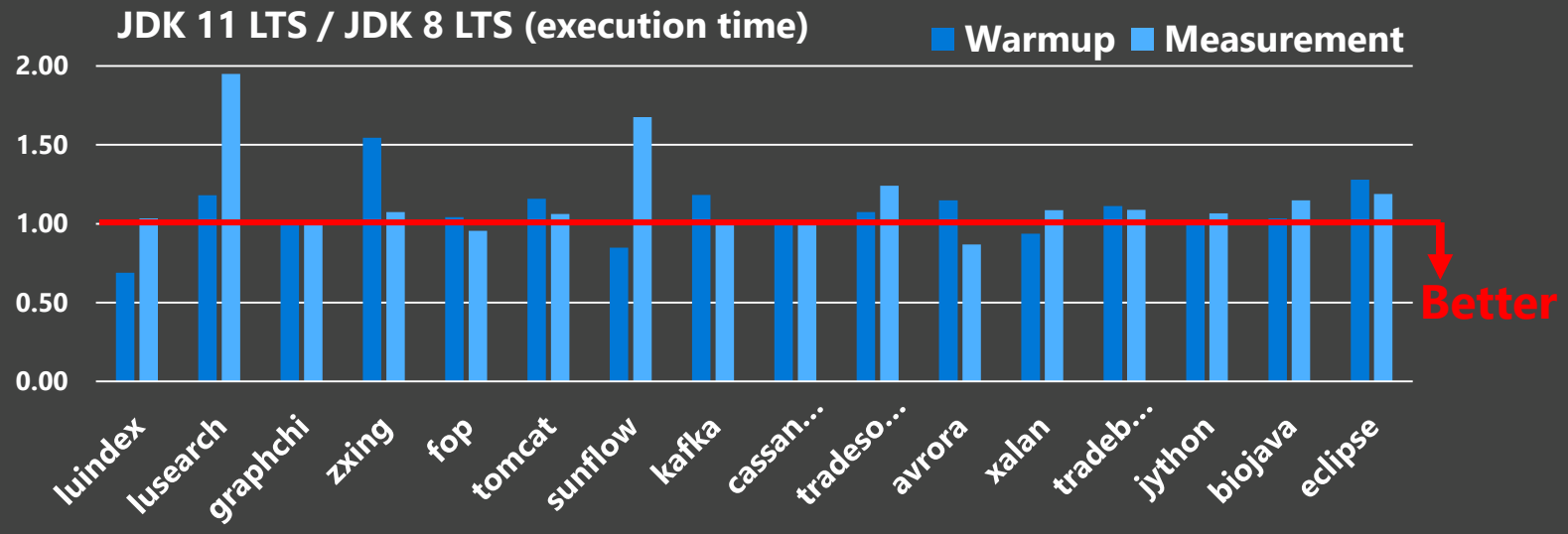




JDK 11 LTS significantly less variability than JDK 8 LTS for responsiveness

DaCapo micro-benchmark: FaaS



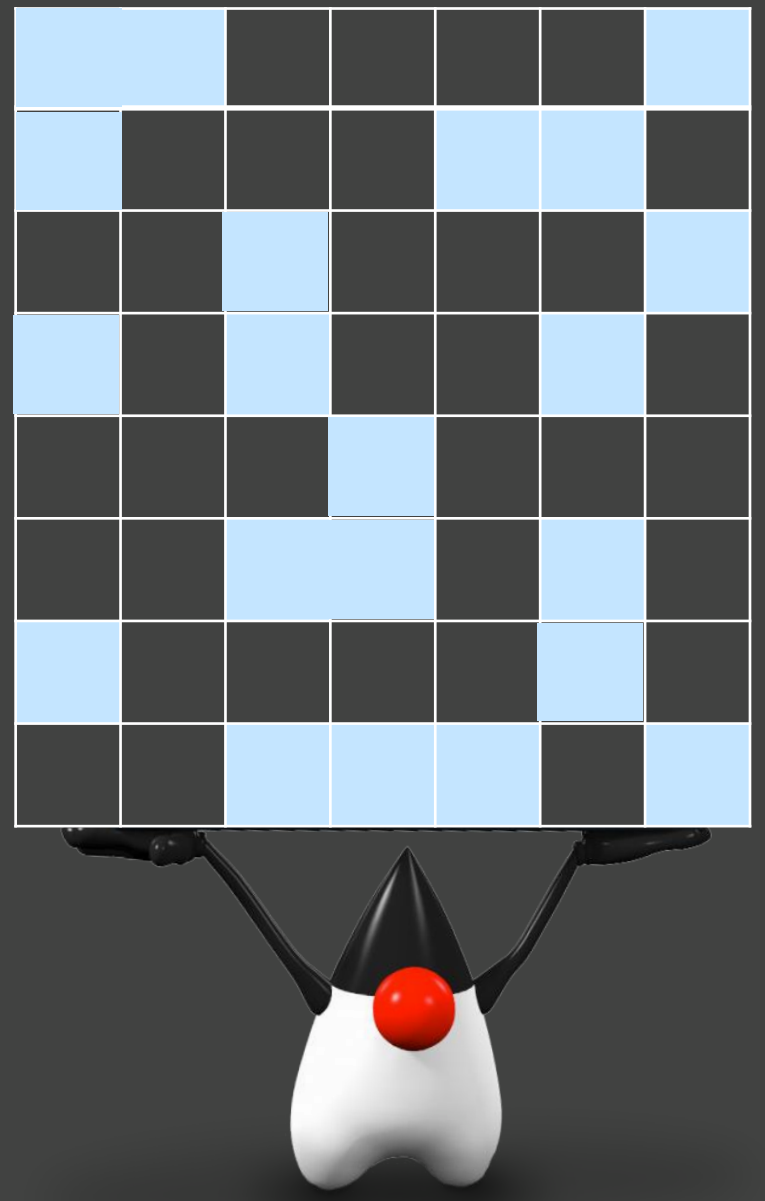


Several components execution time as small as 500ms

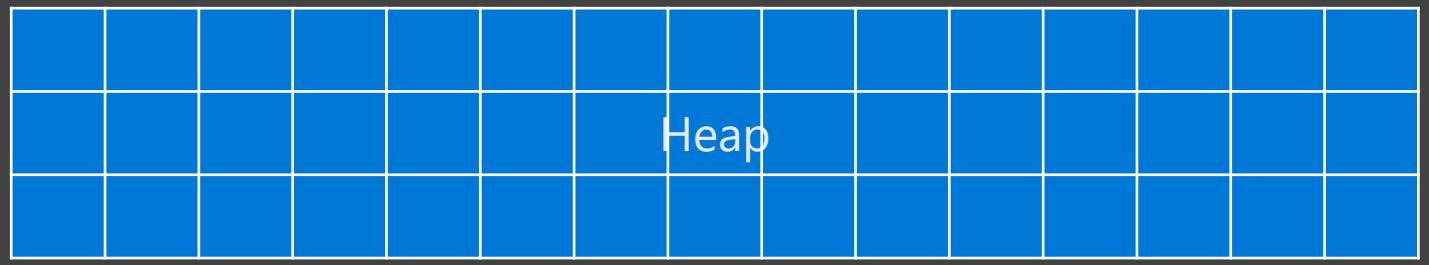
Startup is similar

Execution time with JDK 11 LTS > JDK 8 LTS (G1GC ?)

