Adding Generational Support to Shenandoah GC

Kelvin Nilsen, Senior Development Engineer, Amazon Web Services
(work performed in collaboration with multiple members of Corretto JVM Team)

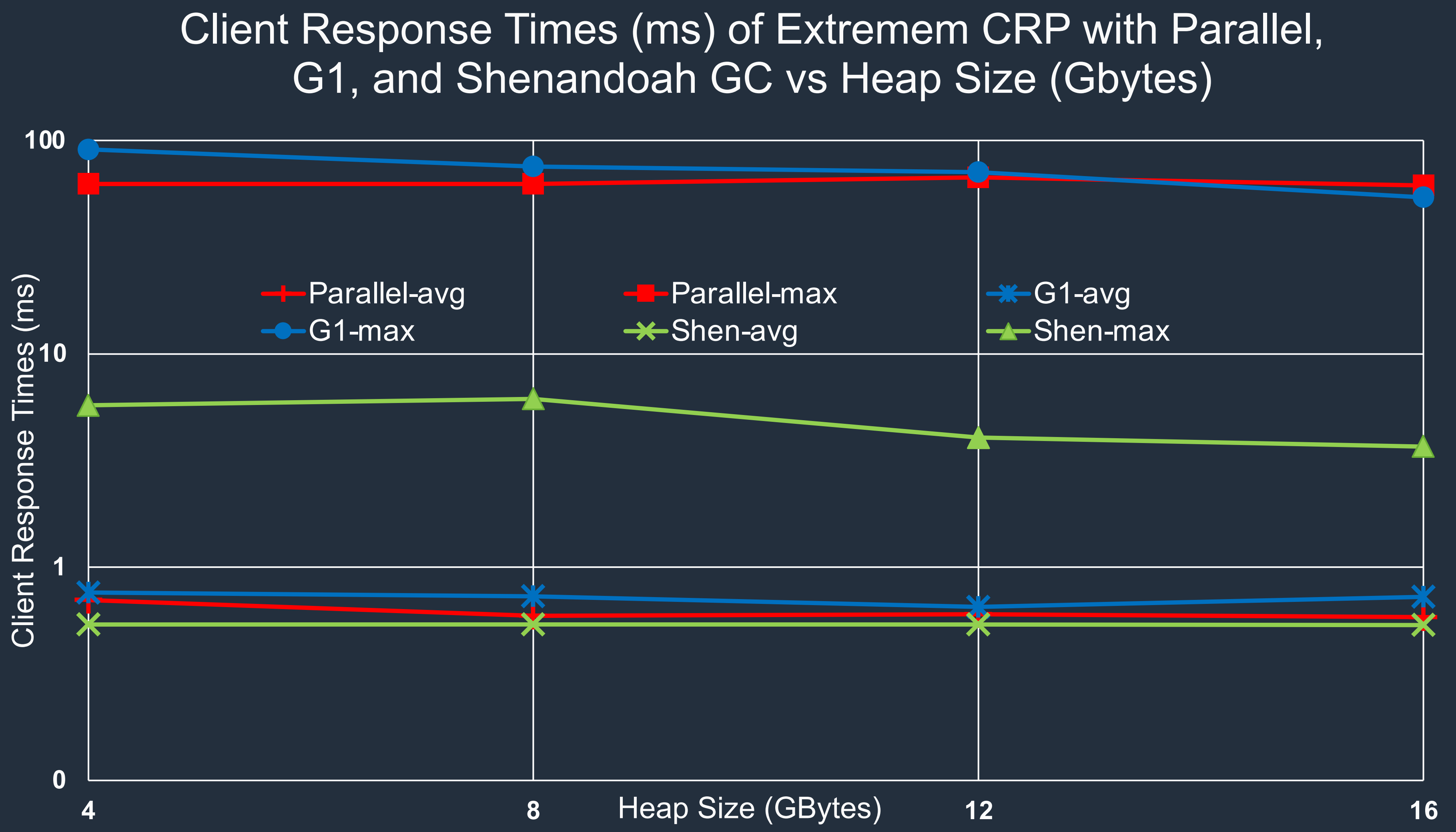
Agenda

- Shenandoah overview and strengths
- Shenandoah limitations addressed by generational enhancements
- Architecture and Design of Generational Shenandoah
- Demonstration
- Project Status and Directions

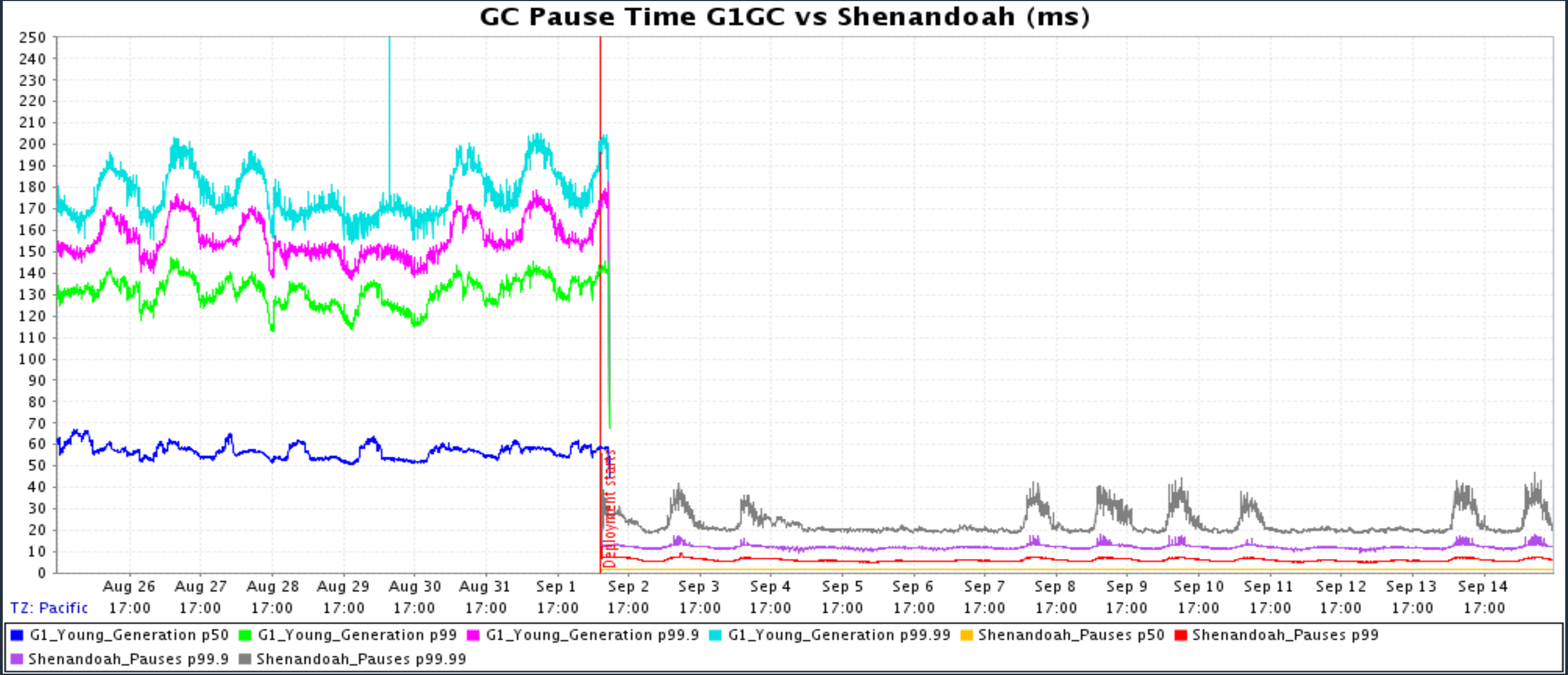
Shenandoah GC

- Targets timeliness needs of 10 ms or below
- Typical pauses are around 5 ms, separated by concurrent execution phases of 50-500 ms
  - Typical pause times decrease to 1 ms or below with concurrent root scanning, concurrent reference processing, concurrent string deduplication in OpenJDK 17
- Trade offs vs. less concurrent GC approaches
  - Requires larger heaps (works best with heap utilization below 30%)
    - Lower allocation rates can run with higher heap utilization
  - May experience throughput penalties (e.g. 10-15% vs. Parallel GC)

