Low Pause Garbage Collection in HotSpot

John Oliver - john@jclarity.com

11th November 2015

Outline

Introduction

What are our goals.

- Control pause times
- Control heap size
- Control throughput
- Not OOM

What HotSpot GCs are there.

- Serial
- Parallel
- CMS
- iCMS
- G1
- Shenandoah

State of Java GC

Recent Changes

Permgen to Metaspace.

- Permanent stuff is now held in native space
- Simplify configuration
- JEP 156: G1 GC: Reduce need for full GCs
- Strings moved to the heap
- Can tune with:
  - **–XX:MaxMetaspaceSize**
  - **–XX:MetaspaceSize**

G1 Ready

- Stabilised
- Producing consistent results
- Not crashing

Future

The future of GC

- JEP 248: Make G1 the Default Garbage Collector
  - Fundamentally changes the default behavior from high throughput to low pause
- JEP 192: String Deduplication in G1

The future of GC

- JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector

Remove Old GC combinations

- JEP 173: Retire Some Rarely-Used GC Combinations
- JEP 214: Remove GC Combinations Deprecated in JDK 8
  - DefNew + CMS
  - ParNew + SerialOld
  - Incremental CMS

Speculative

- JEP draft: Parallelize the Full GC Phase in CMS
  - `https://bugs.openjdk.java.net/browse/JDK-8130200`
- JEP 271: Unified GC Logging

Under The Hood

Other Collectors

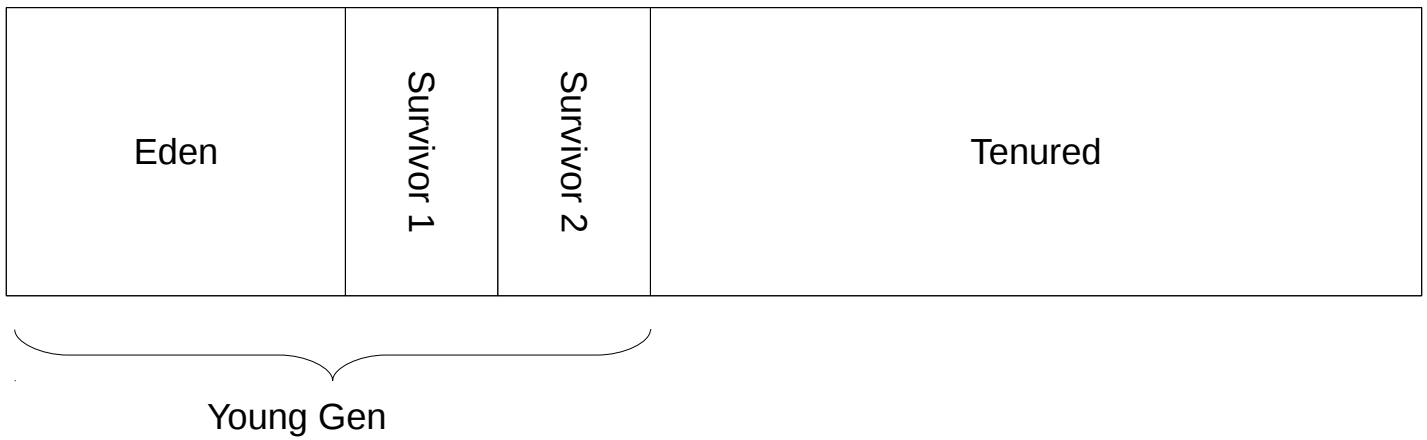
Parallel - High throughput
CMS - Low pause