GC Groundwork

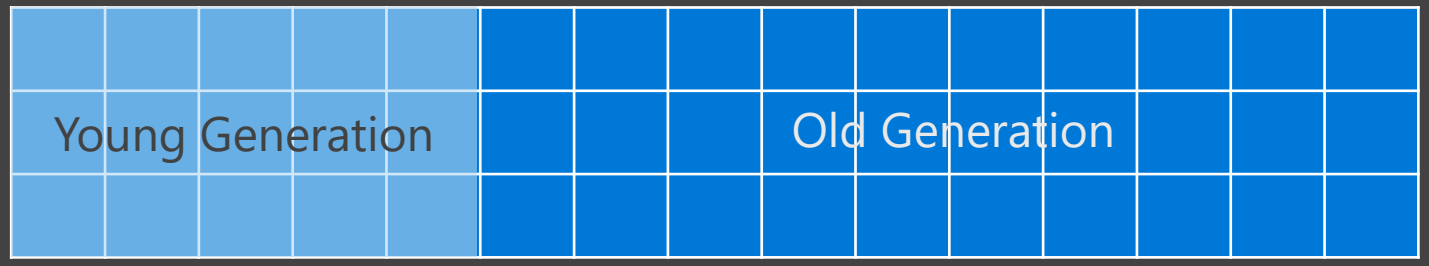


Heap Layout



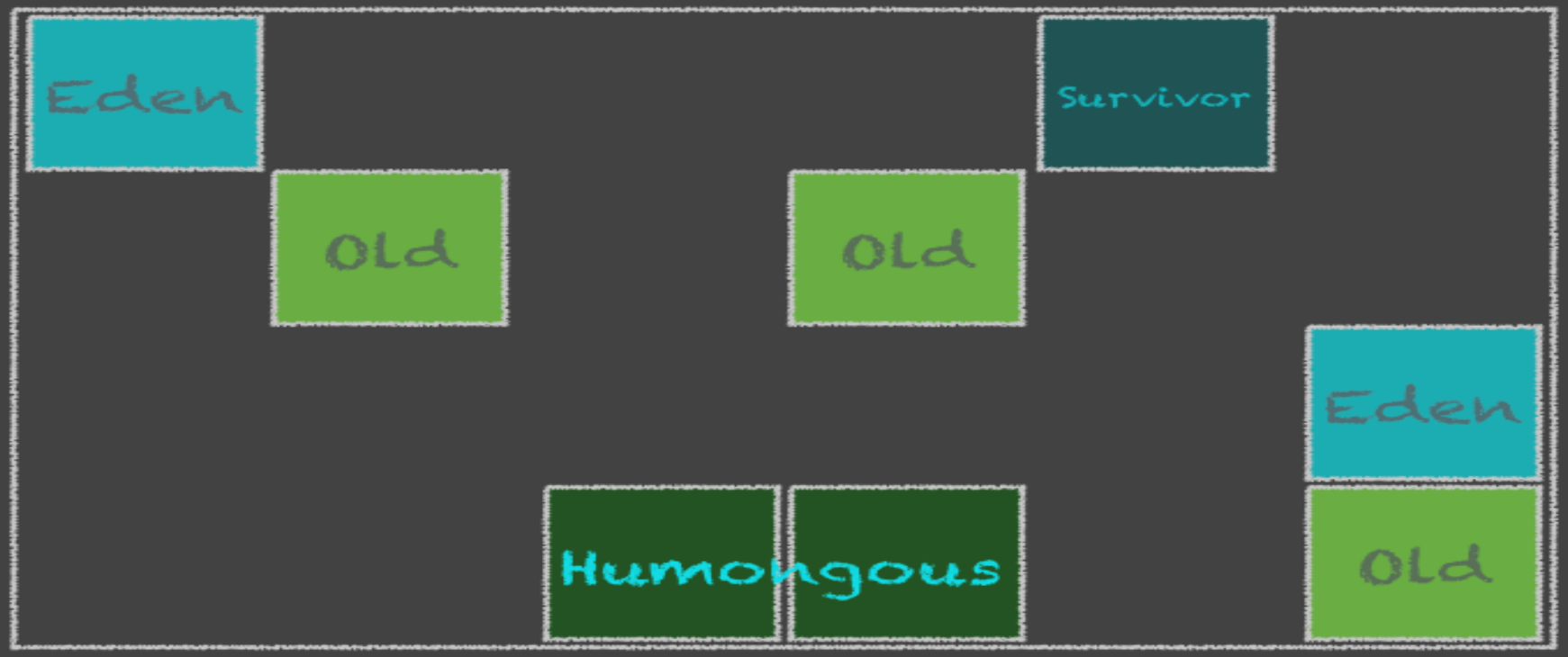
Z GC

Shenandoah GC

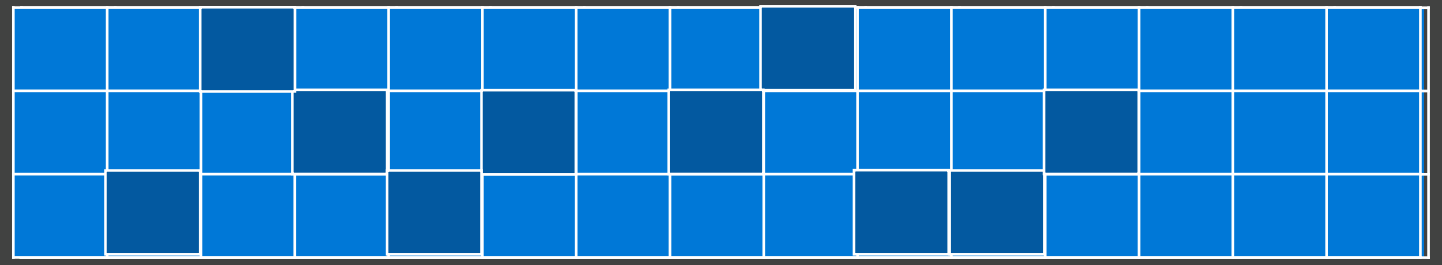


G1 GC

G1 Heap Regions

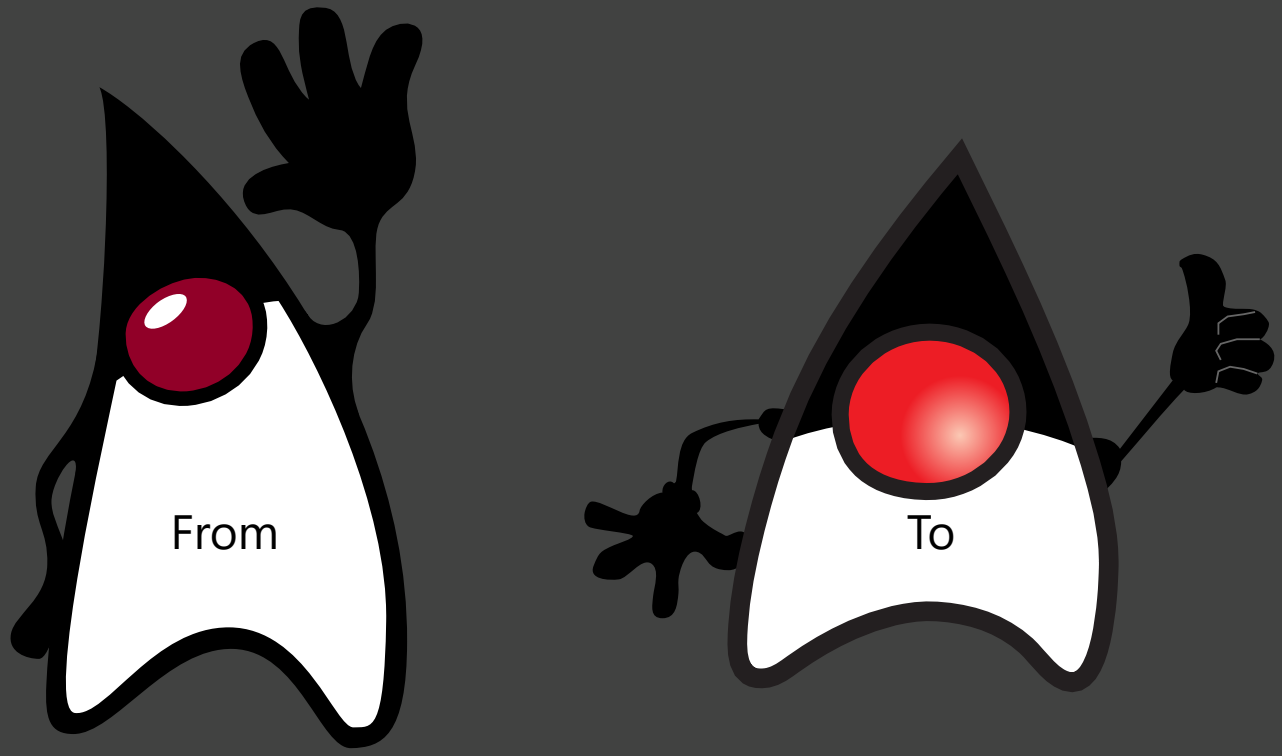


Occupied and Free Regions

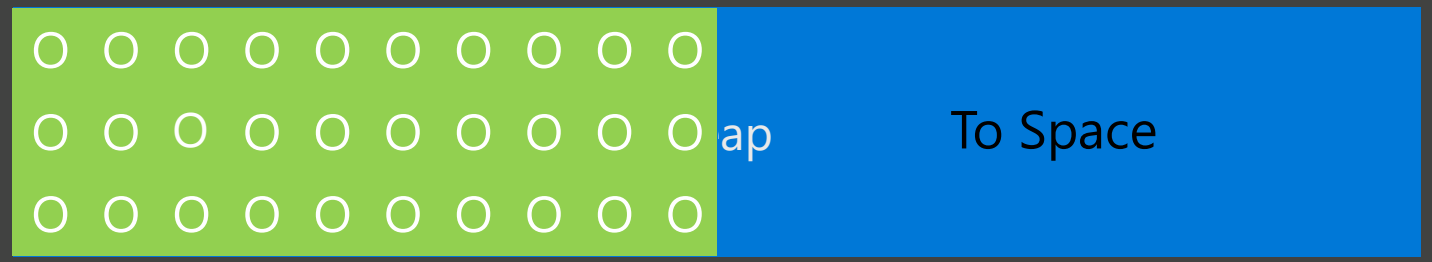


- List of free regions
- In case of generational heap (like G1), the occupied regions could be young, old or humongou