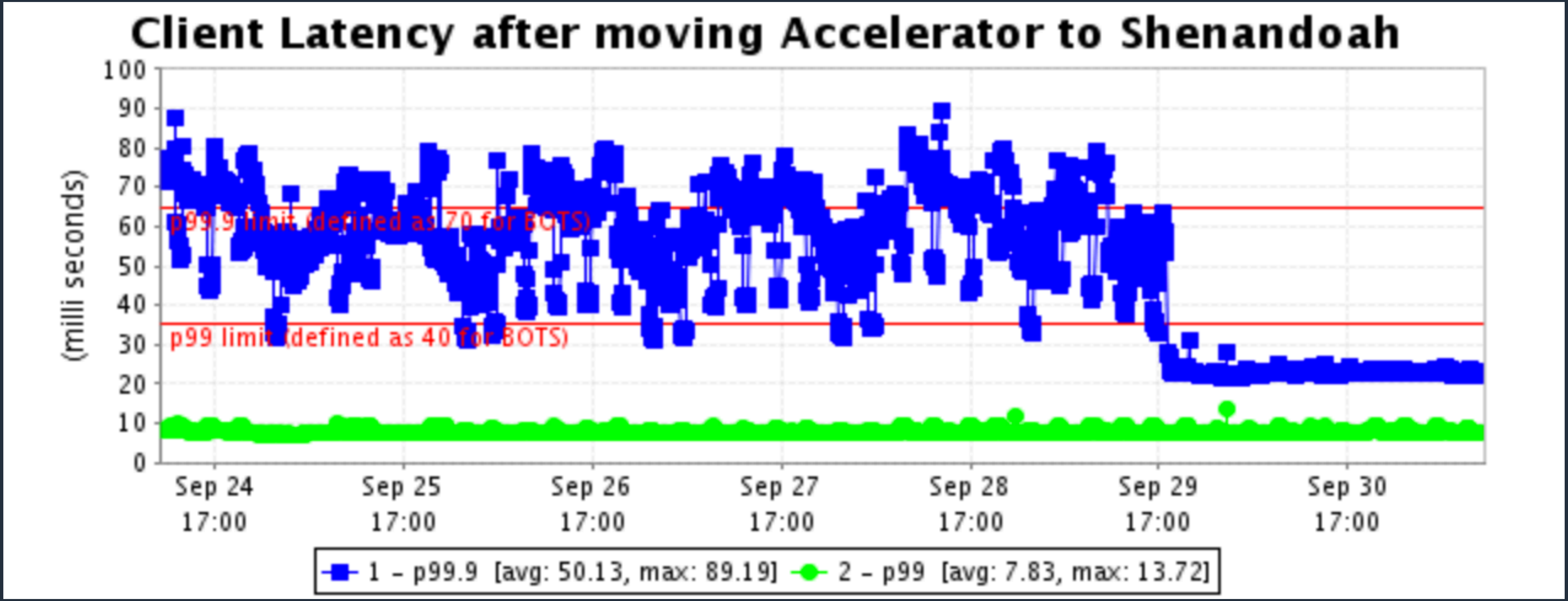
Comparison of Shenandoah GC vs G1 and Parallel GC (OpenJDK 11)



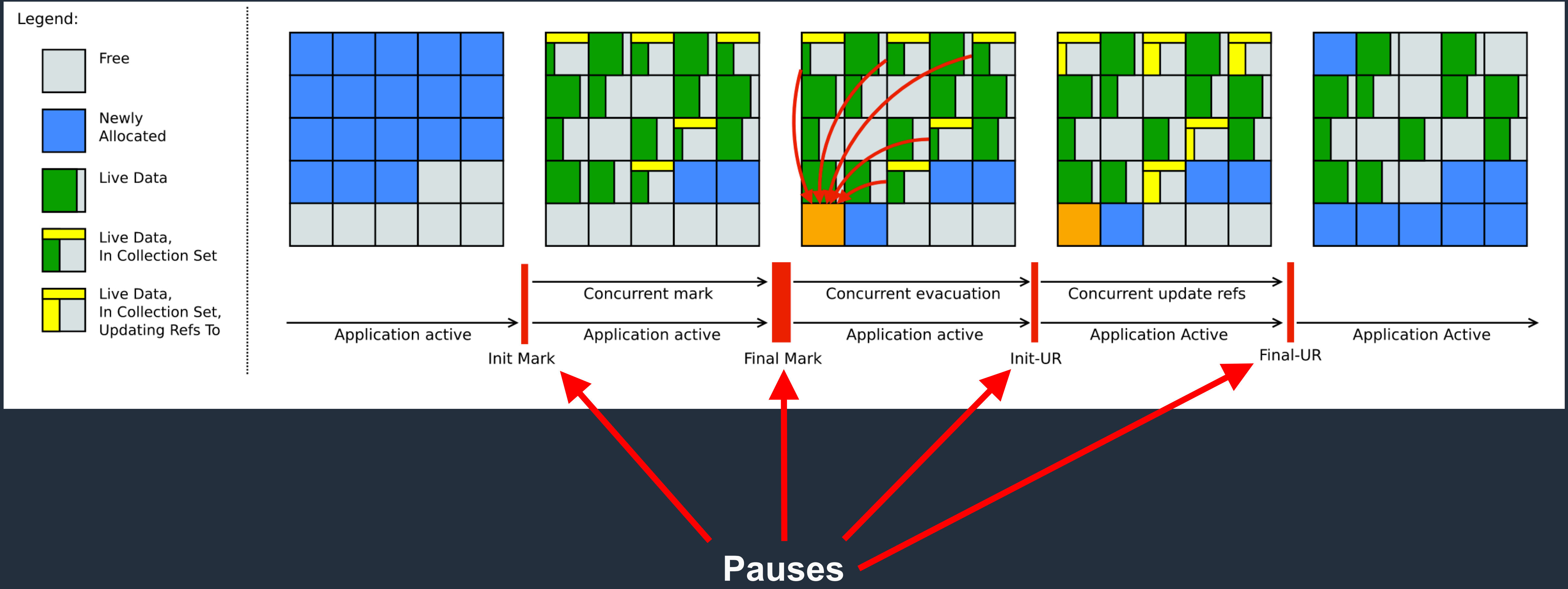
One AWS Service switched to OpenJDK 11 Shenandoah: 9/2/2020



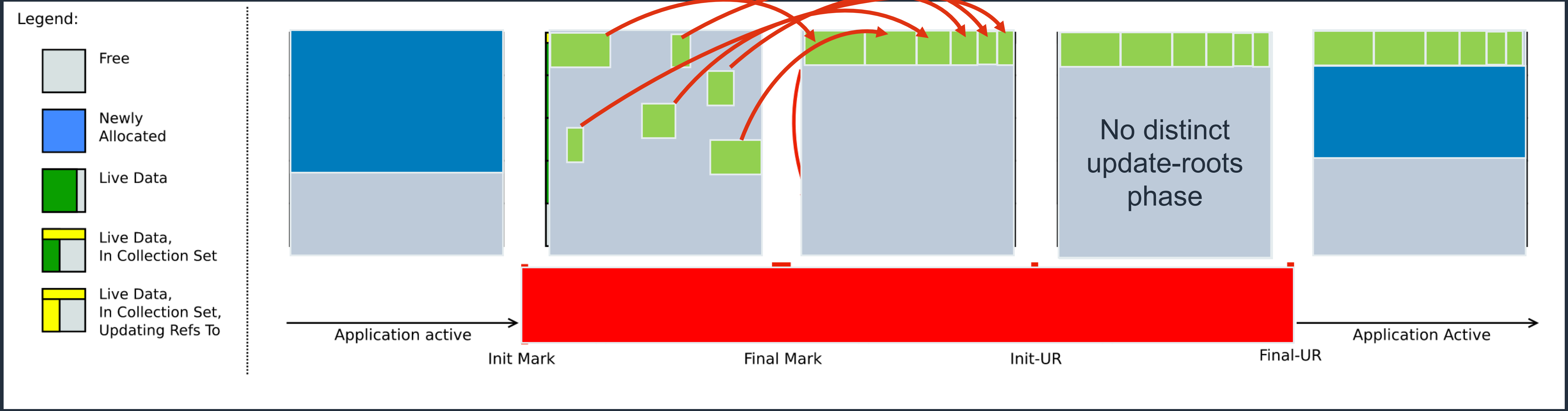
Another AWS service switched to OpenJDK 11 Shenandoah: 9/29/2020