G1

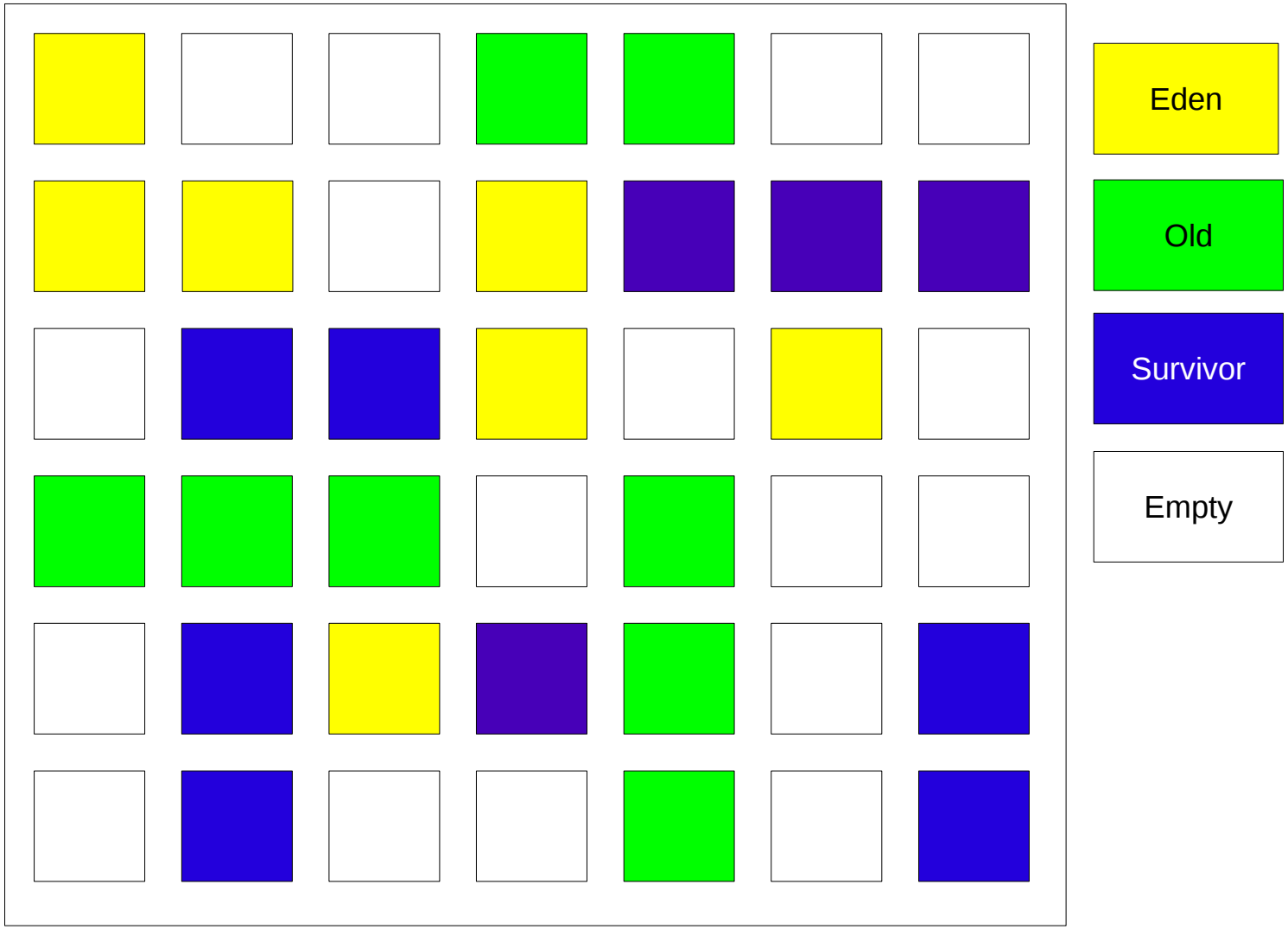
The Rise of G1

- Easier to tune (-**XX:MaxGCPauseMillis=N**)
- Can set pause goals
- Compacting

Heap Layout And GC



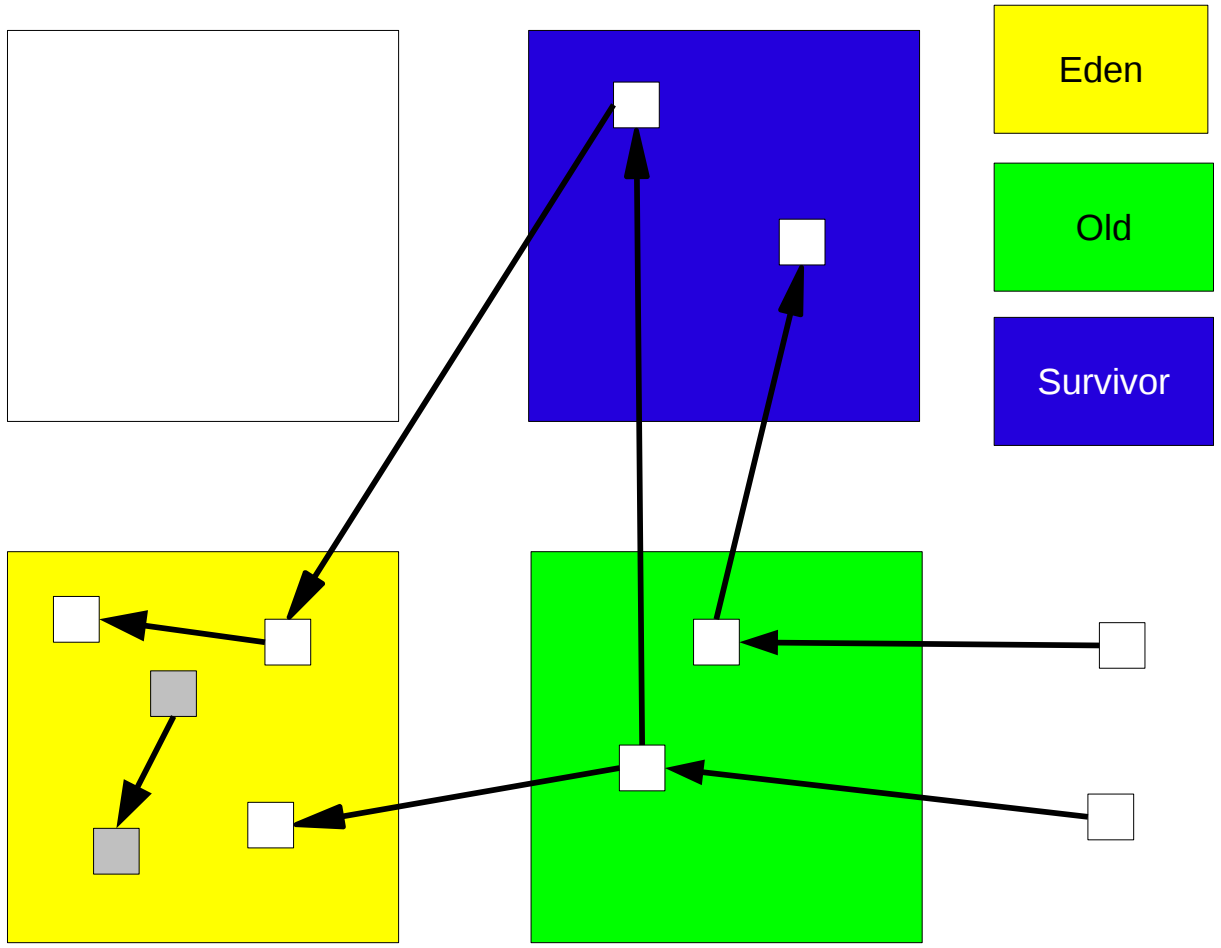
G1 Heap



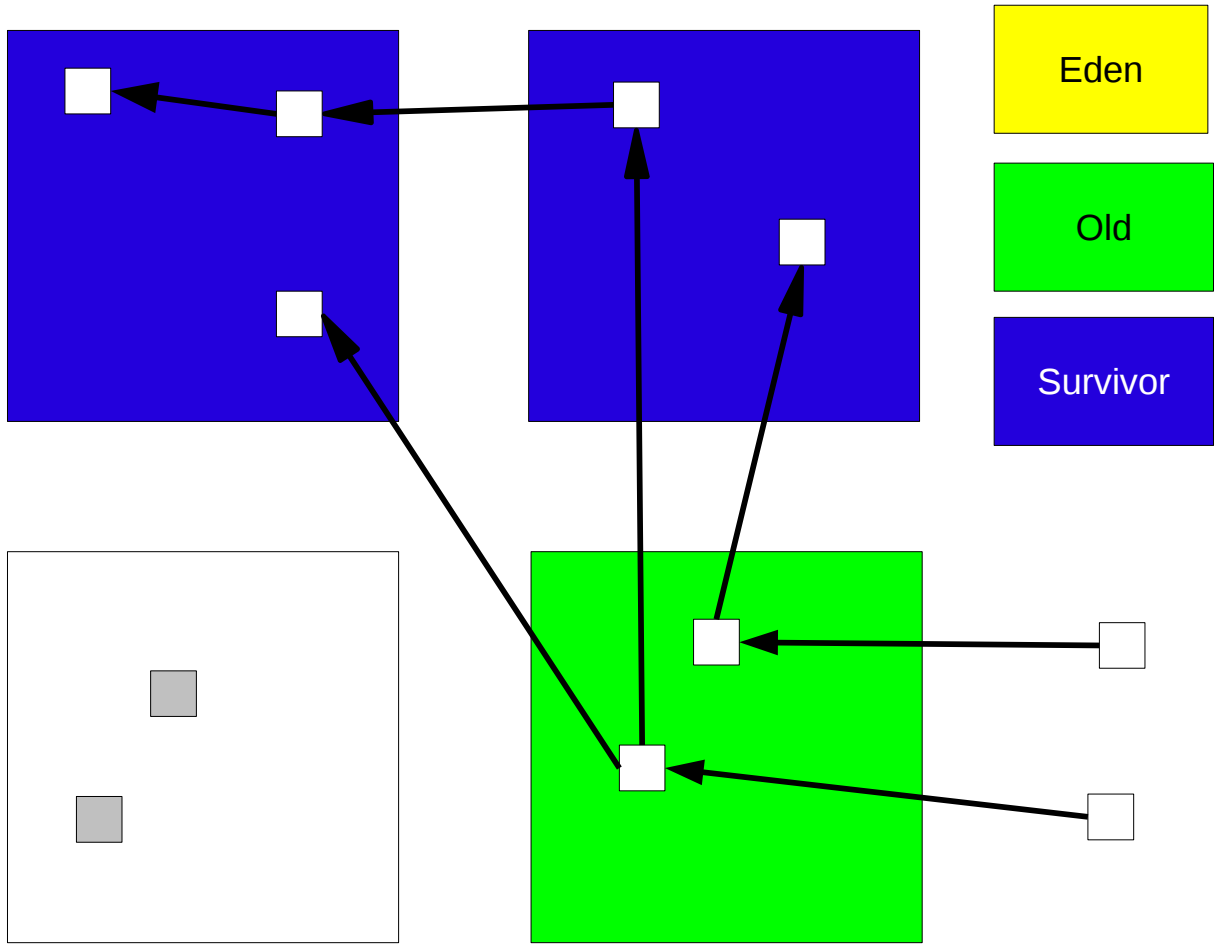
G1 How it works

- Mark and evacuate style
- Snapshot at the beginning
  - Scan from roots
  - Track mutations in the graph