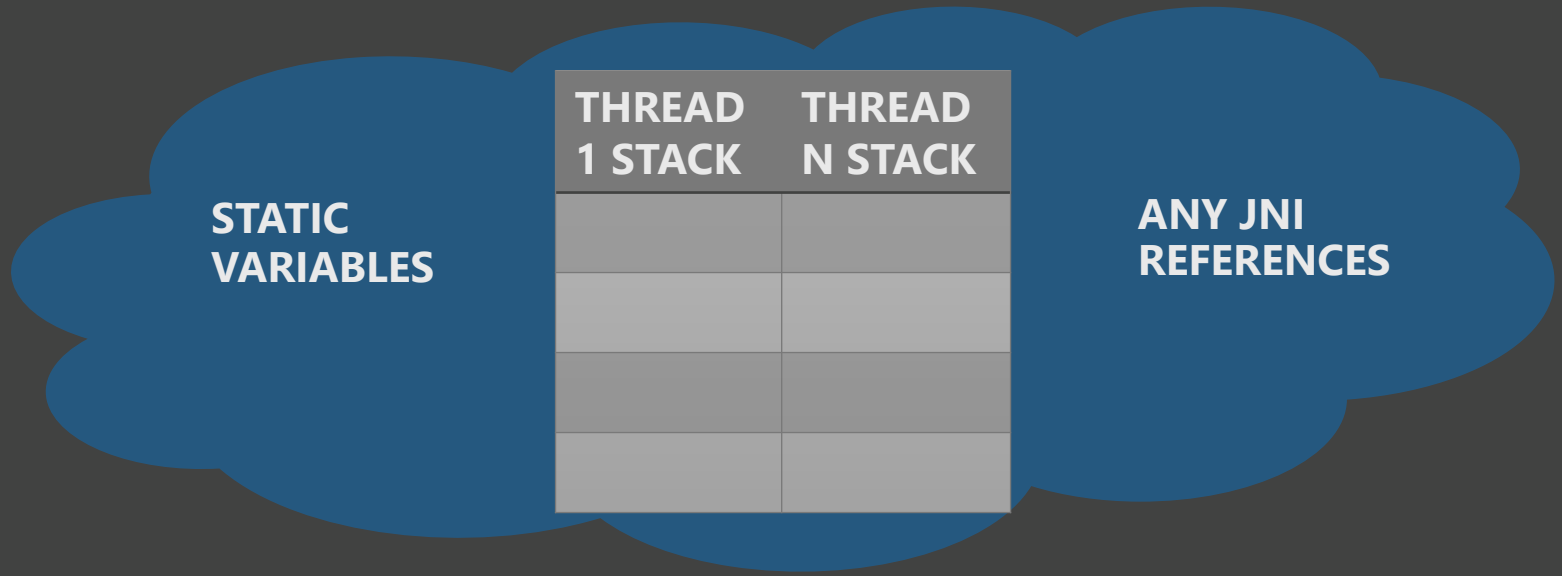
GC Commonalities



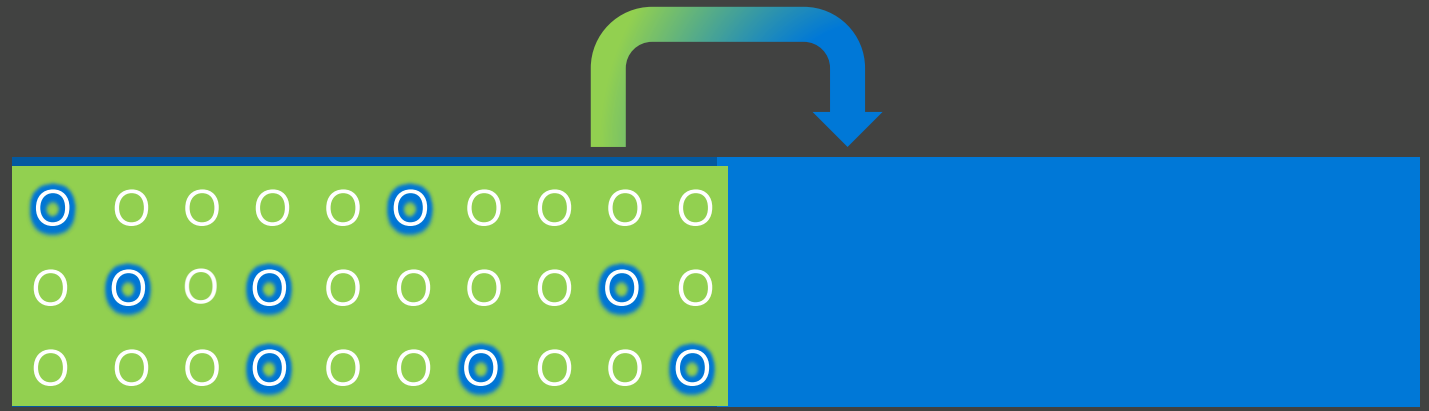
Copying Collector aka Compacting Collector aka Evacuation



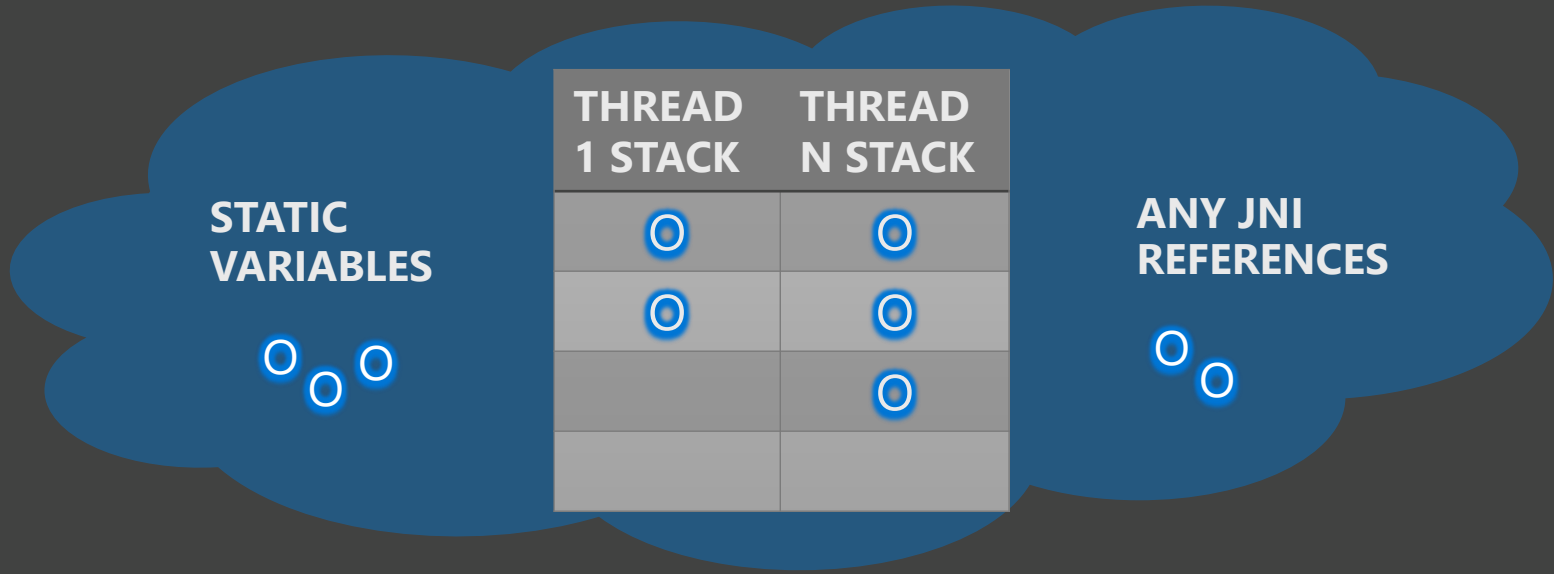
GC ROOTS



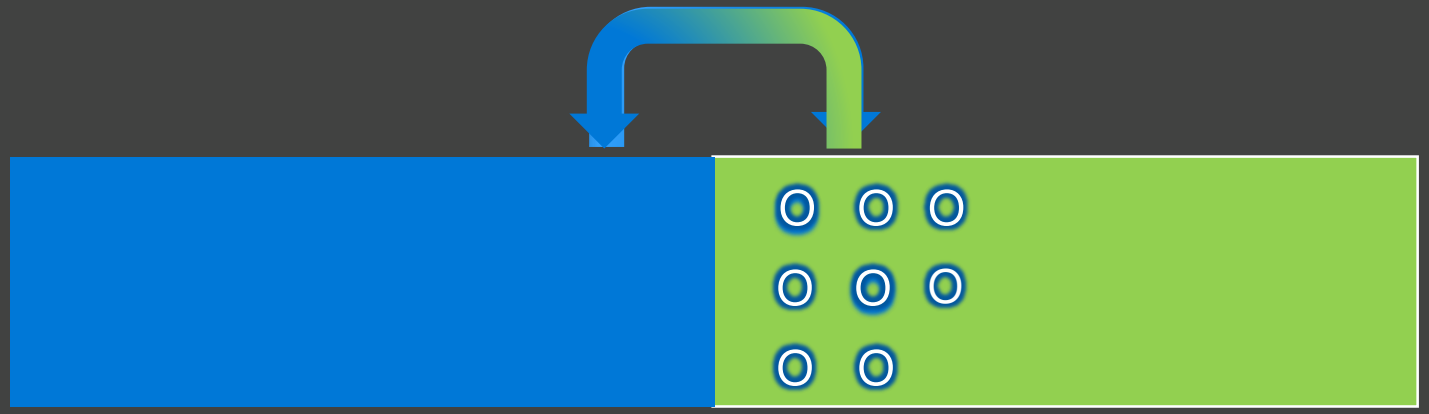
Copying Collector aka Compacting Collector aka Evacuation