Shenandoah Overview

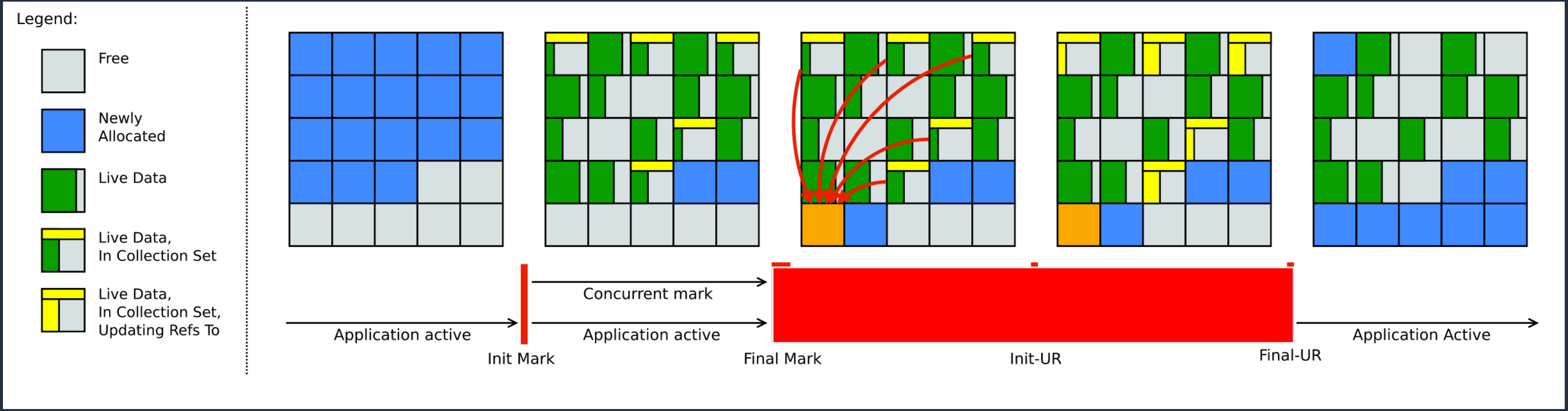


Parallel GC Comparison



- Application is stalled throughout marking, evacuation and updating, since these phases are not "concurrent"
- But application throughput is higher overall, because no "coordination overhead" is required between concurrent application and GC threads

G1 GC Comparison



- Global marking is concurrent, but application is stalled during evacuation and updating of references (young-gen collections are stop-the-world)
- G1 coordination overheads are greater than parallel GC, but lower than Shenandoah
- G1 places “soft bounds” on pause times by limiting the number of heap regions that are collected during each evacuation pass

Shenandoah GC

- Targets timeliness needs of 10 ms or below
- Typical pauses are around 5 ms, separated by concurrent execution phases of 50-500 ms
    - Typical pause times decrease to 1 ms or below with concurrent root scanning, concurrent reference processing, concurrent string deduplication in OpenJDK 17

- Trade offs vs. less concurrent GC approaches
    - Requires larger heaps (works best with heap utilization below 30%)
        - Lower allocation rates can run with higher heap utilization
    - May experience throughput penalties (e.g. 10-15% vs. Parallel GC)

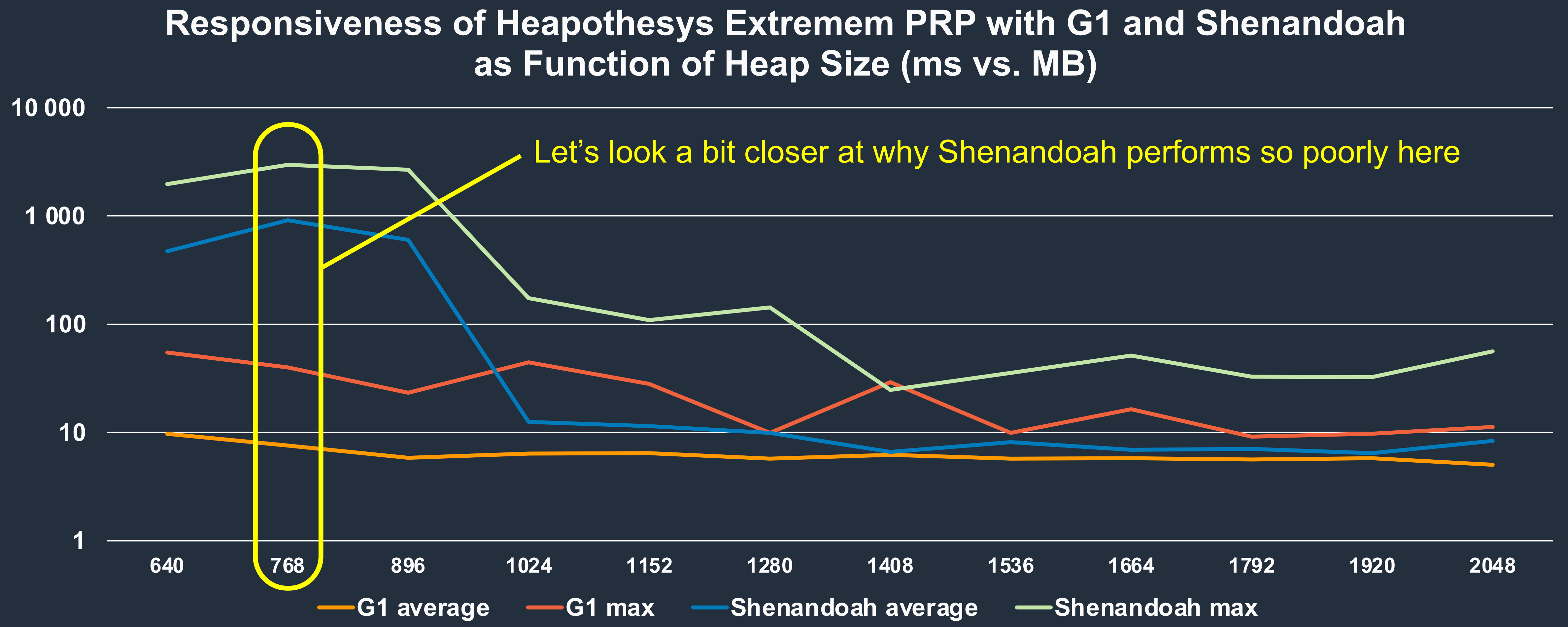
Shenandoah GC with Generational Enhancements

- Targets timeliness needs of 10 ms or below
- Typical pauses are around 5 ms, separated by concurrent execution phases of 50-500 ms
  - Typical pause times decrease to 1 ms or below with concurrent root scanning, concurrent reference processing, concurrent string deduplication in OpenJDK 17
- Trade offs vs. less concurrent GC approaches
  - ~~Requires larger heaps (works best with heap utilization below 30%)~~
    - ~~Lower allocation rates can run with higher heap utilization~~
  - May experience throughput penalties (e.g. 10-15% vs. Parallel GC)
  - The young-gen memory area has generally very low memory utilization, so can sustain very high allocation rates
  - The old-gen memory area may have high memory utilization, but the allocation rate within old-gen is very low, so old-gen concurrent GC can run infrequently at low priority

More Efficient Utilization of Heap Memory

- Whereas non-generational Shenandoah requires heap utilization below ~30% for high allocation rates, Generational Shenandoah will support the same pause time bounds as Shenandoah while:
  - Sustaining 4-16 times higher allocation rates
  - Supporting old-gen utilization of 90% or higher, with
  - Much smaller young-gen memory regions at utilization of 15-30%
  - Enabling combined memory utilization of 75% or higher