G1 Heap



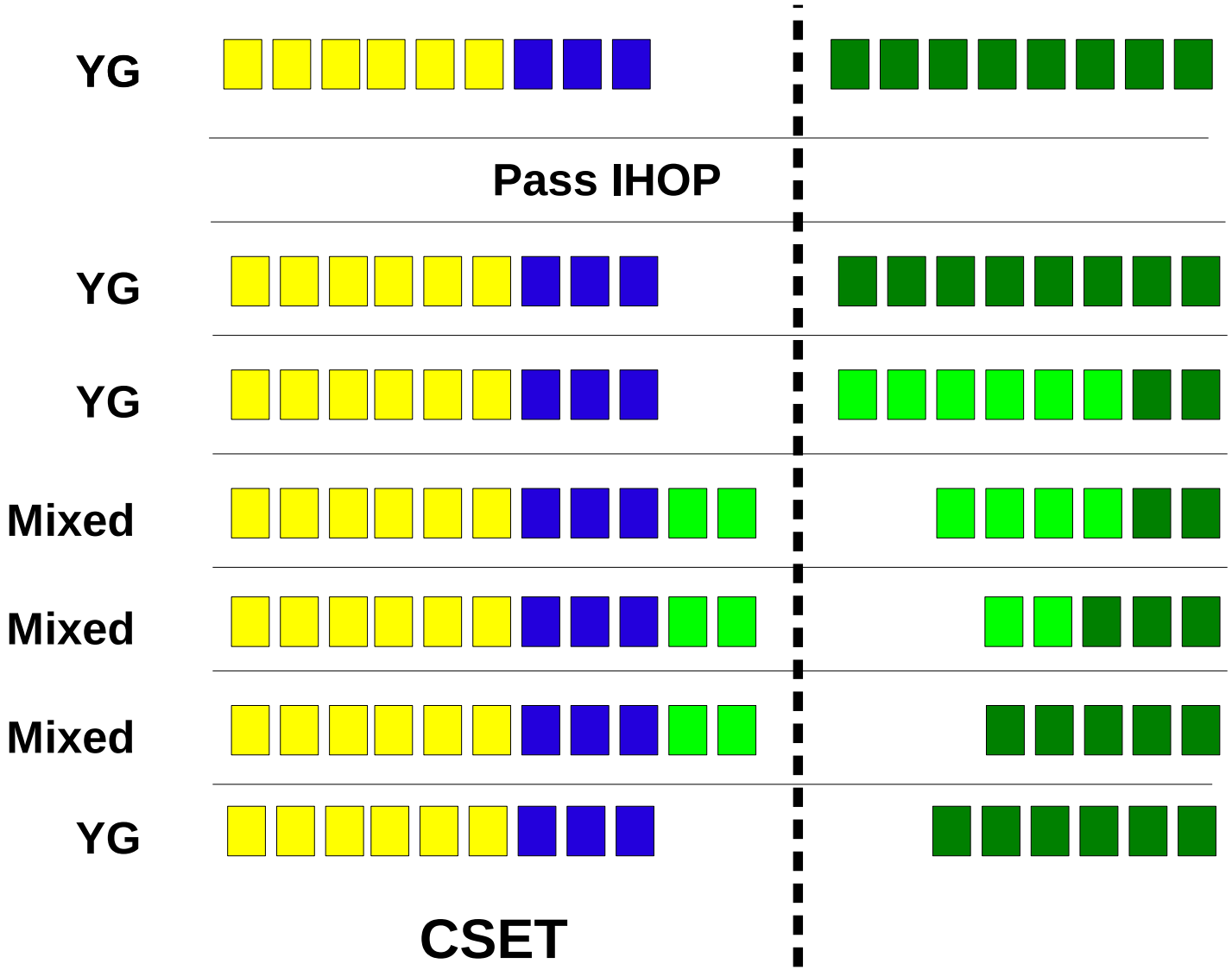
G1 Heap Evacuation



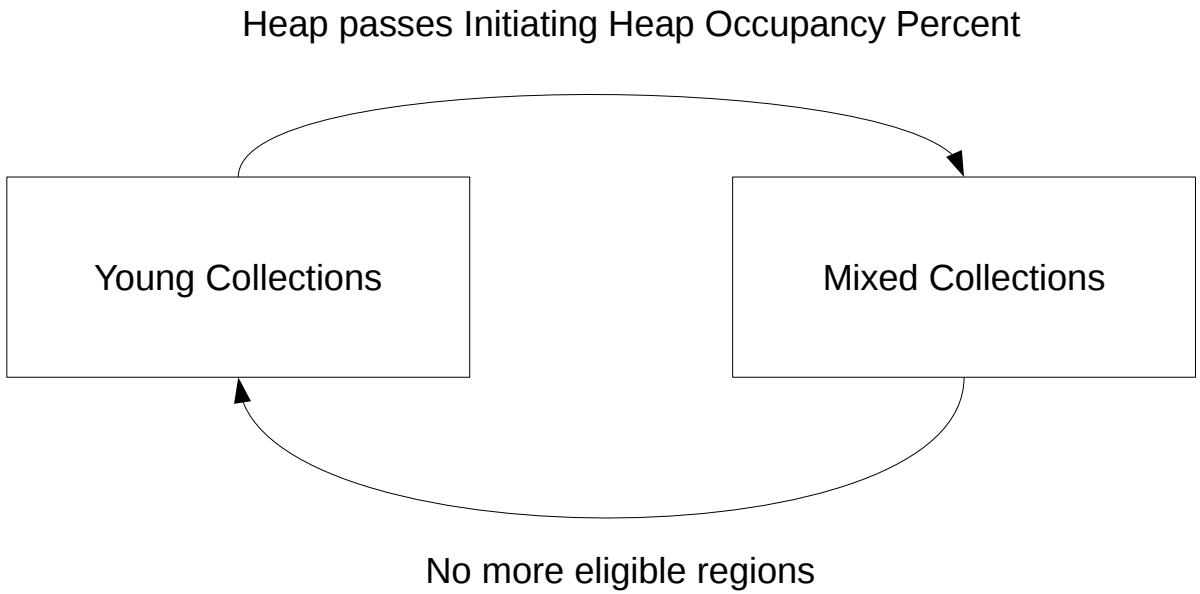
G1 How it works

- Mark concurrently
- Pause to evacuate
- Don't evacuate all at once
- Divide the evacuation work up and every time we have a YG pause, do a bit of OG work

G1 How it works



G1 How it works



G1 How it works

- Start Marking
  - **-XX:InitiatingHeapOccupancyPercent=n**
- Mixed GC until no more eligible regions
  - **-XX:G1MixedGCLiveThresholdPercent**

Tuning Parameters

Xmx/Xms	Heap Size
MaxGCPauseMillis	Target pause limit
G1MixedGcCountTarget	Target number of mixed garbage collections
G1OldCSetRegionThresholdPercent	Limit on the number of old regions in a cset
G1MixedGCLiveThresholdPercent	Threshold for an old region to be included in a mixed garbage collection cycle
G1HeapWastePercent	Level of floating garbage you are ok with

Ref: `http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html` (`http://bit.ly/1AC7JDZ`)

The problem

- The whole of YG is cleaned during every GC
- YG is a low bound on how low you can get your pauses
- If Ergonomics does not reduce YG sufficiently, sucks to be you
- Only way to fix this is reduce the YG workload
- Best way found to do this is forcibly reduce the size of YG.

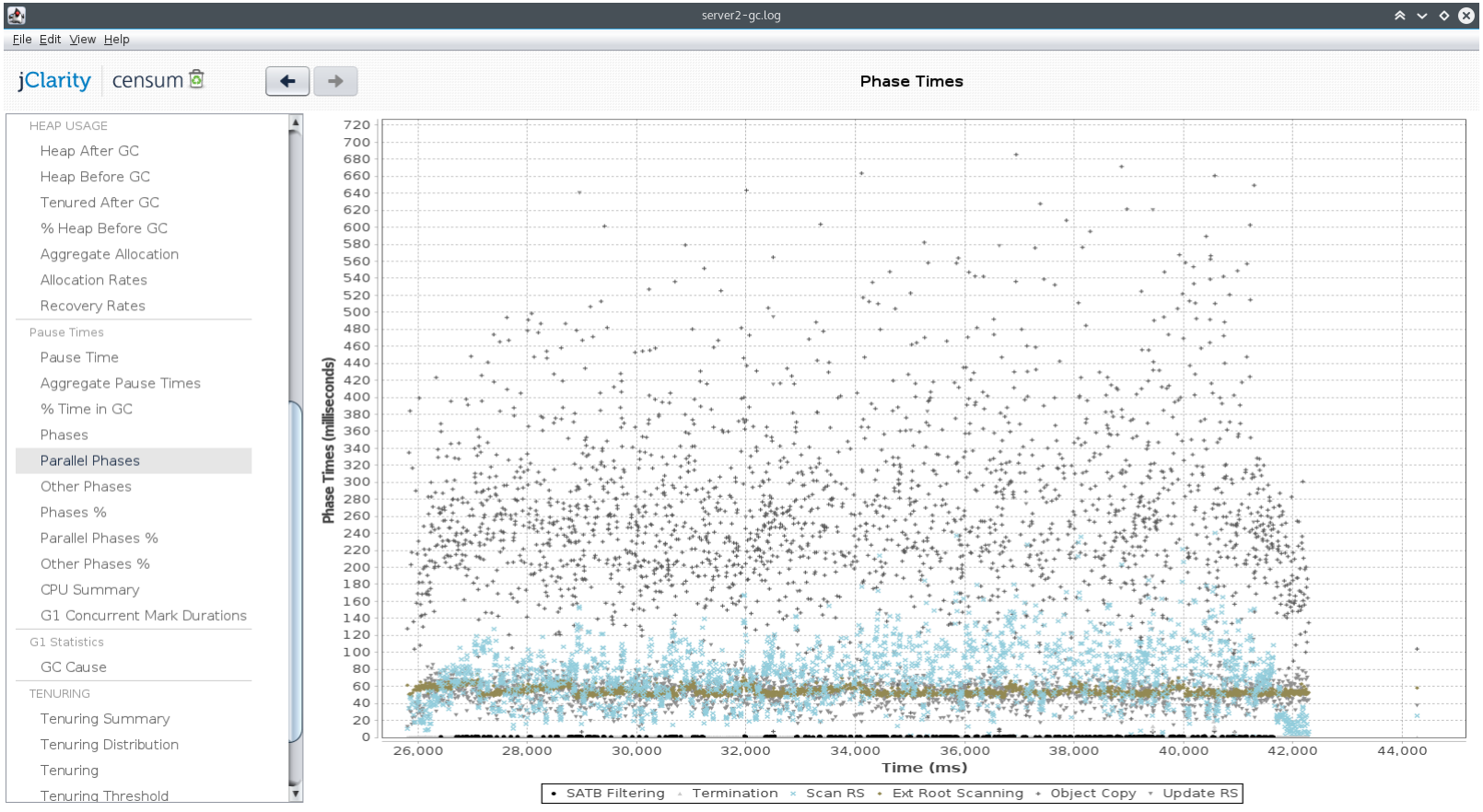
The problem

- “Object Copy” and “Ext Root Scanning” tend to dominate
- Object copy: Time spent copying live objects, when evacuating regions.
- Ext Root Scanning: Time spent scanning external roots (registers, thread stacks, etc)