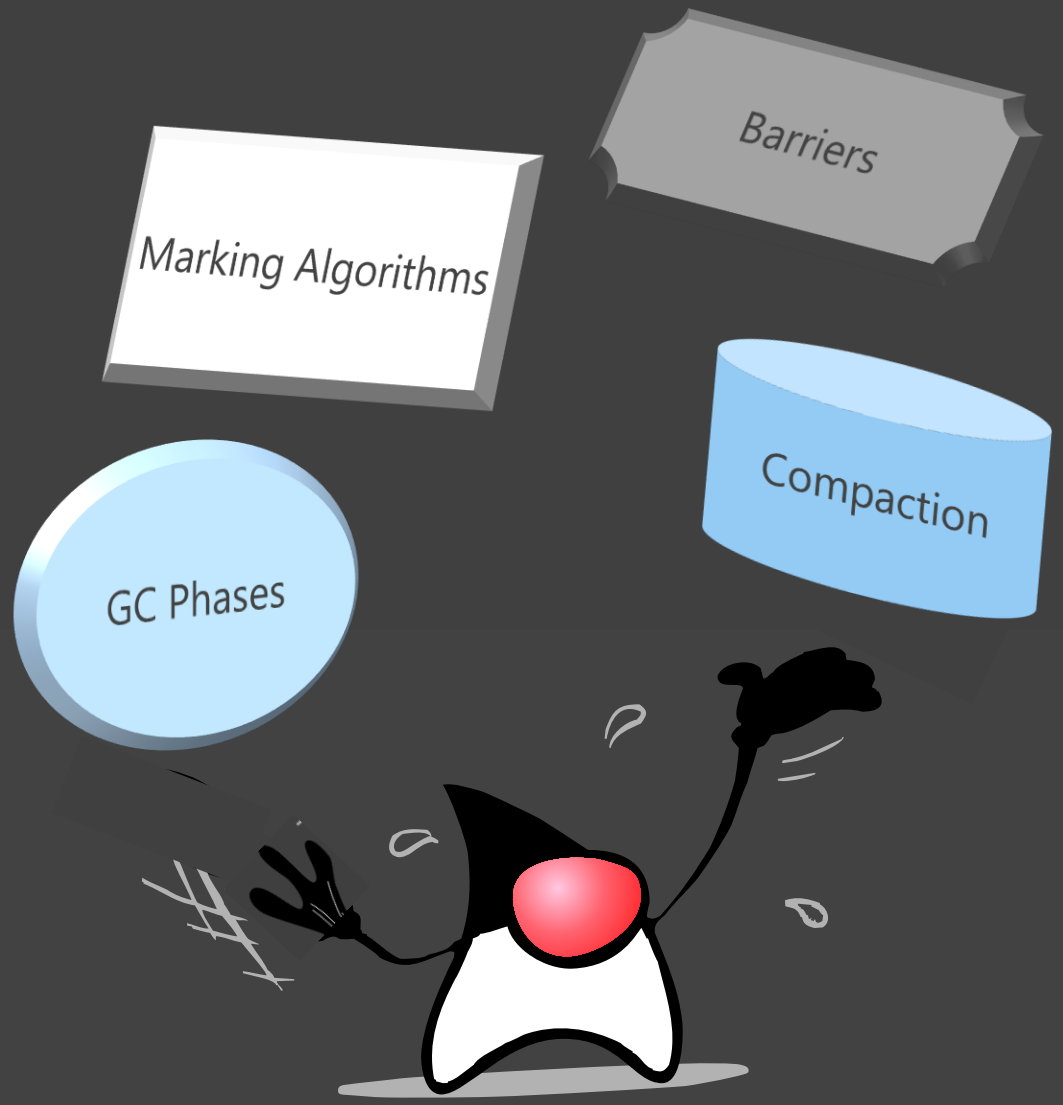
GC ROOTS



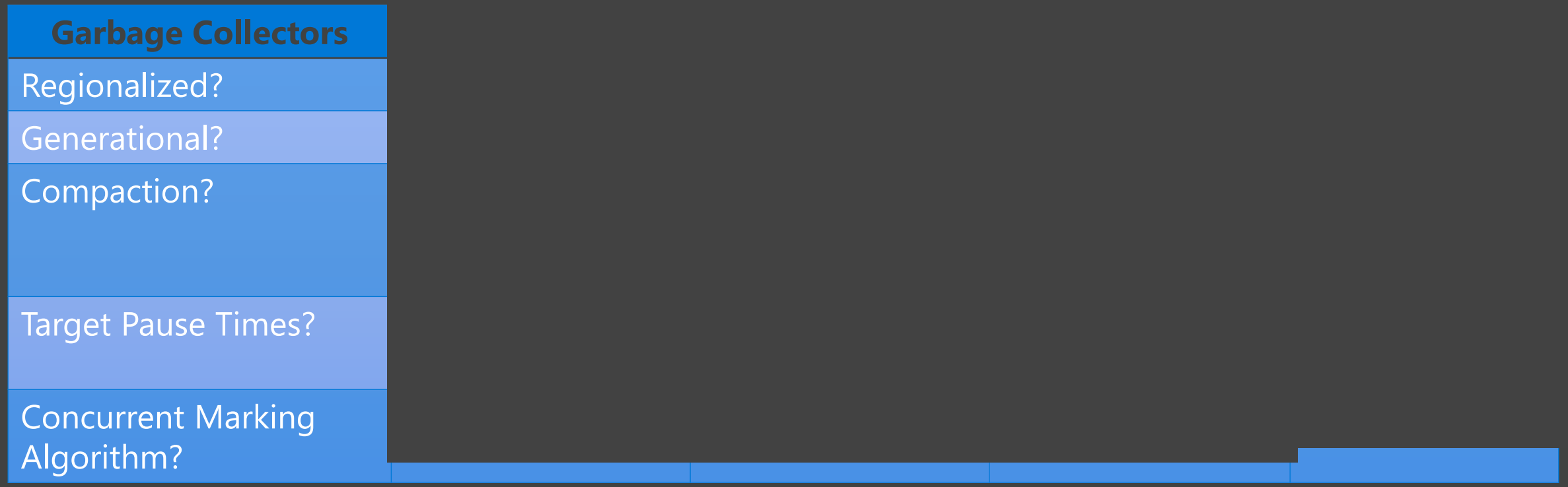
Copying Collector aka Compacting Collector aka Evacuation