Comparing Shenandoah vs G1 on Strenuous Workload

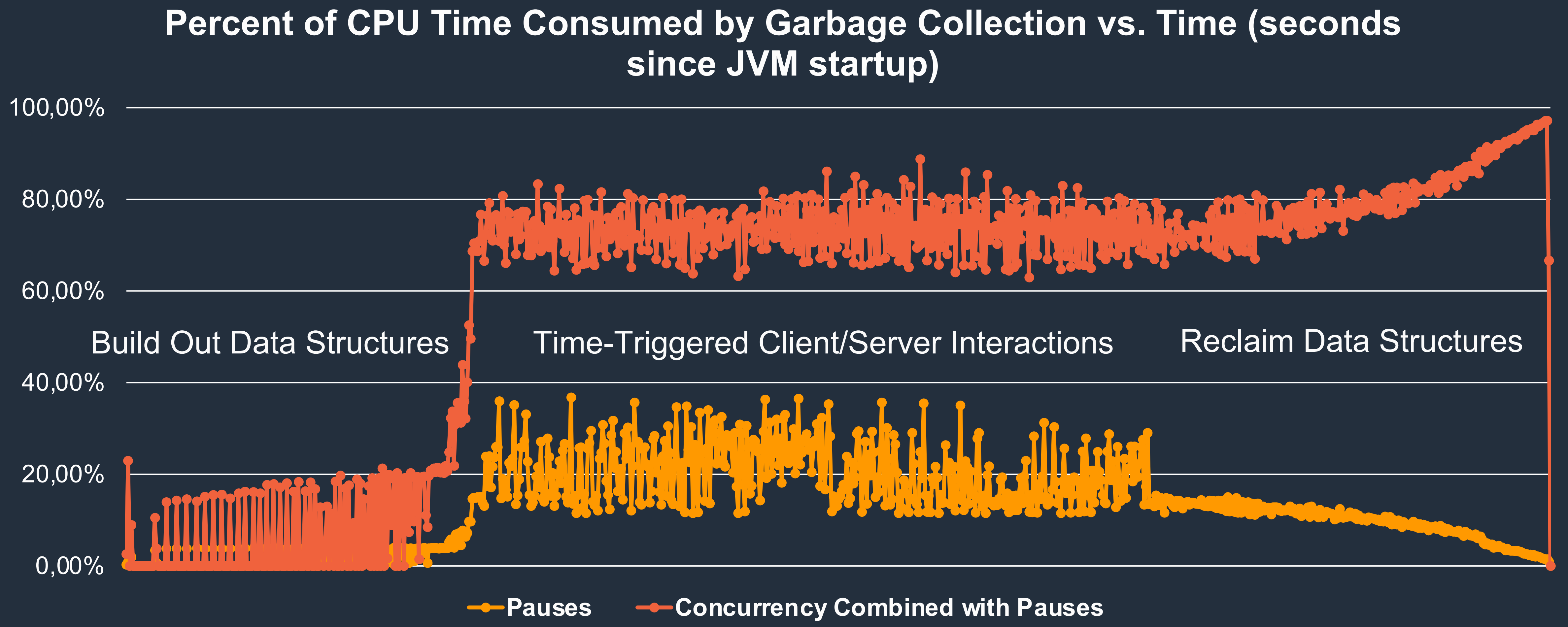


Degenerated GC with Shenandoah

- Each concurrent Shenandoah GC pass is a race between the application and the garbage collector
  - GC racing to replenish free pool
  - Application racing to consume memory

- Shenandoah tries to be smart about deciding when to start GC
  - If it starts too soon, it wastes efforts doing more frequent GC than necessary
  - If it starts too late, the application wins the race. When allocation pool is exhausted, Shenandoah does “degenerated” stop-the-world GC
  - Corretto JVM engineer William Kemper recently implemented improvements to make the choice of when to start GC more effective

With 768 MB Heap, Shenandoah performed 381 degenerated GC out of 3,763 total GC passes during 10 minute execution run



Experimental Observations of What Shenandoah Does Well

- Concurrent execution of Shenandoah allows applications to run without “GC pauses” as long as GC is able to win each race to replenish allocation pool

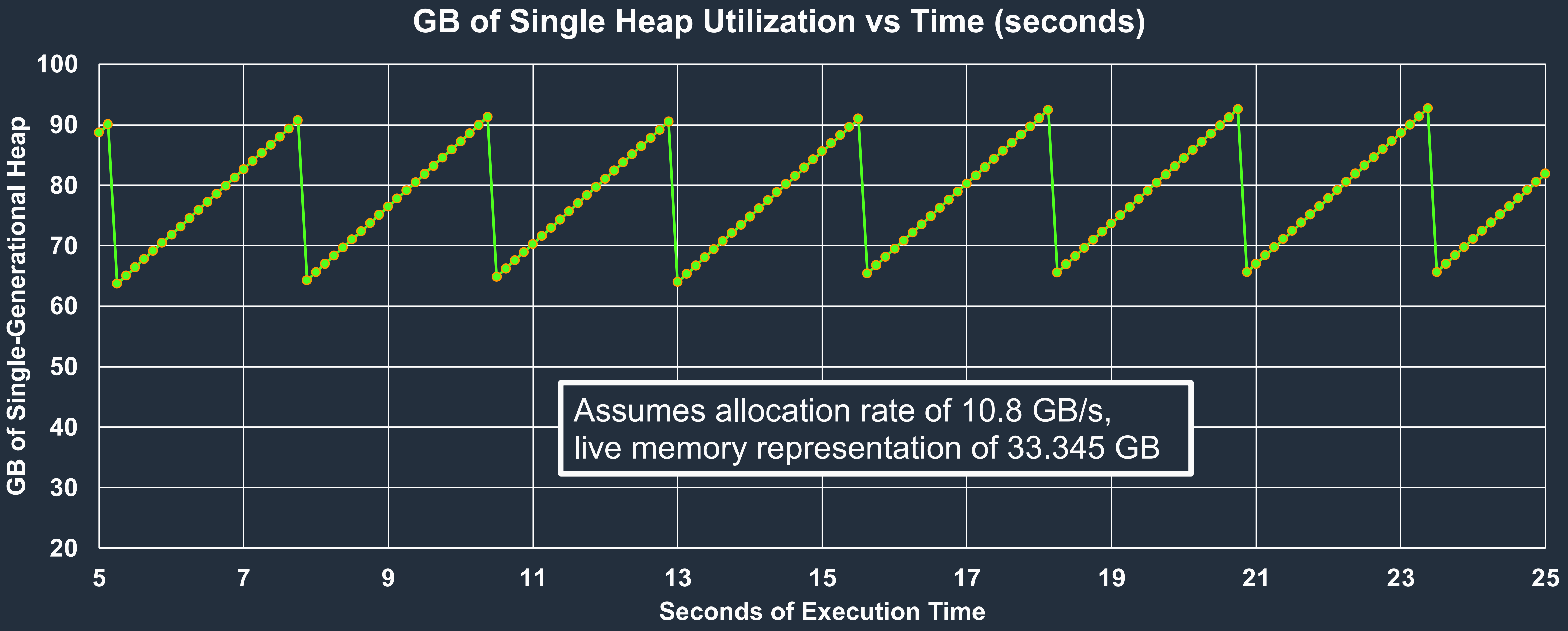
  - Low allocation rates (e.g. 512 MB/s or lower):
    pauses bounded below 10 ms even at high (e.g. 90%) memory utilization

  - High allocation rates (4 GB/s or higher):
    limit utilizations to 30% or below to maintain pause times below 10 ms

Data-Driven Design for Generational Shenandoah

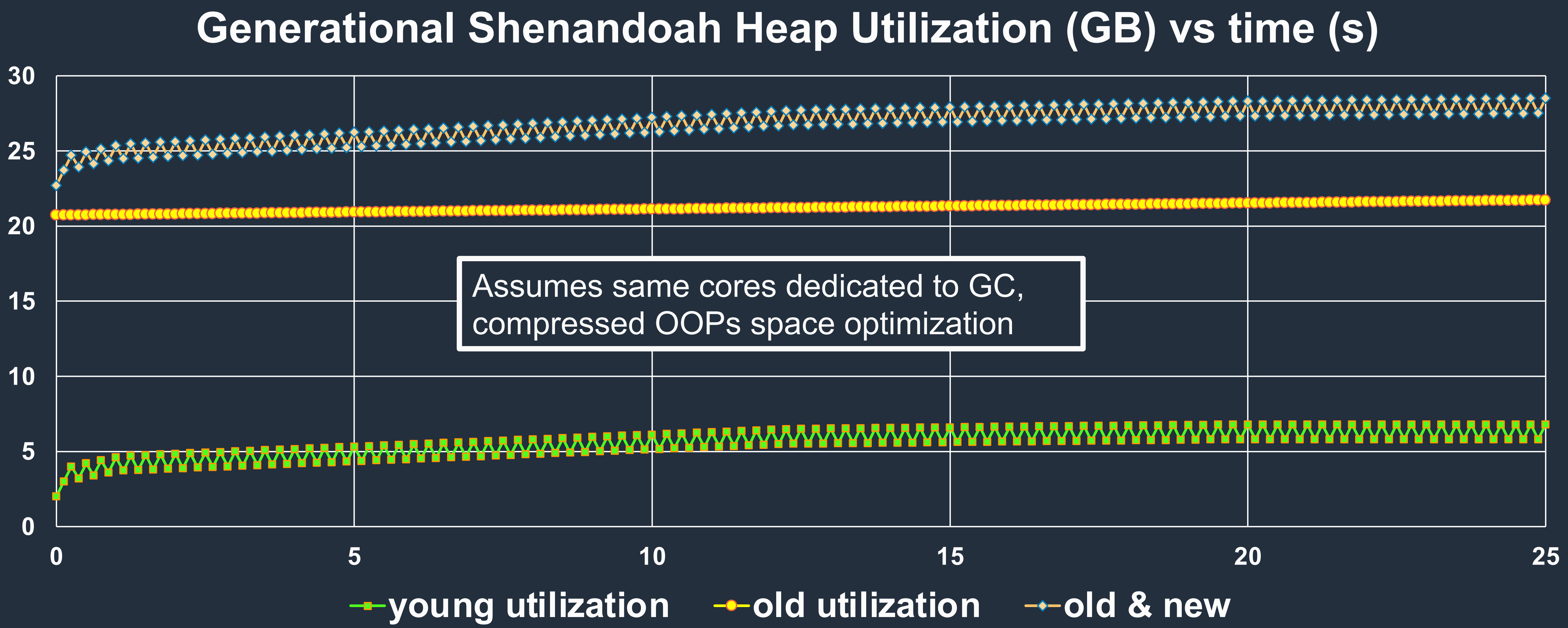
- Use two “instances” of the Shenandoah GC collector to concurrently collect young-gen and old-gen memory
- When “most objects die young”:
  - Young-gen memory has very high allocation rate and very low memory utilization
  - Old-gen memory has very low allocation rate and relatively high memory utilization
- Young-gen collector runs very frequently at a high priority
- Old-gen collector runs rarely at a low priority
  - Multiple young-gen collections run to completion while old-gen collector is working
  - Between young-gen collections, all GC threads can focus attention on old-gen efforts
  - While young-gen collection is active, very limited efforts dedicated to old-gen GC