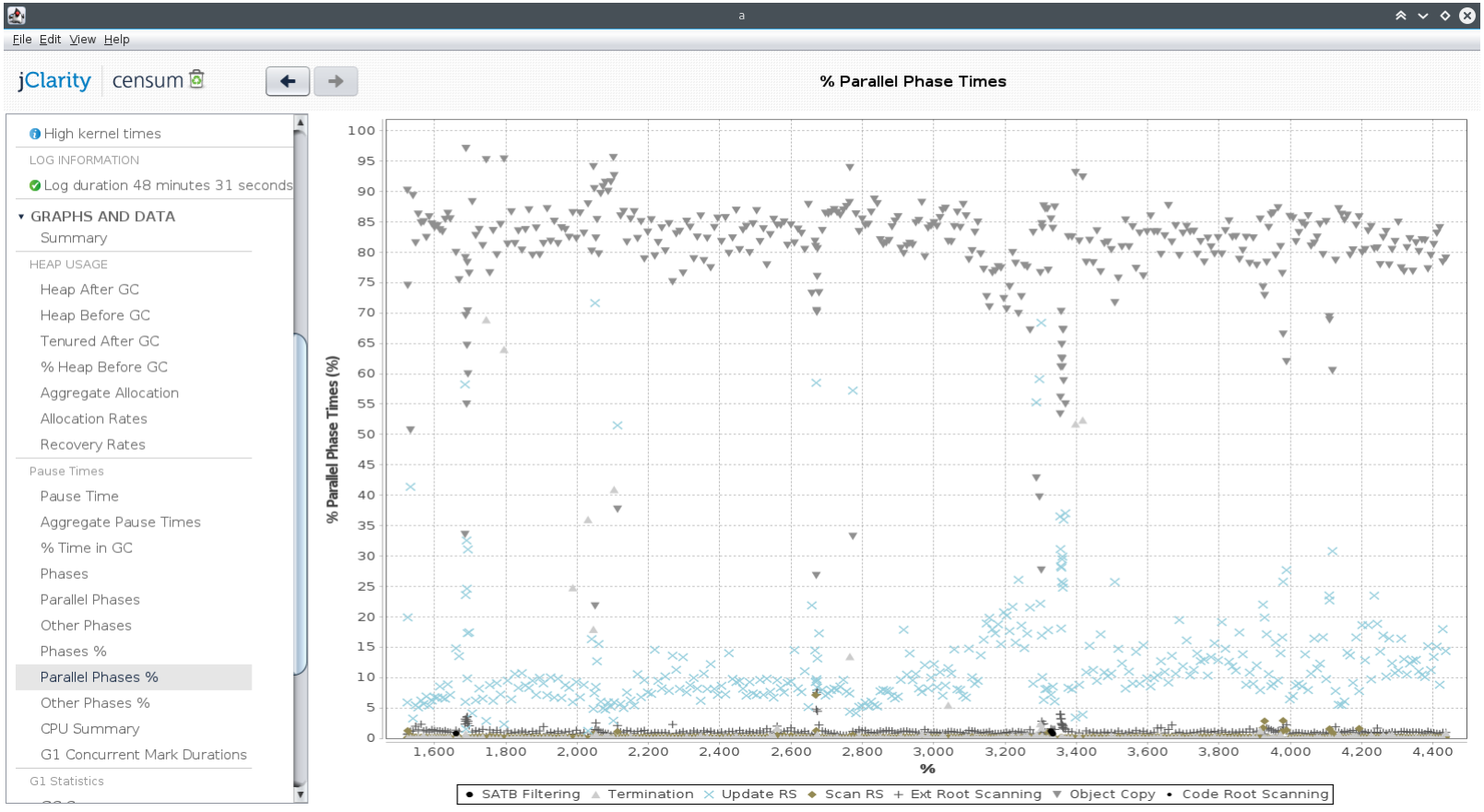
The problem

- On our log DB (many of these are older logs):
  - 40% of all "Parallel Time" is spent in Object Copy
  - 50% in Ext Root Scanning
- However Object Copy seems to dominate apps that are under GC pressure
- Little can be done to speed up this process.

The problem


server2-gc.log
File Edit View Help
jClarity censum
Phase Times
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
% Heap Before GC
Aggregate Allocation
Allocation Rates
Recovery Rates
Pause Times
Pause Time
Aggregate Pause Times
% Time in GC
Phases
Parallel Phases
Other Phases
Phases %
Parallel Phases %
Other Phases %
CPU Summary
G1 Concurrent Mark Durations
G1 Statistics
GC Cause
TENURING
Tenuring Summary
Tenuring Distribution
Tenuring
Tenuring Threshold
Phase Times (milliseconds)
720 700 680 660 640 620 600 580 560 540 520 500 480 460 440 420 400 380 360 340 320 300 280 260 240 220 200 180 160 140 120 100 80 60 40 20 0
26,000 28,000 30,000 32,000 34,000 36,000 38,000 40,000 42,000 44,000
Time (ms)
SATB Filtering
Termination
Scan RS
Ext Root Scanning
Object Copy
Update RS

The problem


a
File Edit View Help
jClarity censum
% Parallel Phase Times
High kernel times
LOG INFORMATION
Log duration 48 minutes 31 seconds
GRAPHS AND DATA
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
% Heap Before GC
Aggregate Allocation
Allocation Rates
Recovery Rates
Pause Times
Pause Time
Aggregate Pause Times
% Time in GC
Phases
Parallel Phases
Other Phases
Phases %
Parallel Phases %
Other Phases %
CPU Summary
G1 Concurrent Mark Durations
G1 Statistics
% Parallel Phase Times (%)
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
1,600
1,800
2,000
2,200
2,400
2,600
2,800
3,000
3,200
3,400
3,600
3,800
4,000
4,200
4,400
%
SATB Filtering
Termination
Update RS
Scan RS
Ext Root Scanning
Object Copy
Code Root Scanning

Conclusion

- Easier to tune
- Bound by Object copy
  - Have to pause to evacuate
  - Need concurrent relocation

Shenandoah

Shenandoah

- Compacting
- Concurrently relocates objects
- “Goal is to have $<$ 10ms pause times for 100gb+ heaps.”
- “If you are running with a heap of 20GB or less or if you are running with fewer than eight cores, you are probably better off with one of the current GC algorithms”

Shenandoah

- Roman Kennke - https://rkennke.wordpress.com/
- Christine H. Flood - https://christineflood.wordpress.com/
- http://openjdk.java.net/jeps/189

Shenandoah

- Regional heap similar to G1
- Evacuate areas with high garbage (garbage first)
- Non-generational