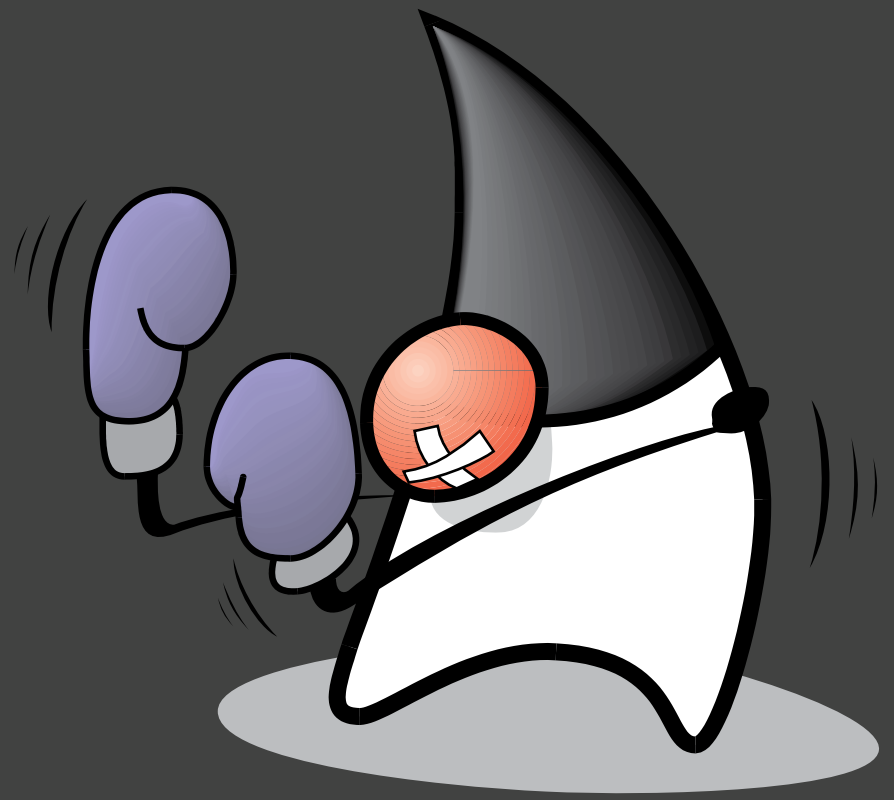
GC Differences



GCs

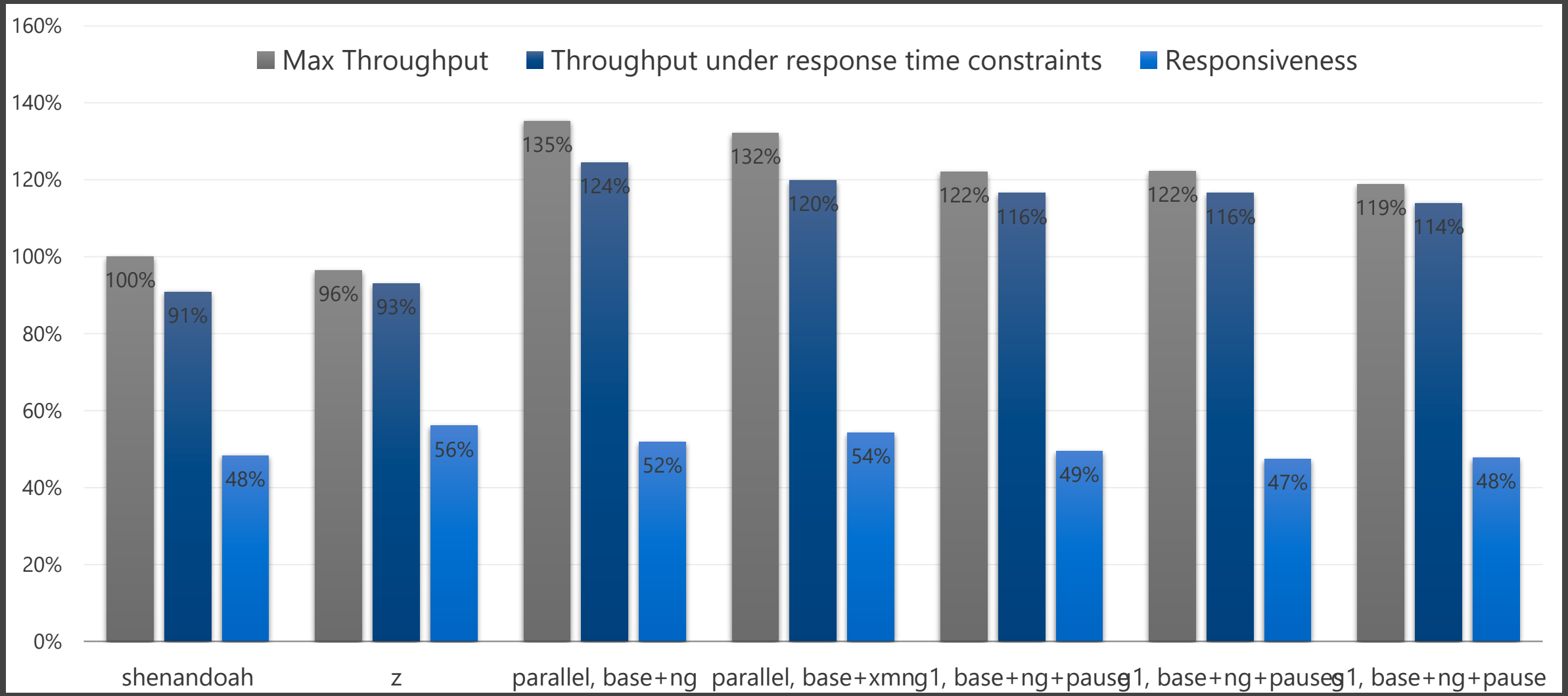

Garbage Collectors
Regionalized?
Generational?
Compaction?
Target Pause Times?
Concurrent Marking Algorithm?

Performance!