Comparison between Non-Generational and Generational GC on Simulated Workload



Assumes allocation rate of 10.8 GB/s, live memory representation of 33.345 GB

Generational GC Shrinks Heap Requirement by Two Thirds



Assumes same cores dedicated to GC, compressed OOPs space optimization

Open Source Coordination

- JEP 440: Generational Shenandoah, authored by Bernd Mathiske, Kelvin Nilsen, William Kemper, is under review (https://openjdk.java.net/jeps/404)

- Source is maintained at https://github.com/openjdk/shenandoah

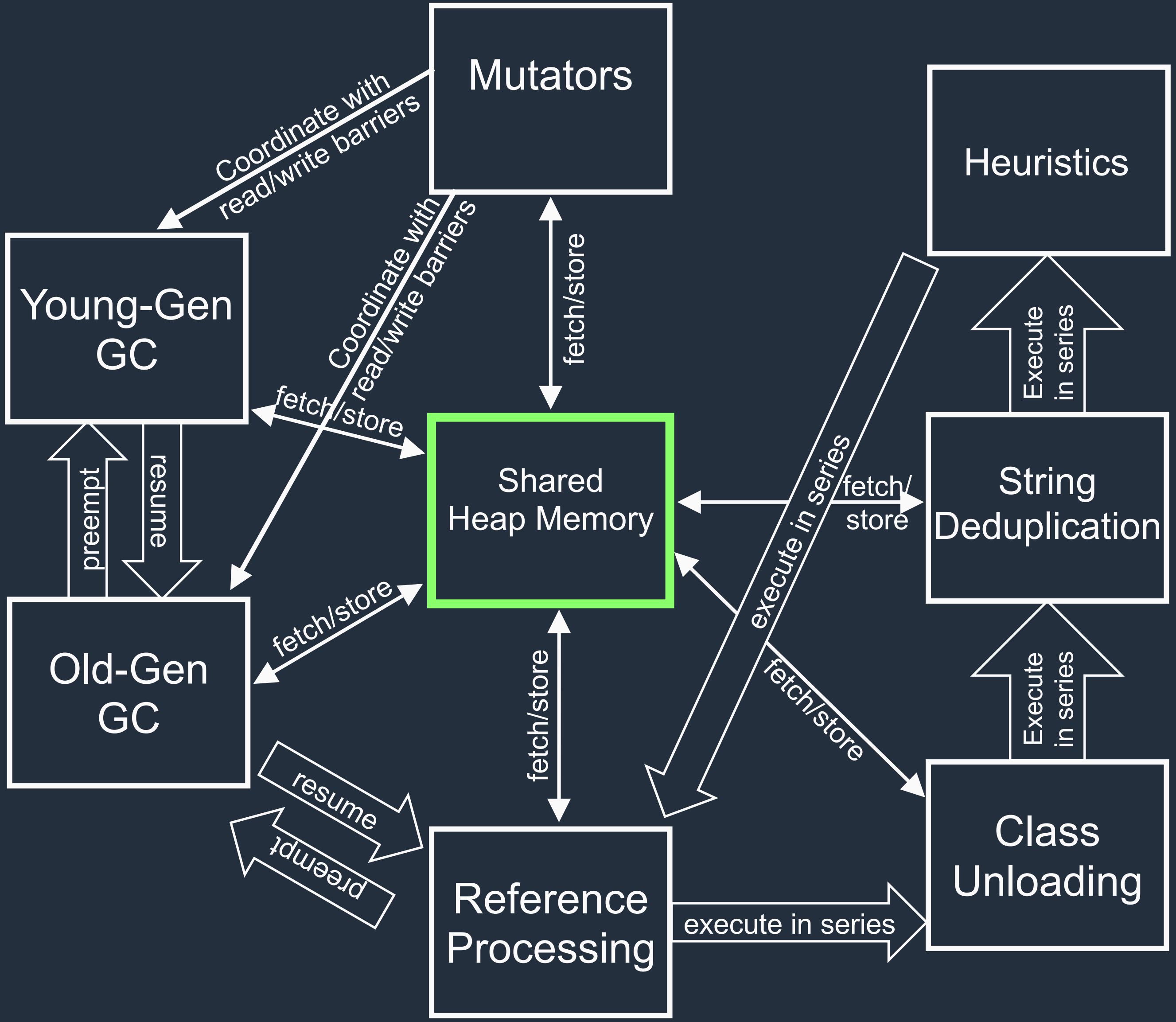
Generational Shenandoah Design Tenets

- Don’t punish the mutator
- Prioritize young-gen GC efforts to sustain allocation rates
- Focus on single-socket deployments before NUMA
- Leverage open-source community
- Demonstrate incremental progress with manageable milestones
- Align milestones with product development progress
- Support production deployments with staged feature releases

Key Implementation Details

- Young-gen GC runs at high frequency, old-gen remembered set included in root set
  - Remembered set scanning is concurrent
- Old-gen GC begins by sharing a concurrent root scan with young-gen GC
- While either young-gen or old-gen GC is concurrently marking, the SATB barrier is enabled
  - If young-gen concurrent marking is enabled, each young-gen object referenced from within a SATB buffer is marked
  - If old-gen concurrent marking is enabled, each old-gen object referenced from within a SATB buffer is marked
- If young-gen GC wants to run while old-gen GC is active, the entire gang of worker threads suspends efforts on old-gen and focuses on young-gen GC
- After old-gen GC completes concurrent marking:
  - Old-gen GC sets aside certain old-gen regions as candidates for future collection sets
  - During each subsequent young-gen evacuation pass, a subset of old-gen candidates is folded into collection set