Shenandoah Phases

- Mark
- Evacuate

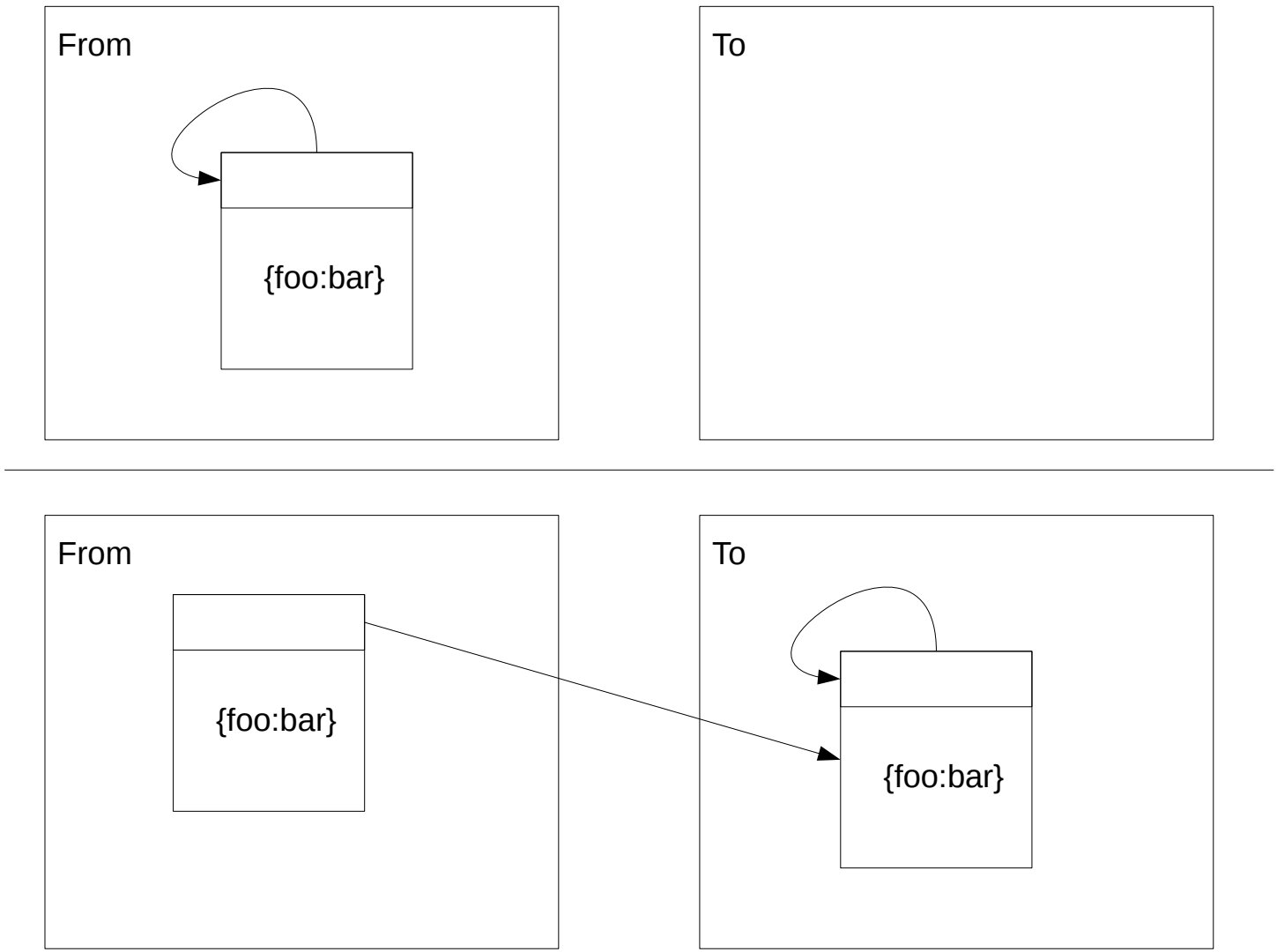
Concurrent Mark

- Snapshot at the beginning
- Full heap

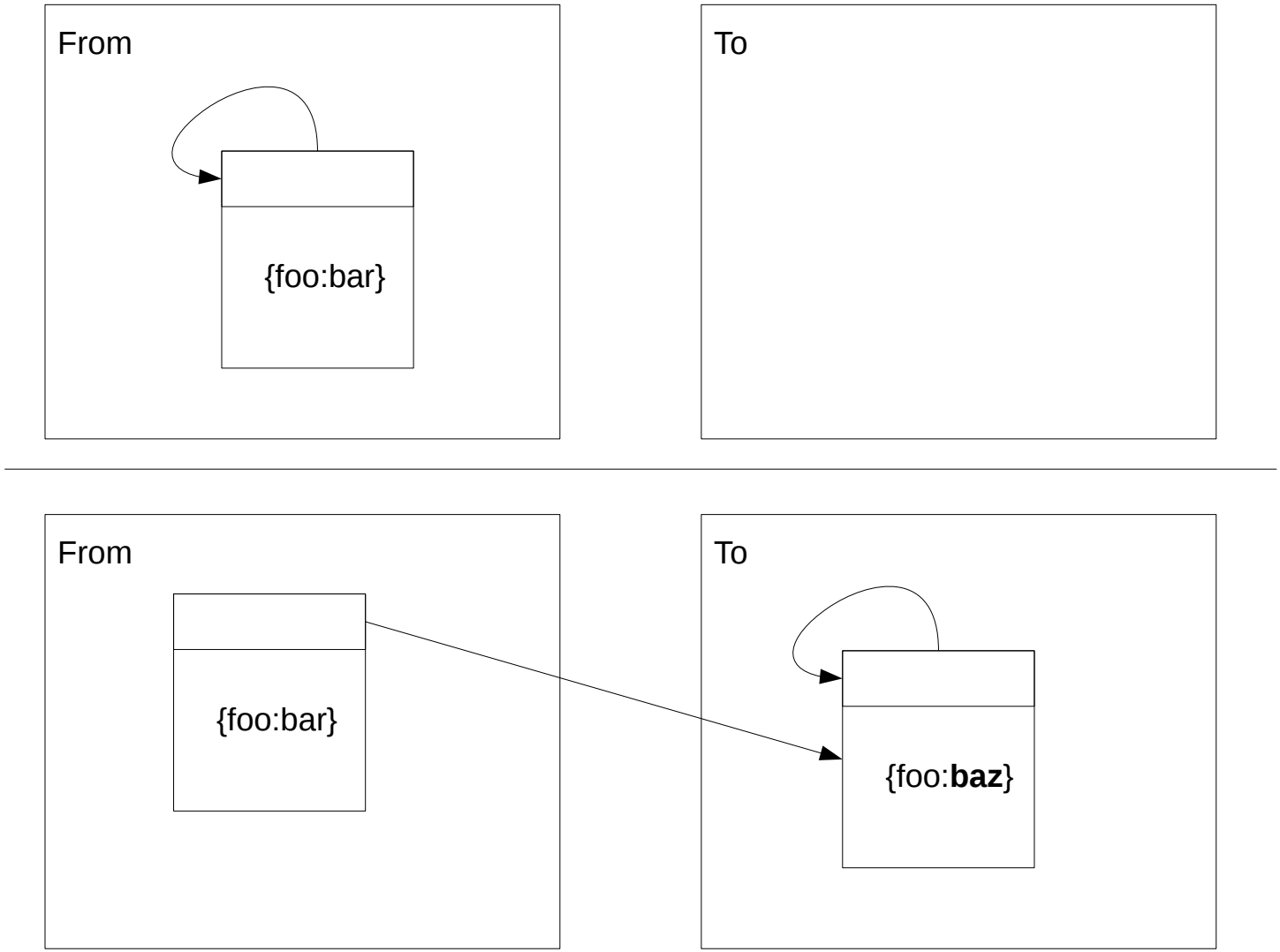
Concurrent Relocation

- Use Brooks pointers
  - All objects have a pointer that normally point to themselves
  - When relocated pointer updated to point to new location
  - Reading and writing threads follow pointer to the “real” object

Brooks Pointers



Brooks Pointers



Brooks Pointers

- Read(Read(pointer))
- Objects evacuated from “From” spaces, to “To” spaces
- Objects relocated on a write
- Pointer in old object updated via CAS operation

Fixing Up Pointers

- 2 Options
  - Fix up pointers in the next mark (default)
    - May require a lot of head room in the heap
    - Pay a cost of indirection on relocated objects between GC's
    - Cache miss?
  - Accept another pause
- No remembered sets

Write Barrier

- Enforce only write in To space
- Perform relocation if needed
  - Avoid read relocation storm
- Ensure previous pointers are still marked

Costs

- Extra heap space for forwarding pointer
- Read/Write barrier
- Pointer chasing may make memory access pattern unpredictable
- Shenandoah devs believe overhead can be kept reasonably low

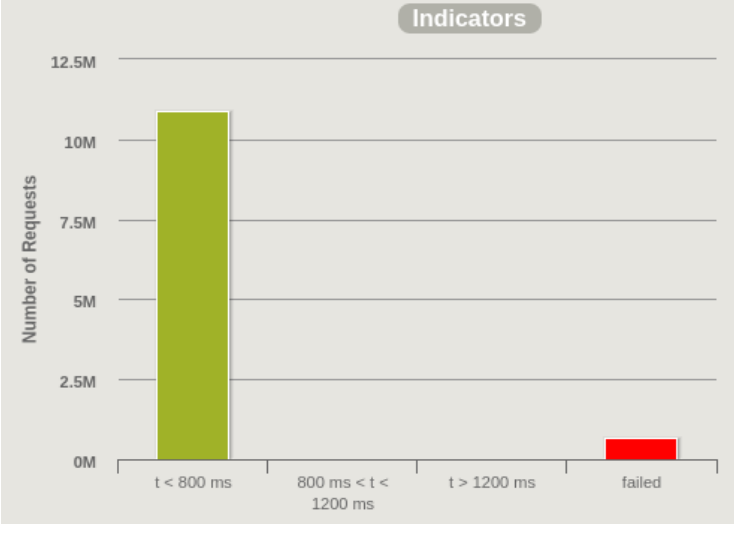
G1 vs Shenandoah

Tests

- Shenandoah
  - **-XX:+UseShenandoahGC**
    -XX:-ClassUnloadingWithConcurrentMark
    -XX:MaxHeapSize=25G
- G1
  - **-XX:+UseG1GC -XX:MaxGCPauseMillis=100**
    -XX:MaxHeapSize=25G
- Environment
  - Spring web app
  - Load test from external machine running Gatling
  - AWS, 8 core, 32GB ram