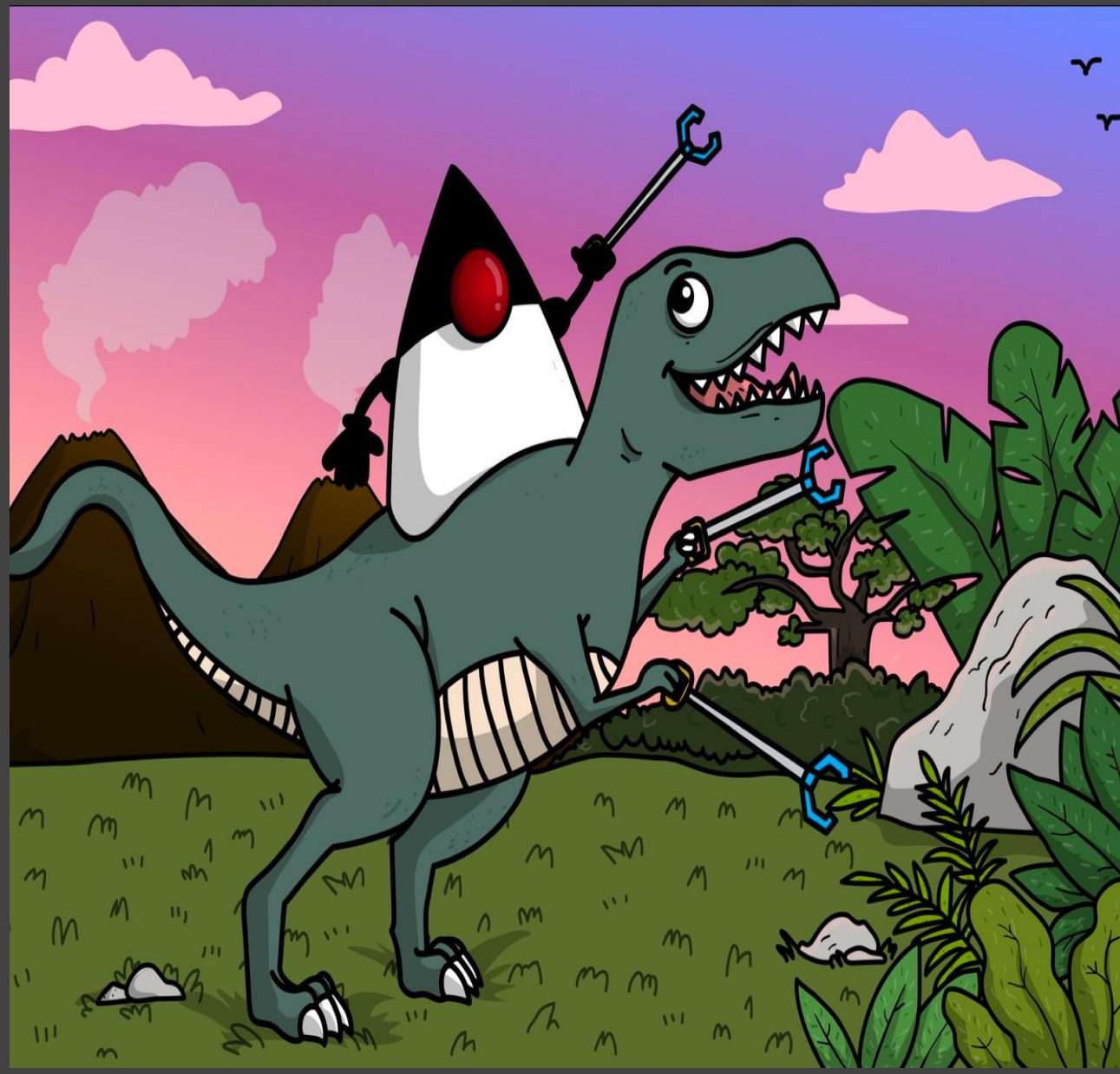


GC Performance

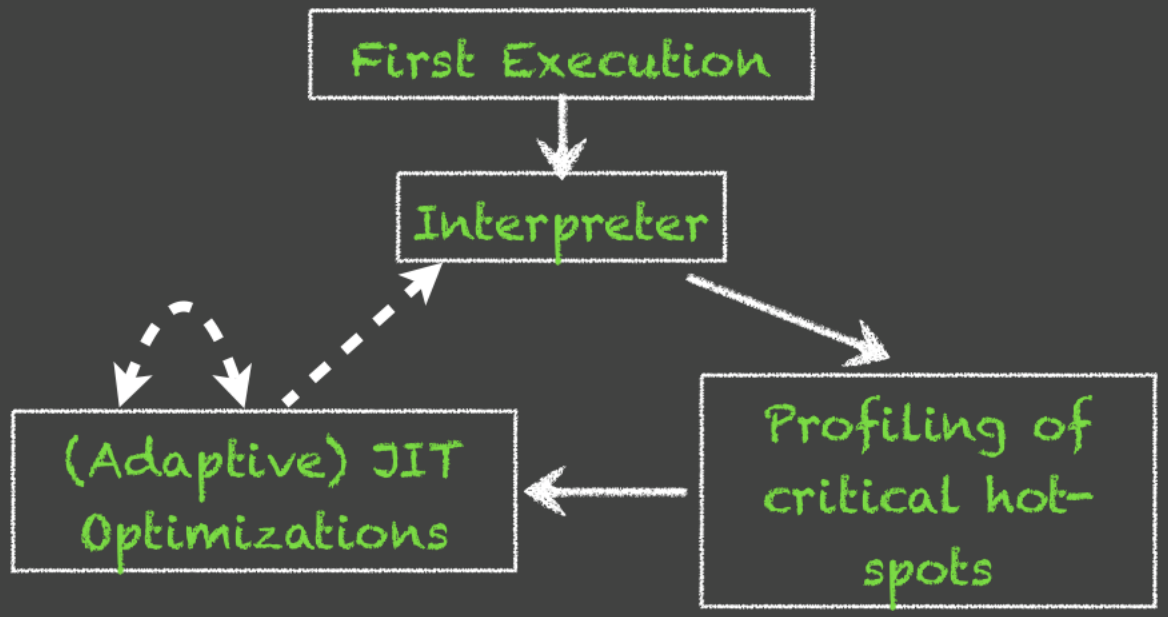
Throughput and Responsiveness – Higher is Better



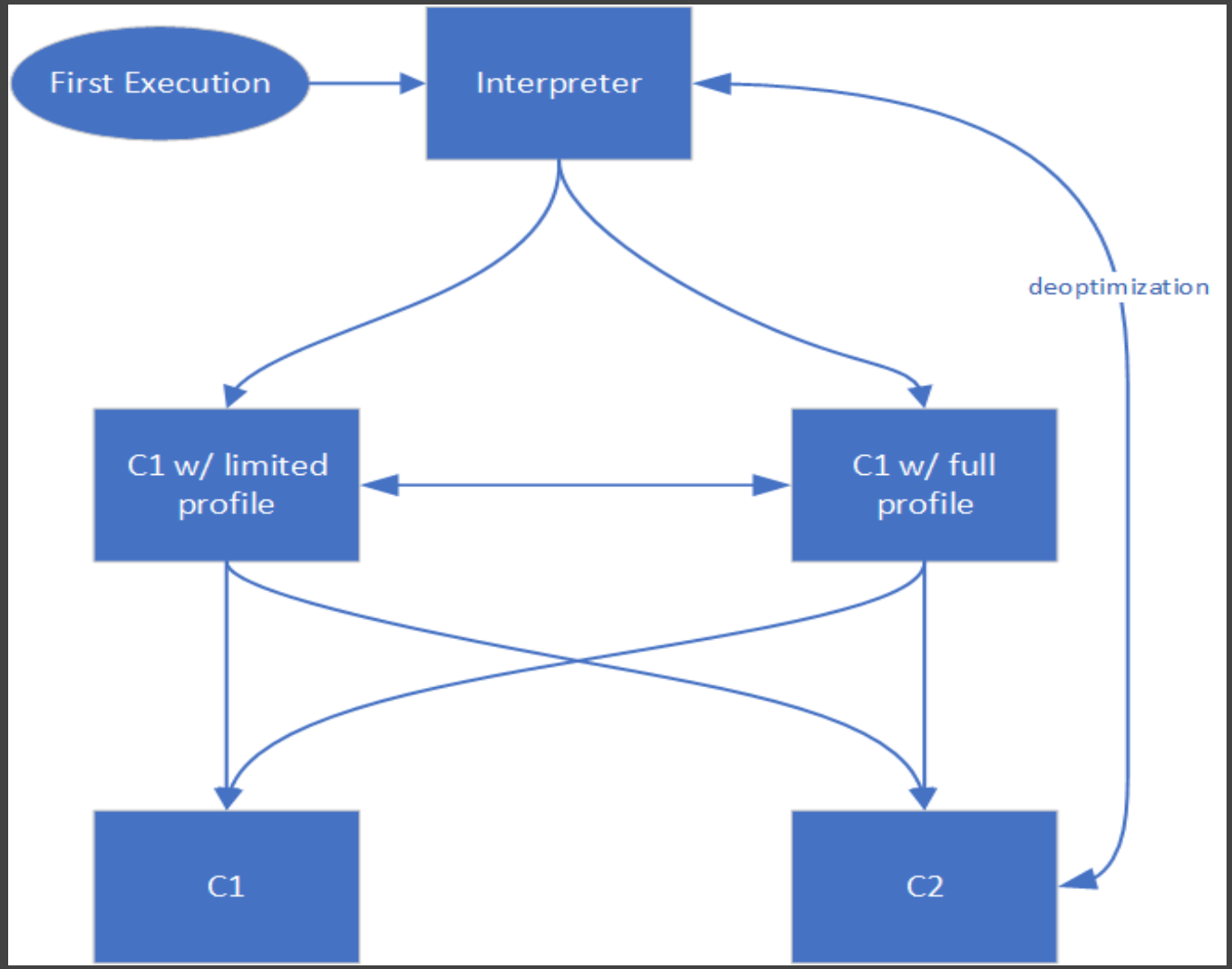
AOT Groundwork



OpenJDK JIT Compilation (prior to Tiered Compilation)