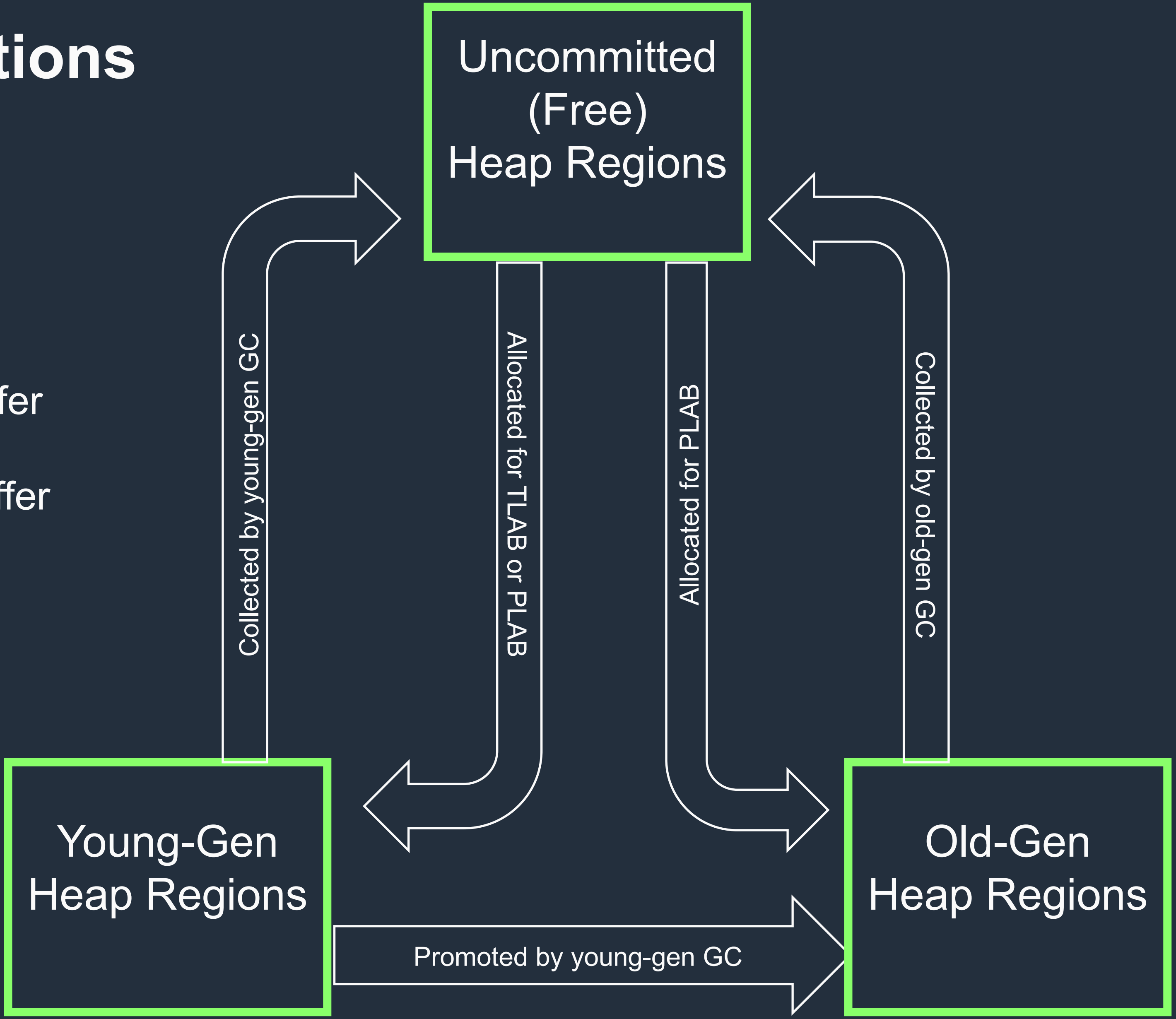
Architecture Overview



Heap-Region State Transitions

TLAB: Thread-Local Allocation Buffer

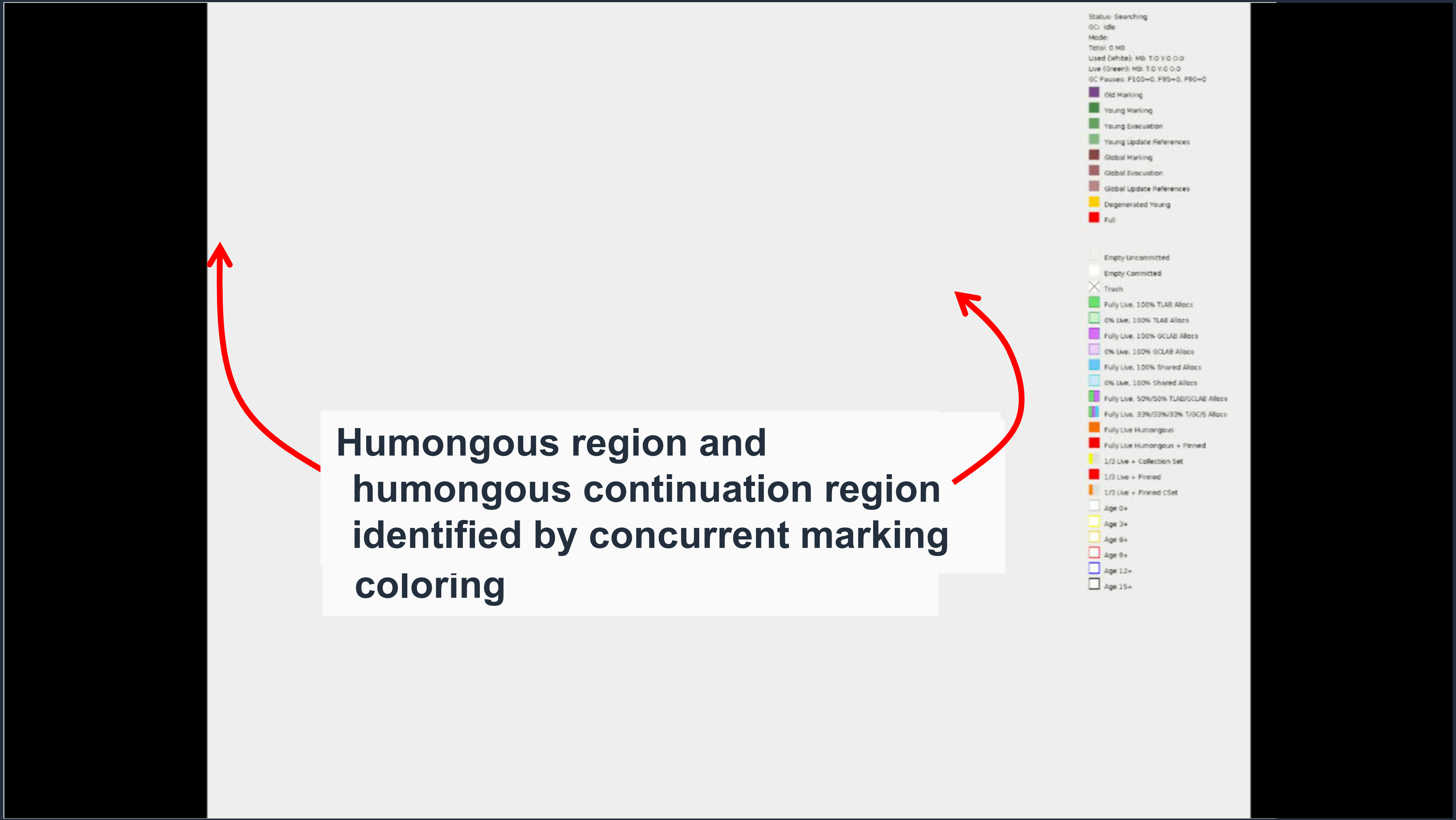
PLAB: Parallel Local Allocation Buffer
(Used for GC evacuations
and promotions)



Our Accomplishments So Far

- Milestone 1: Pure young collection
- Milestone 2: Size-restricted young gen with scheduling heuristics
- Milestone 3: Tenuring and promotion
- Milestone 4: Per-region heap status visualization
- Milestone 5: Global collection after young collection
- Milestone 6: Young collection after global collection, repeat alternations
- Milestone 7: Concurrent old marking
- Milestone 8: Concurrent old and young collections

Visualization of Concurrent Old- and Young-Gen GC

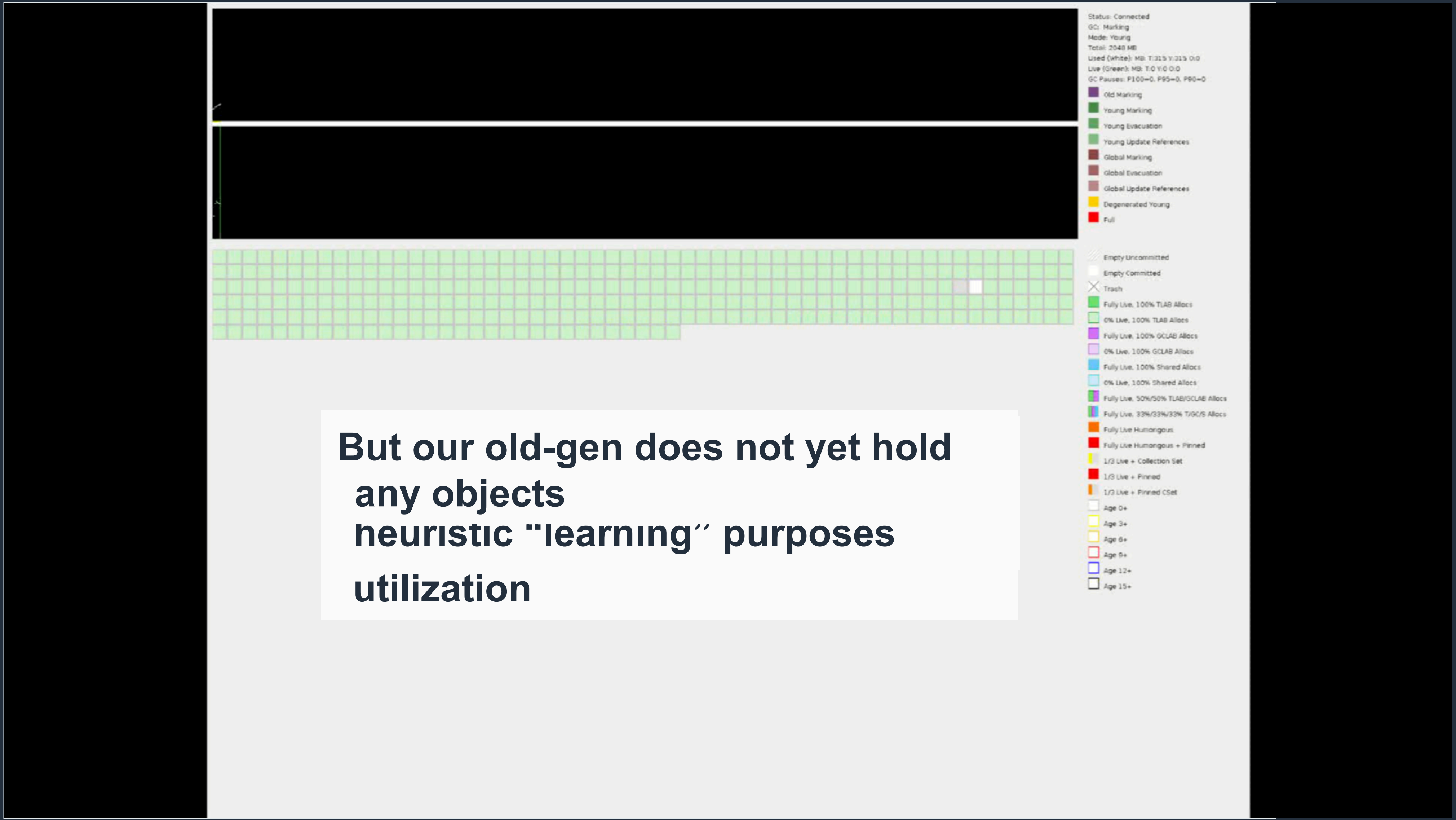


```
Status: Searching
GC: Idle
Mode:
Total: 0 MB
Used (White): MB: T:0 Y:0 O:0
Live (Green): MB: T:0 Y:0 O:0
GC Pauses: P100=0, P95=0, P90=0
```