G1 vs Shenandoah

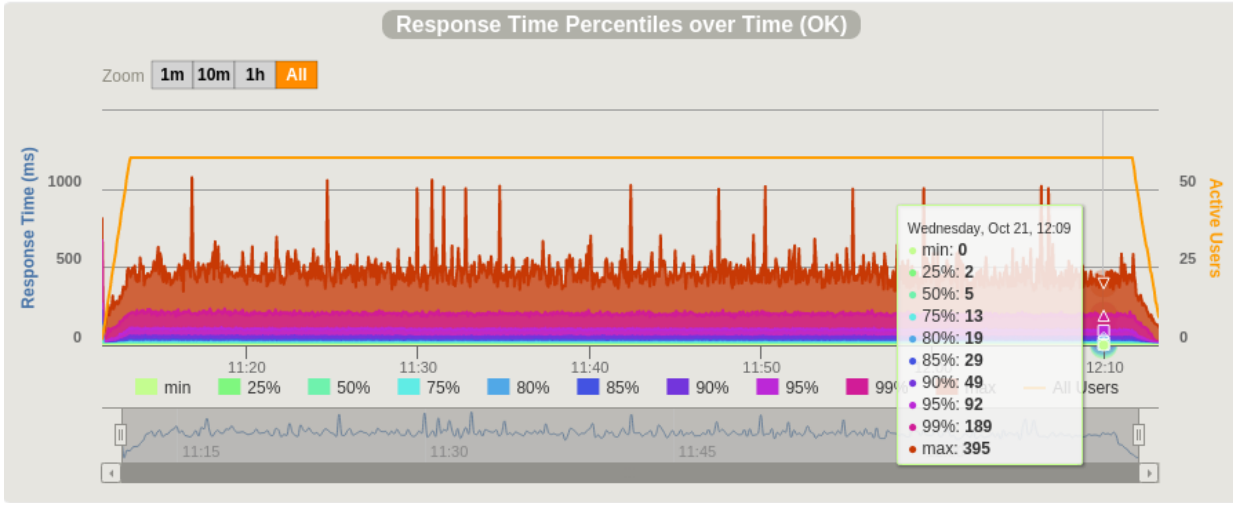
G1



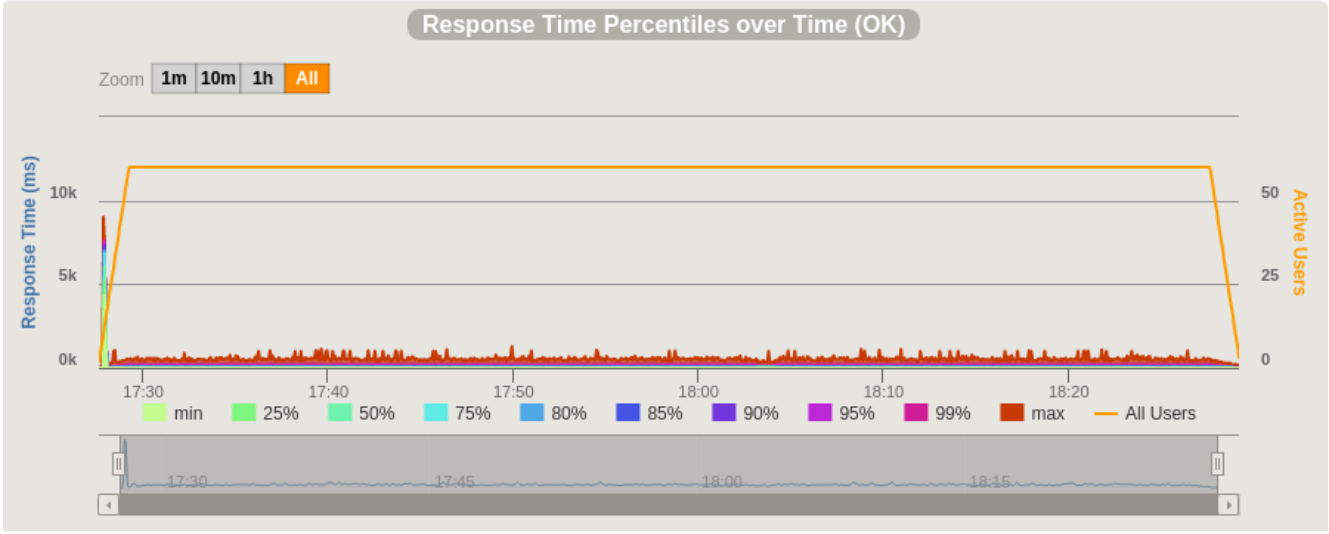
Shenandoah

G1 vs Shenandoah - Response Times

G1

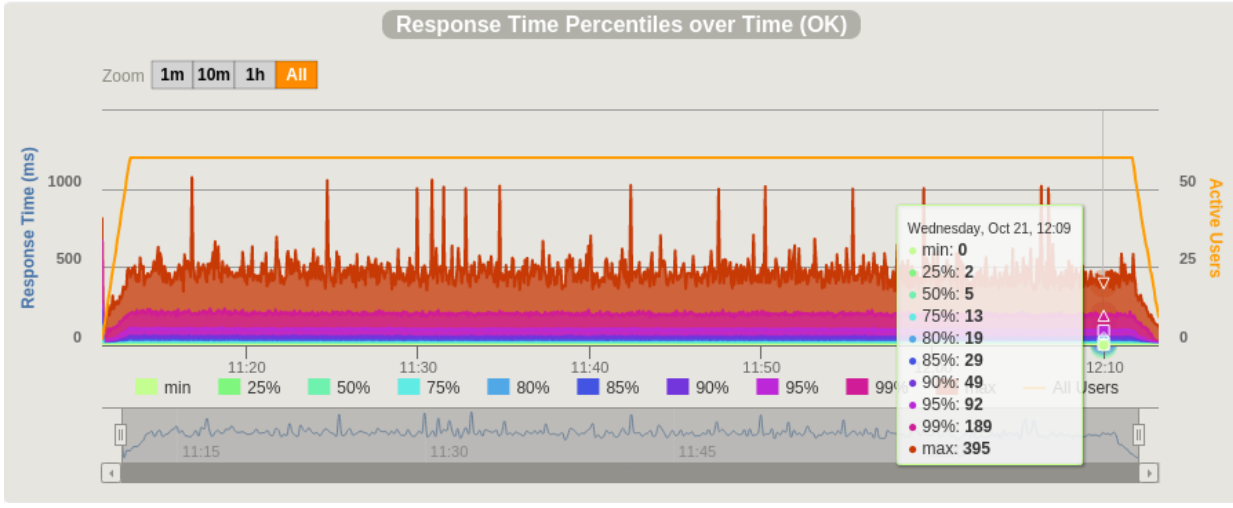


Shenandoah

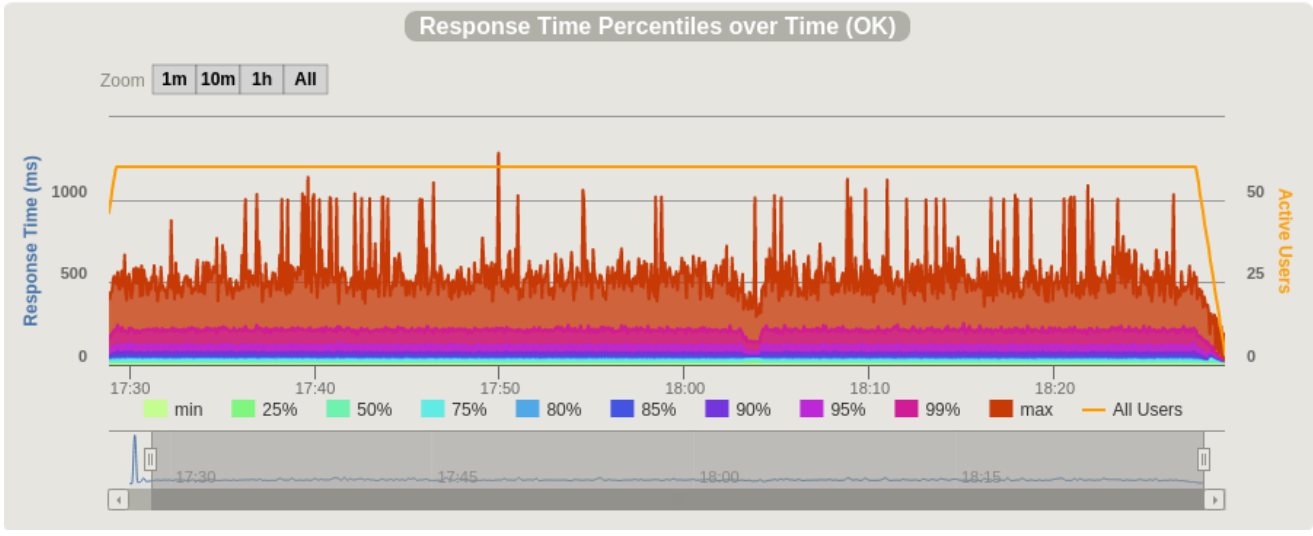


G1 vs Shenandoah- Response Times (No Full GC)