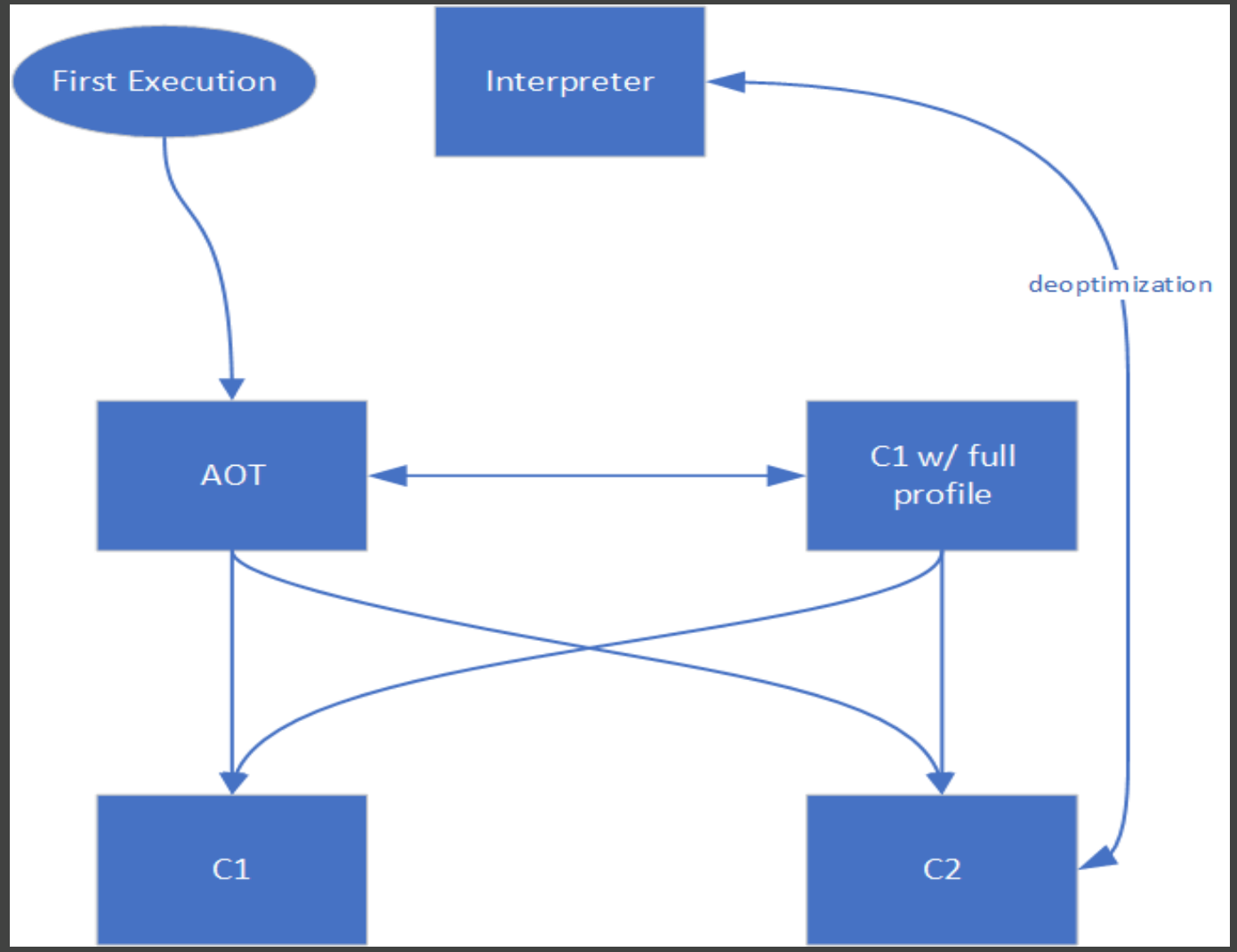


Tiered Compilation without AOT



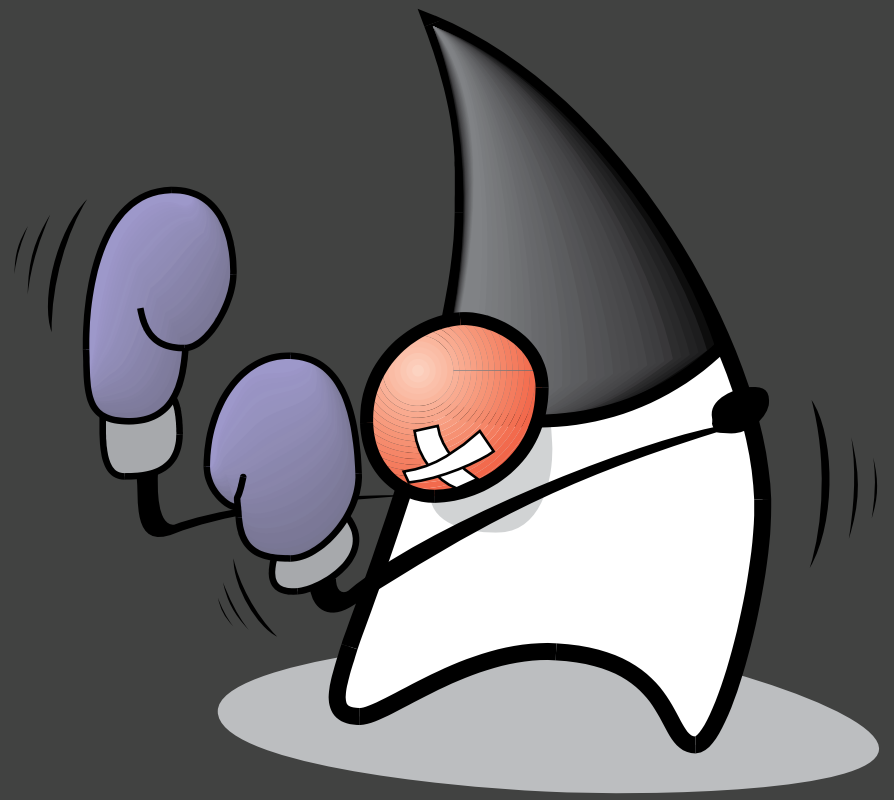
https://devblogs.microsoft.com/java/aot-compilation-in-hotspot-introduction/

Tiered Compilation with AOT



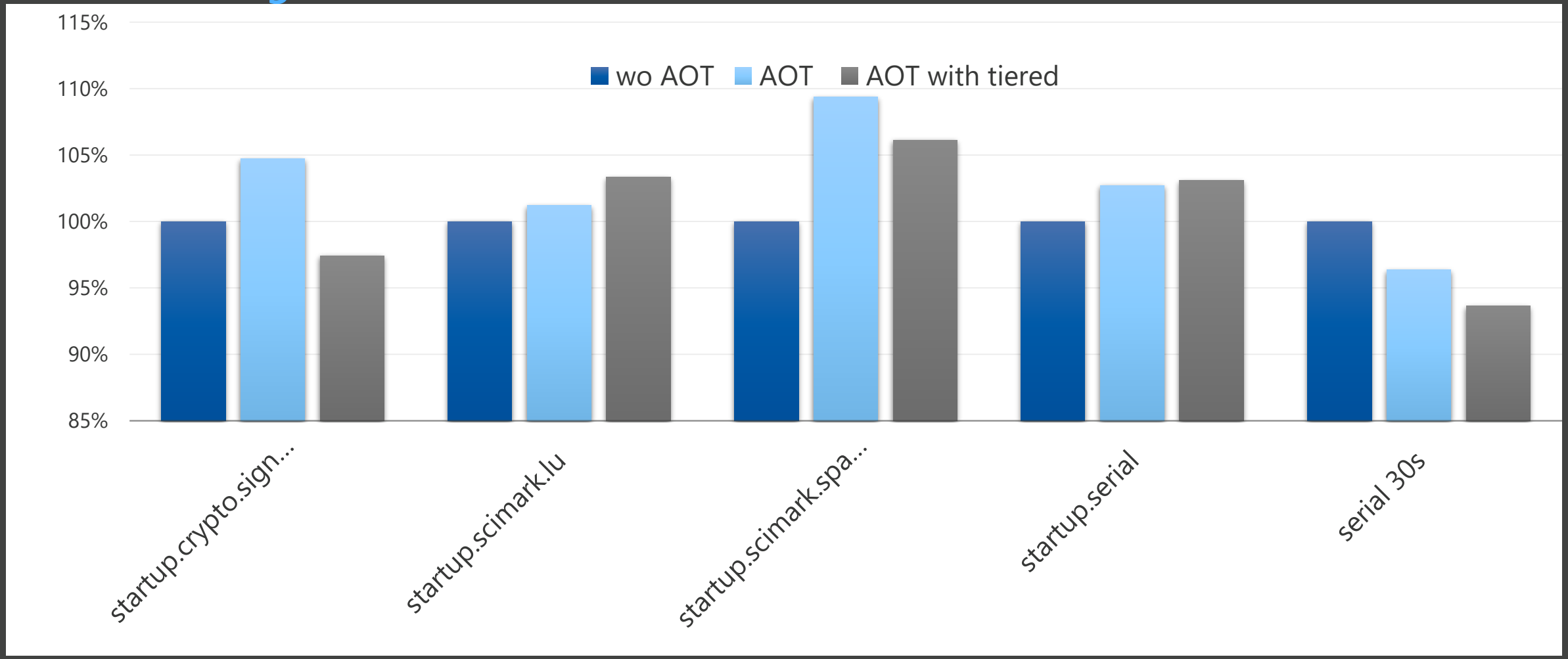
https://devblogs.microsoft.com/java/aot-compilation-in-hotspot-introduction/

Performance!