- 🟪 Old Marking
- 🟩 Young Marking
- 🟩 Young Evacuation
- 🟩 Young Update References
- 🟥 Global Marking
- 🟥 Global Evacuation
- 🟥 Global Update References
- 🟨 Degenerated Young
- 🟥 Full
- ▨ Empty Uncommitted
- ⬜ Empty Committed
- ✕ Trash
- 🟩 Fully Live, 100% TLAB Allocs
- 🟩 0% Live, 100% TLAB Allocs
- 🟪 Fully Live, 100% GCLAB Allocs
- 🟪 0% Live, 100% GCLAB Allocs
- 🟦 Fully Live, 100% Shared Allocs
- 🟦 0% Live, 100% Shared Allocs
- 🟩 Fully Live, 50%/50% TLAB/GCLAB Allocs
- 🟩 Fully Live, 33%/33%/33% T/GC/S Allocs
- 🟧 Fully Live Humongous
- 🟥 Fully Live Humongous + Pinned
- 🟨 1/3 Live + Collection Set
- 🟥 1/3 Live + Pinned
- 🟧 1/3 Live + Pinned CSet
- ⬜ Age 0+
- ⬜ Age 3+
- ⬜ Age 6+
- ⬜ Age 9+
- ⬜ Age 12+
- ⬜ Age 15+

Humongous region and humongous continuation region identified by concurrent marking coloring

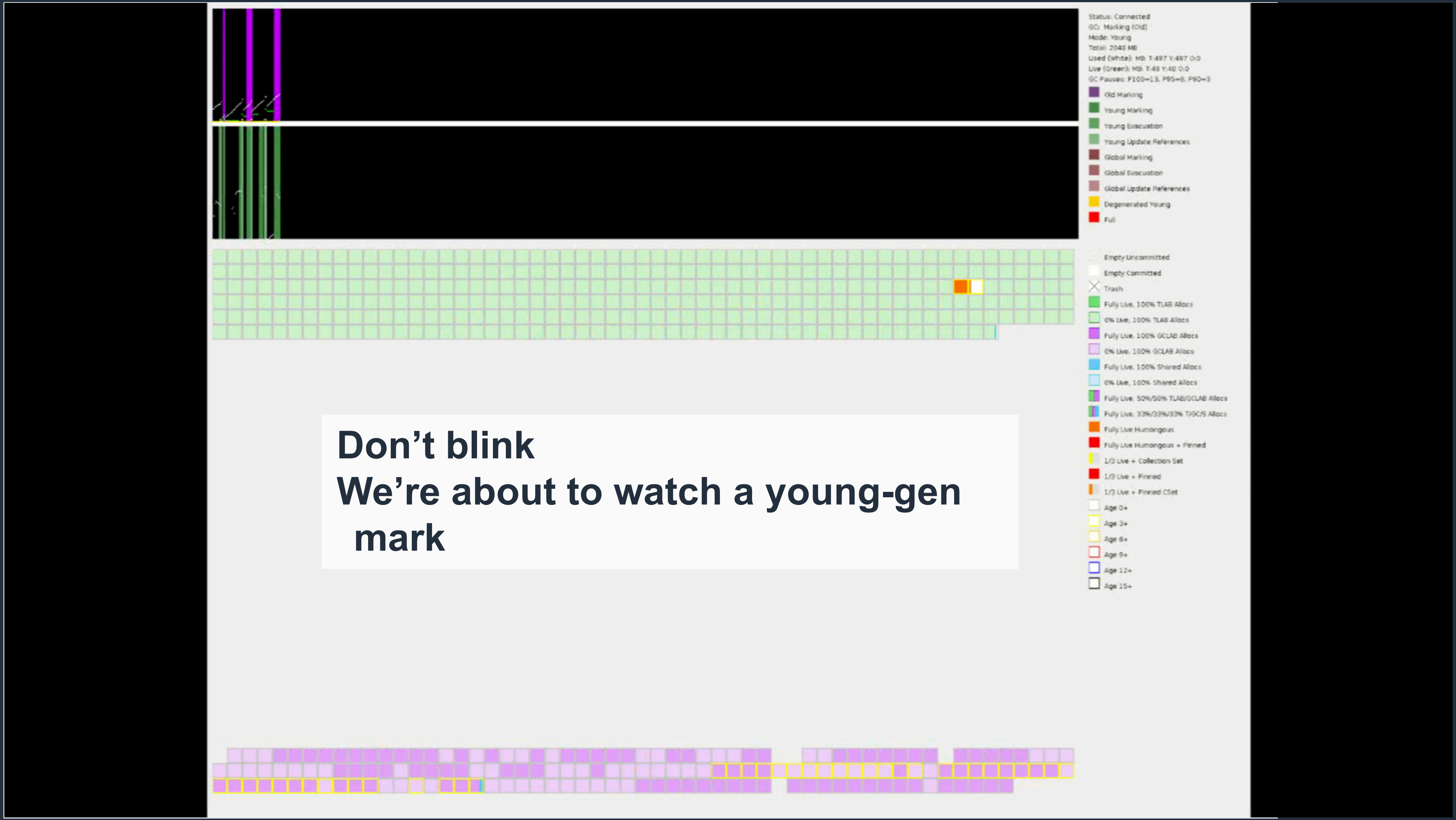
Visualization of Concurrent Old- and Young-Gen GC


Old Marking
Young Marking
Young Evacuation
Young Update References
Global Marking
Global Evacuation
Global Update References
Degenerated Young
Full
Empty Uncommitted
Empty Committed
Trash
Fully Live, 100% TLAB Allocs
0% Live, 100% TLAB Allocs
Fully Live, 100% GCLAB Allocs
0% Live, 100% GCLAB Allocs
Fully Live, 100% Shared Allocs
0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
Fully Live Humongous
Fully Live Humongous + Pinned
1/3 Live + Collection Set
1/3 Live + Pinned
1/3 Live + Pinned CSet
Age 0+
Age 3+
Age 6+
Age 9+
Age 12+
Age 15+

Status: Connected
GC: Marking
Mode: Young
Total: 2048 MB
Used (White): MB: T:315 Y:315 O:0
Live (Green): MB: T:0 Y:0 O:0
GC Pauses: P100=0, P95=0, P90=0

But our old-gen does not yet hold any objects
heuristic "learning" purposes
utilization