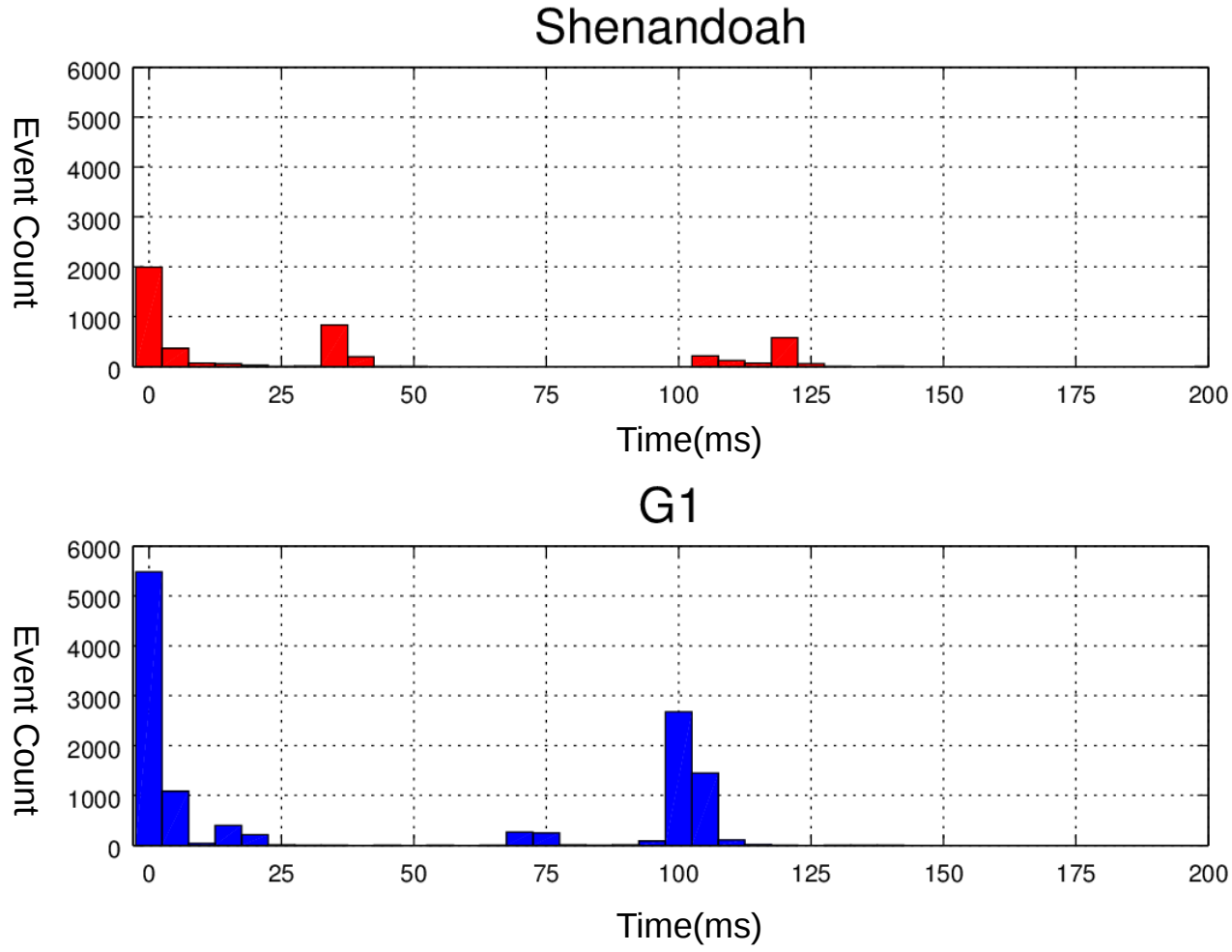
G1



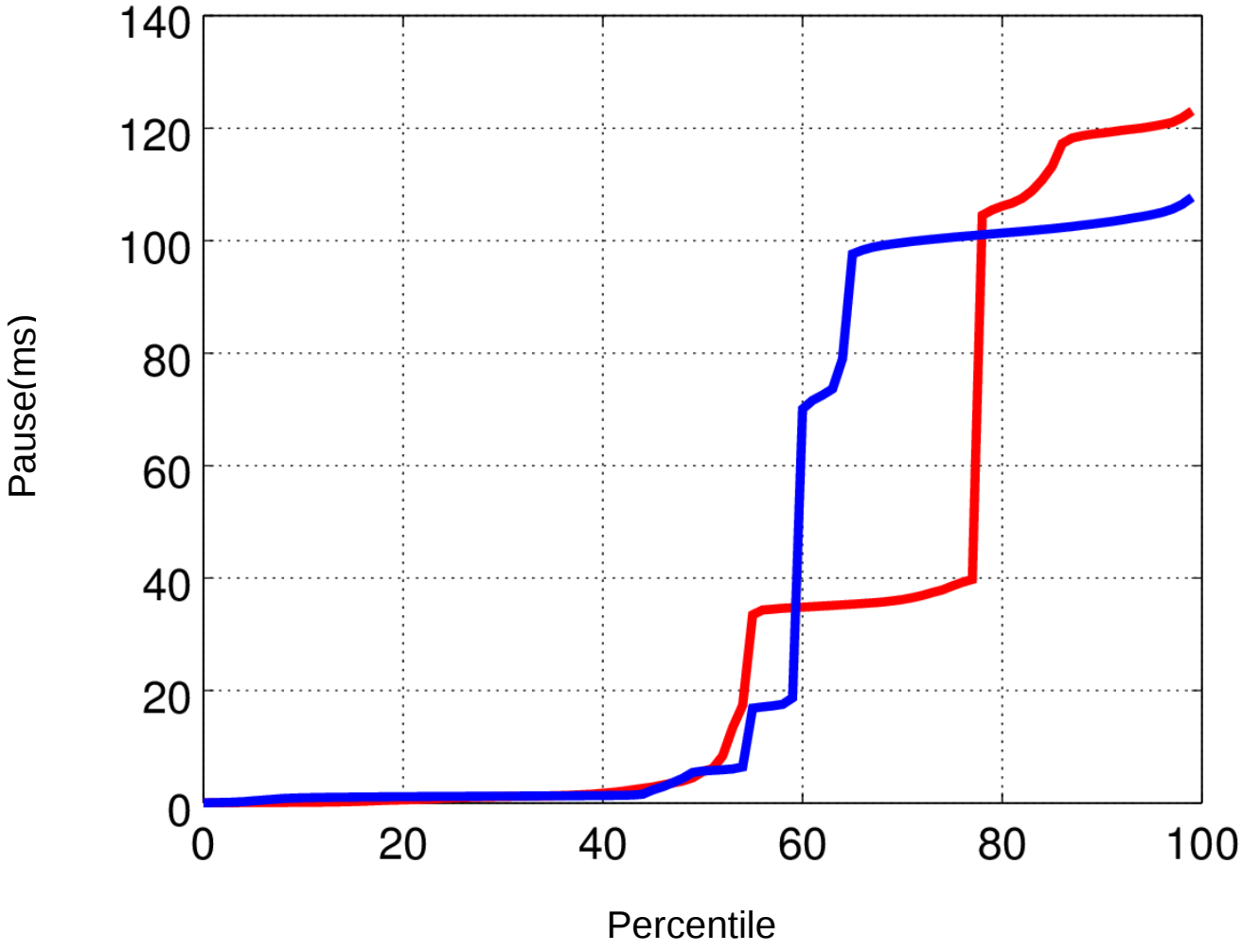
Shenandoah



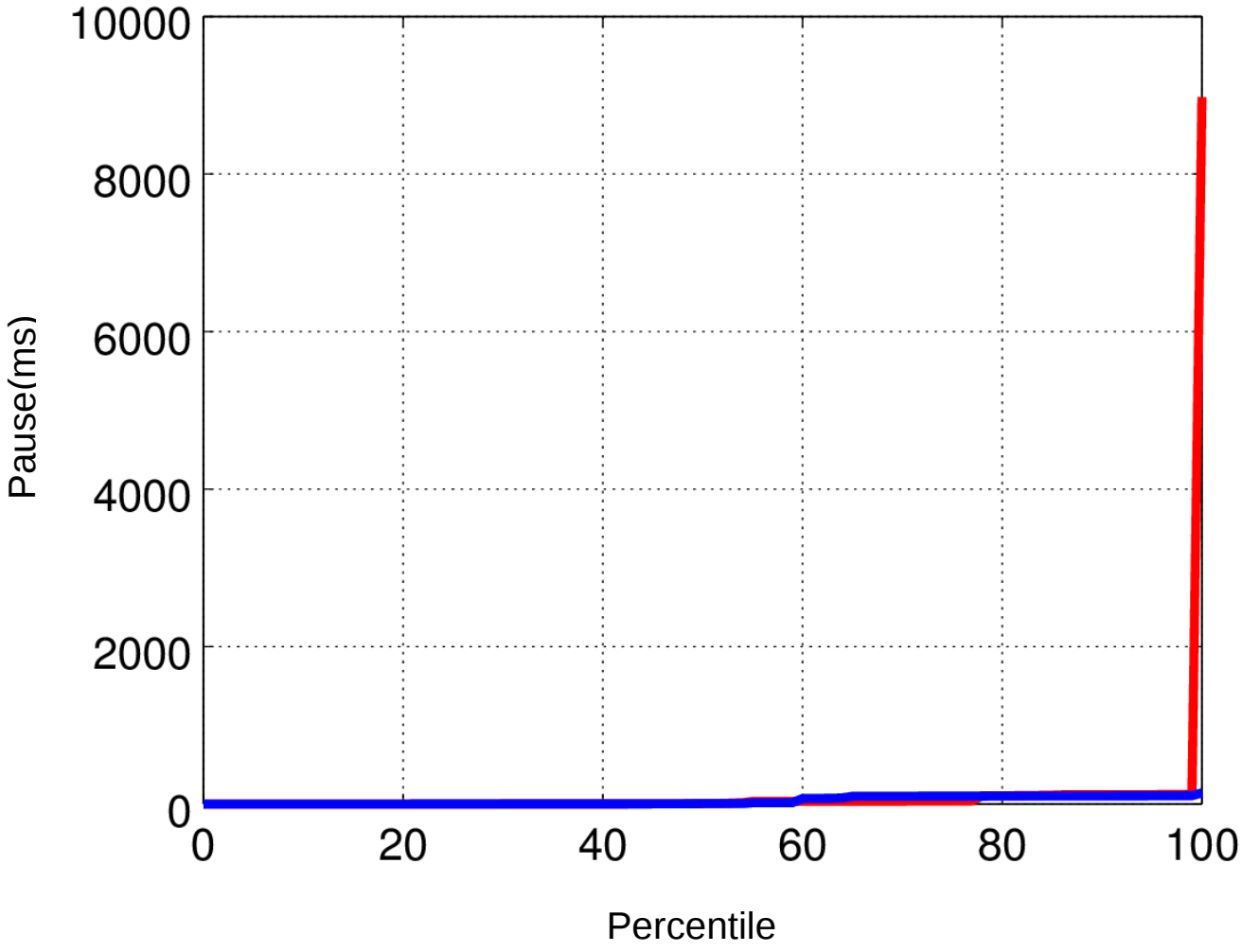
G1 vs Shenandoah - GC event counts



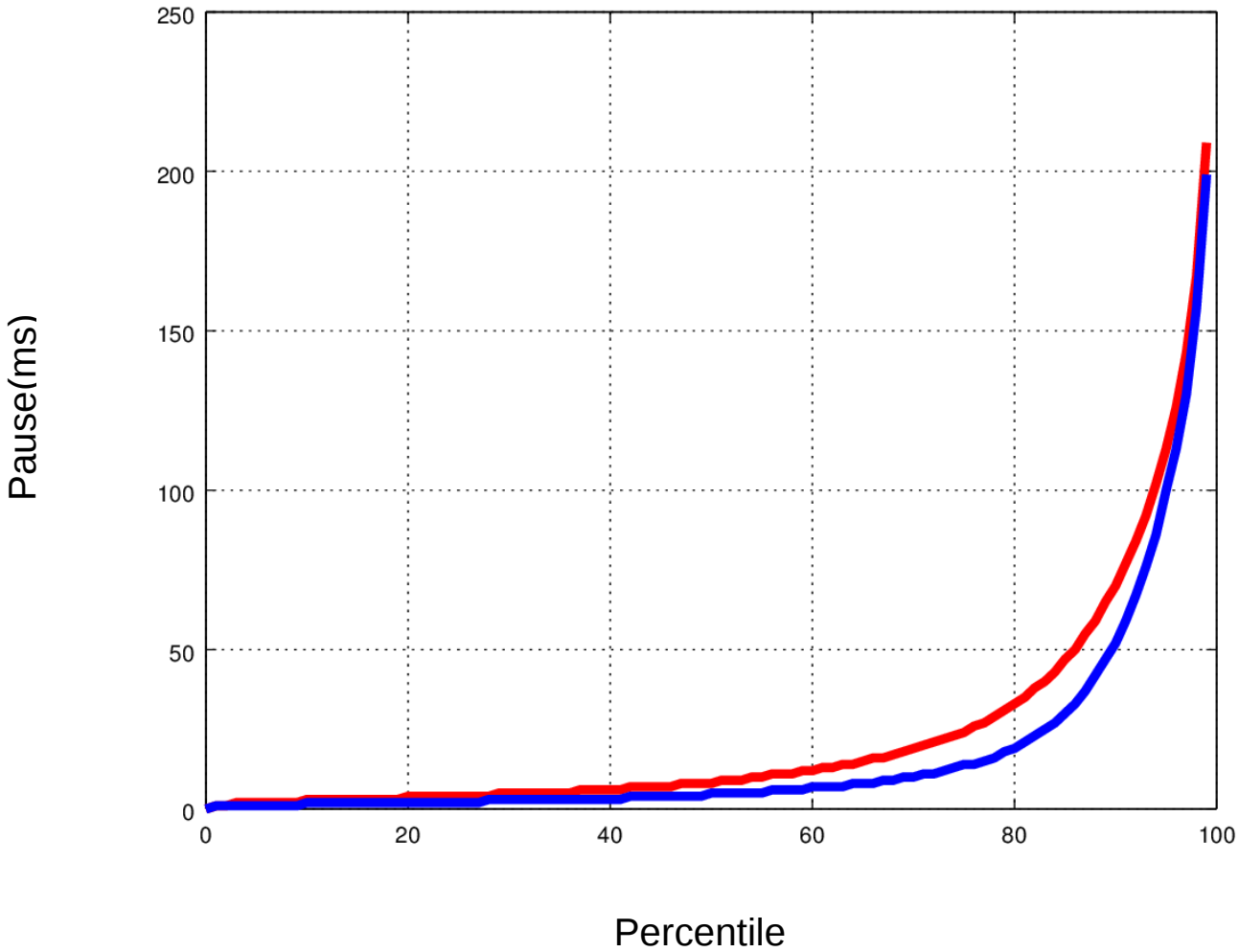
G1 vs Shenandoah Pause Times, 0-99th Percentile



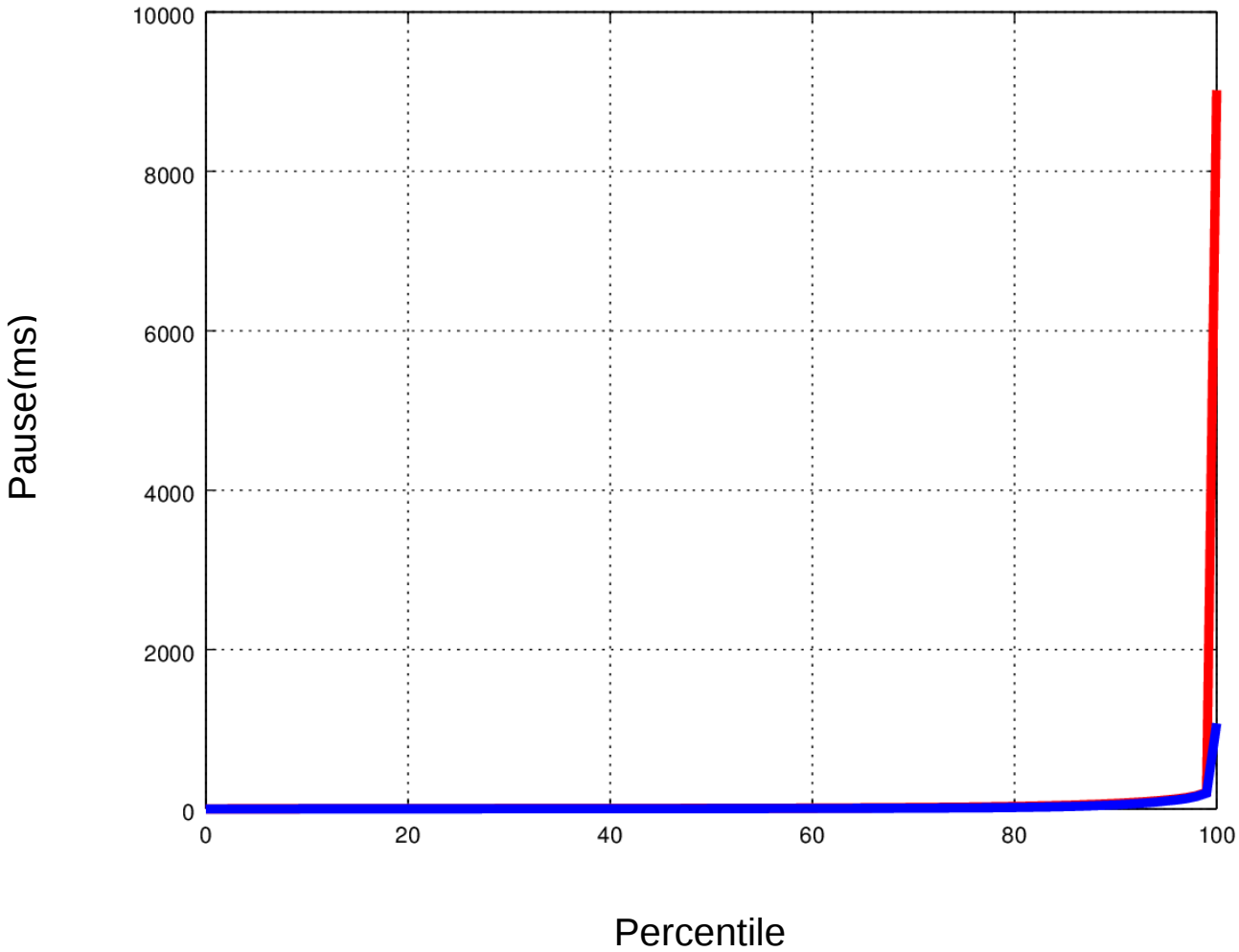
G1 vs Shenandoah Pause Times, 0-100th Percentile



G1 vs Shenandoah Response Times, 0-99th Percentile



G1 vs Shenandoah Response Times, 0-100th Percentile



Results

- Still very early days for Shenandoah
  - Far younger than G1 and far less engineering time
  - Has comparable results to G1
- Beware of single benchmarks and results
  - Application stopped time
- STOP THE PRESSES
  - Since performing this talk, following advice from Roman Kennke I have managed to produce stable results with <60ms pauses at the 99th percentile and no full GCs.

Conclusion

Conclusion

- G1
  - Production ready
  - Bound by object copy
- Shenandoah
  - Looks very promising but not ready yet
  - Will remove the bound on object copy
  - Java 10
- Join Friends of jClarity - friends@jclarity.com
- Send in your GC logs

Questions

- john@jclarity.com
- Friends of jClarity - friends@jclarity.com