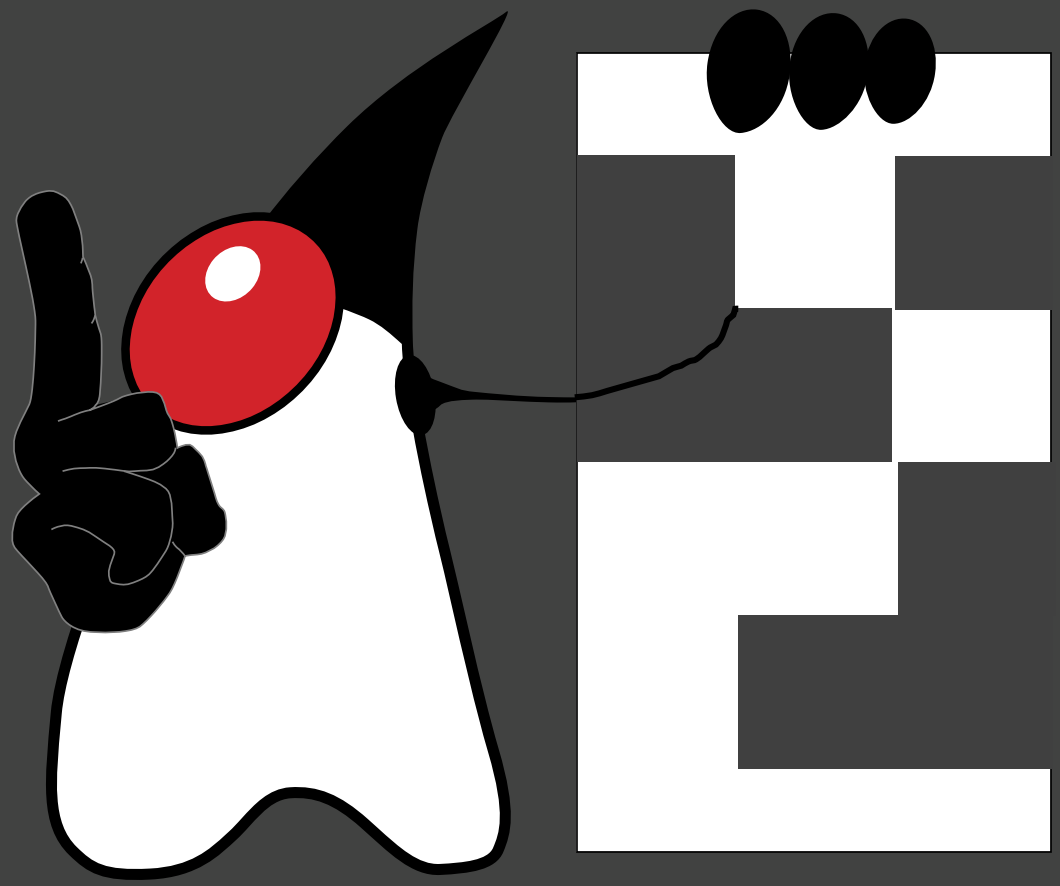


AOT Performance

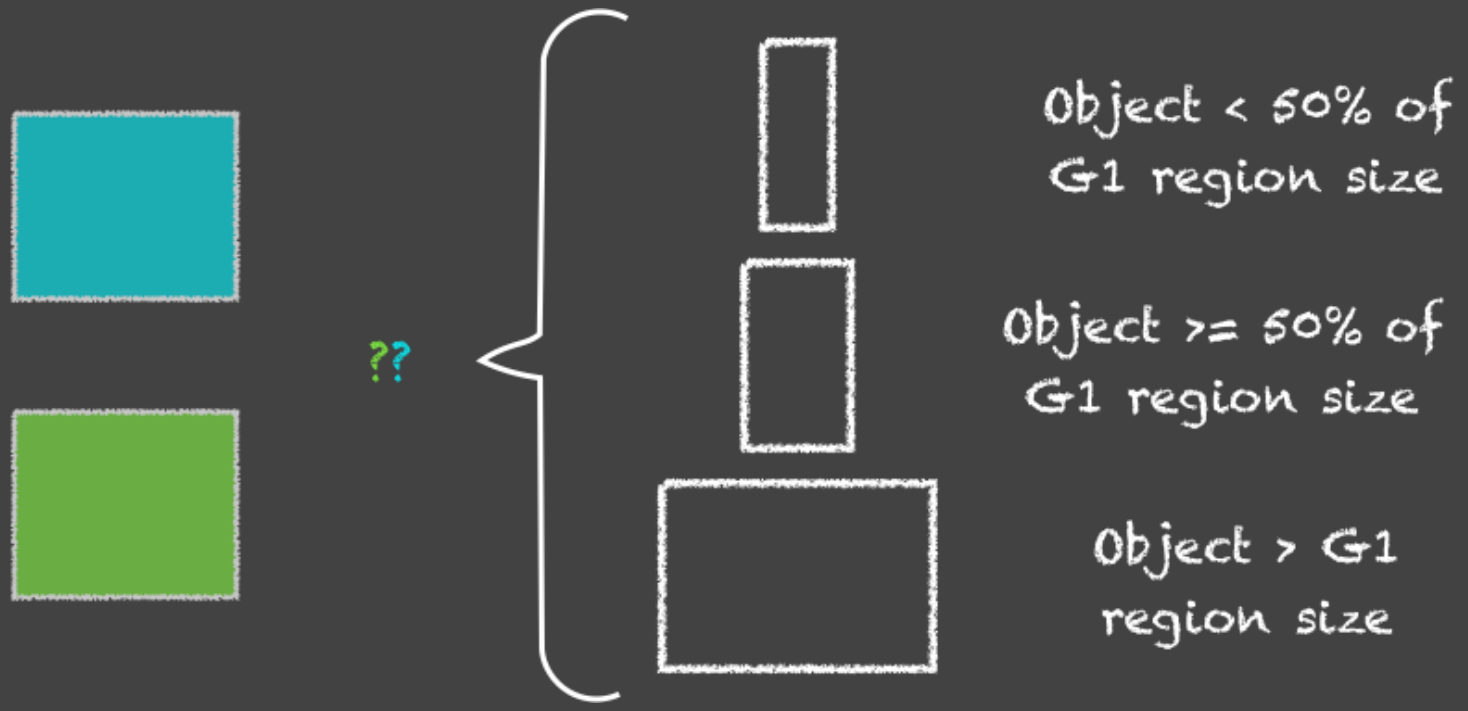
JVM 2008 – Higher is Better



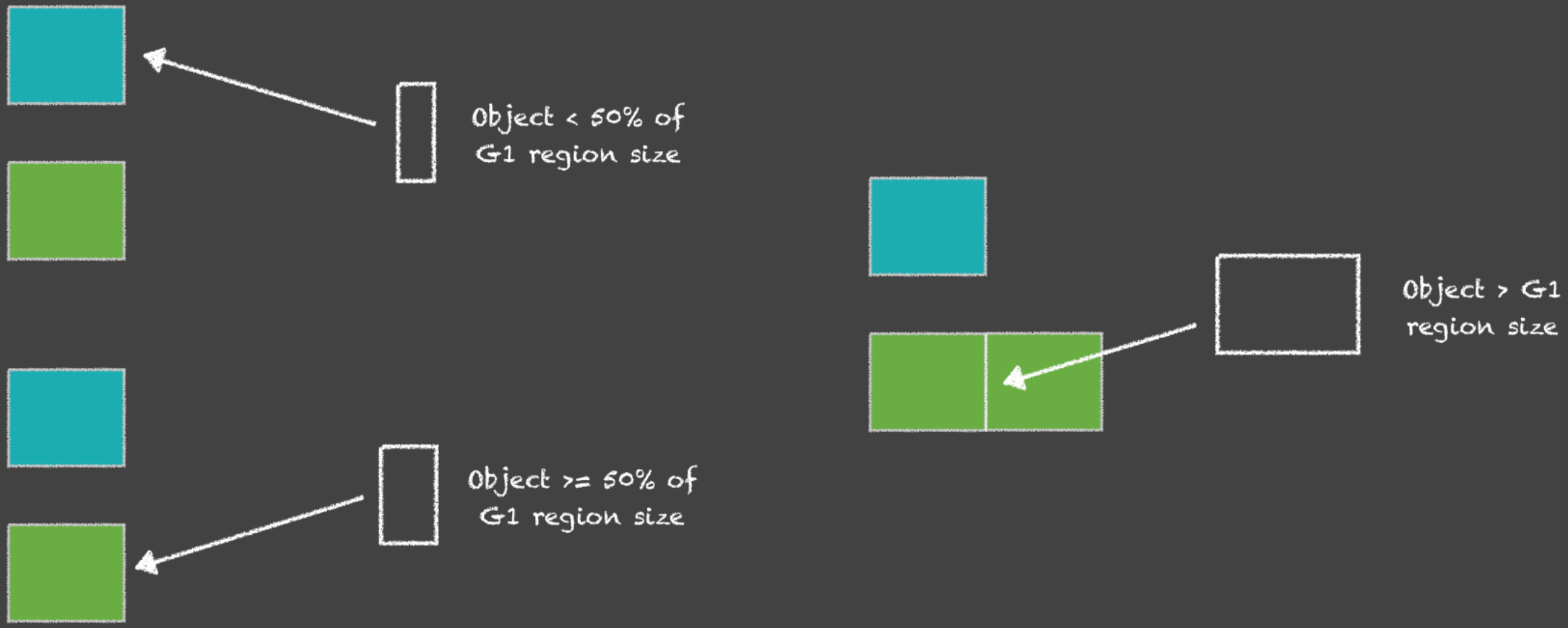
G1 GC and Humongous Objects



What Constitutes a Humongous Object?



What Constitutes a Humongous Region?

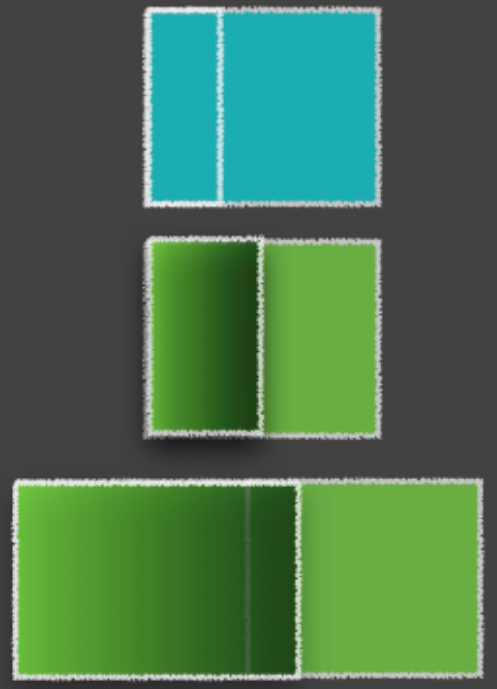


Humongous Objects

Object NOT Humongous

Object Humongous

Object Humongous ->
Needs Contiguous Regions



DaCapo Performance Issue