Visualization of Concurrent Old- and Young-Gen GC



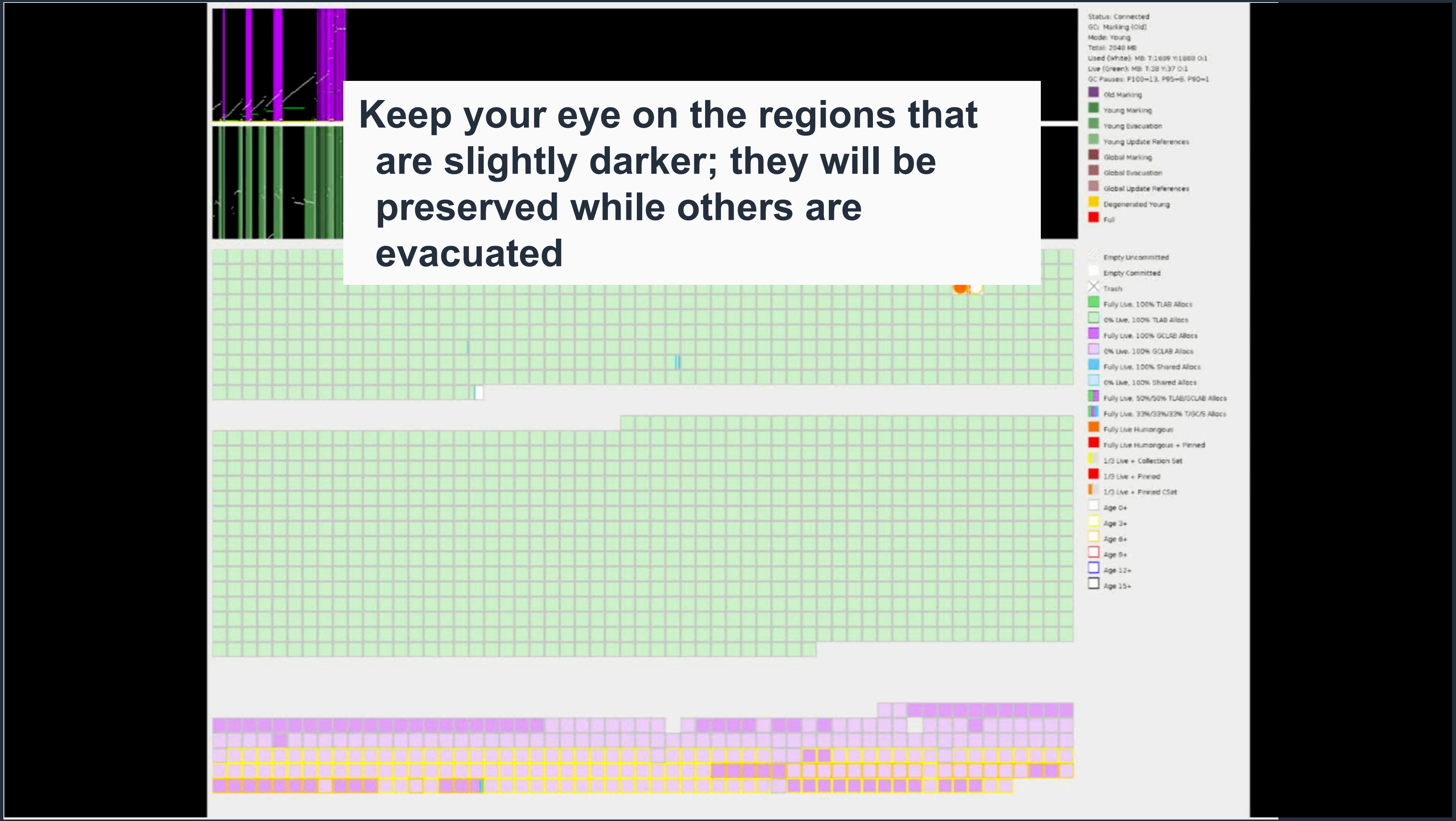
Status: Connected
GC: Marking (Old)
Mode: Young
Total: 2048 MB
Used (White): MB: T:497 Y:497 O:0
Live (Green): MB: T:48 Y:48 O:0
GC Pauses: P100=13, P95=8, P90=3

Old Marking
Young Marking
Young Evacuation
Young Update References
Global Marking
Global Evacuation
Global Update References
Degenerated Young
Full
Empty Uncommitted
Empty Committed
Trash
Fully Live, 100% TLAB Allocs
0% Live, 100% TLAB Allocs
Fully Live, 100% GCLAB Allocs
0% Live, 100% GCLAB Allocs
Fully Live, 100% Shared Allocs
0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
Fully Live Humongous
Fully Live Humongous + Pinned
1/3 Live + Collection Set
1/3 Live + Pinned
1/3 Live + Pinned CSet
Age 0+
Age 3+
Age 6+
Age 9+
Age 12+
Age 15+

Don’t blink

We’re about to watch a young-gen mark

Visualization of Concurrent Old- and Young-Gen GC

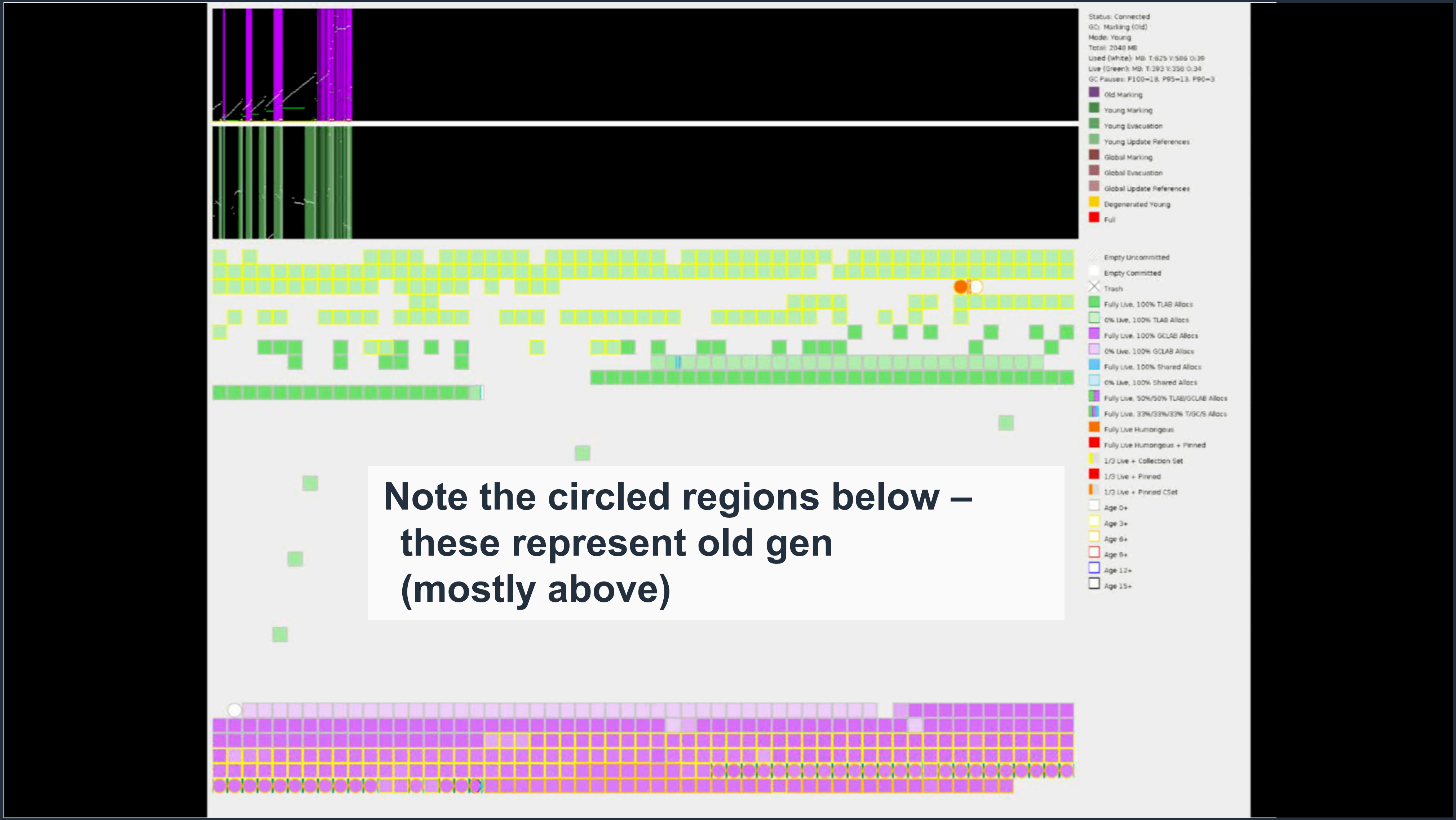


Keep your eye on the regions that are slightly darker; they will be preserved while others are evacuated

```
Status: Connected
GC:  Marking (Old)
Mode: Young
Total: 2048 MB
Used (White): MB: T:1689 Y:1688 O:1
Live (Green): MB: T:38 Y:37 O:1
GC Pauses: P100=13, P95=8, P90=1
```

Old Marking
Young Marking
Young Evacuation
Young Update References
Global Marking
Global Evacuation
Global Update References
Degenerated Young
Full
Empty Uncommitted
Empty Committed
Trash
Fully Live, 100% TLAB Allocs
0% Live, 100% TLAB Allocs
Fully Live, 100% GCLAB Allocs
0% Live, 100% GCLAB Allocs
Fully Live, 100% Shared Allocs
0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
Fully Live Humongous
Fully Live Humongous + Pinned
1/3 Live + Collection Set
1/3 Live + Pinned
1/3 Live + Pinned CSet
Age 0+
Age 3+
Age 6+
Age 9+
Age 12+
Age 15+

Visualization of Concurrent Old- and Young-Gen GC

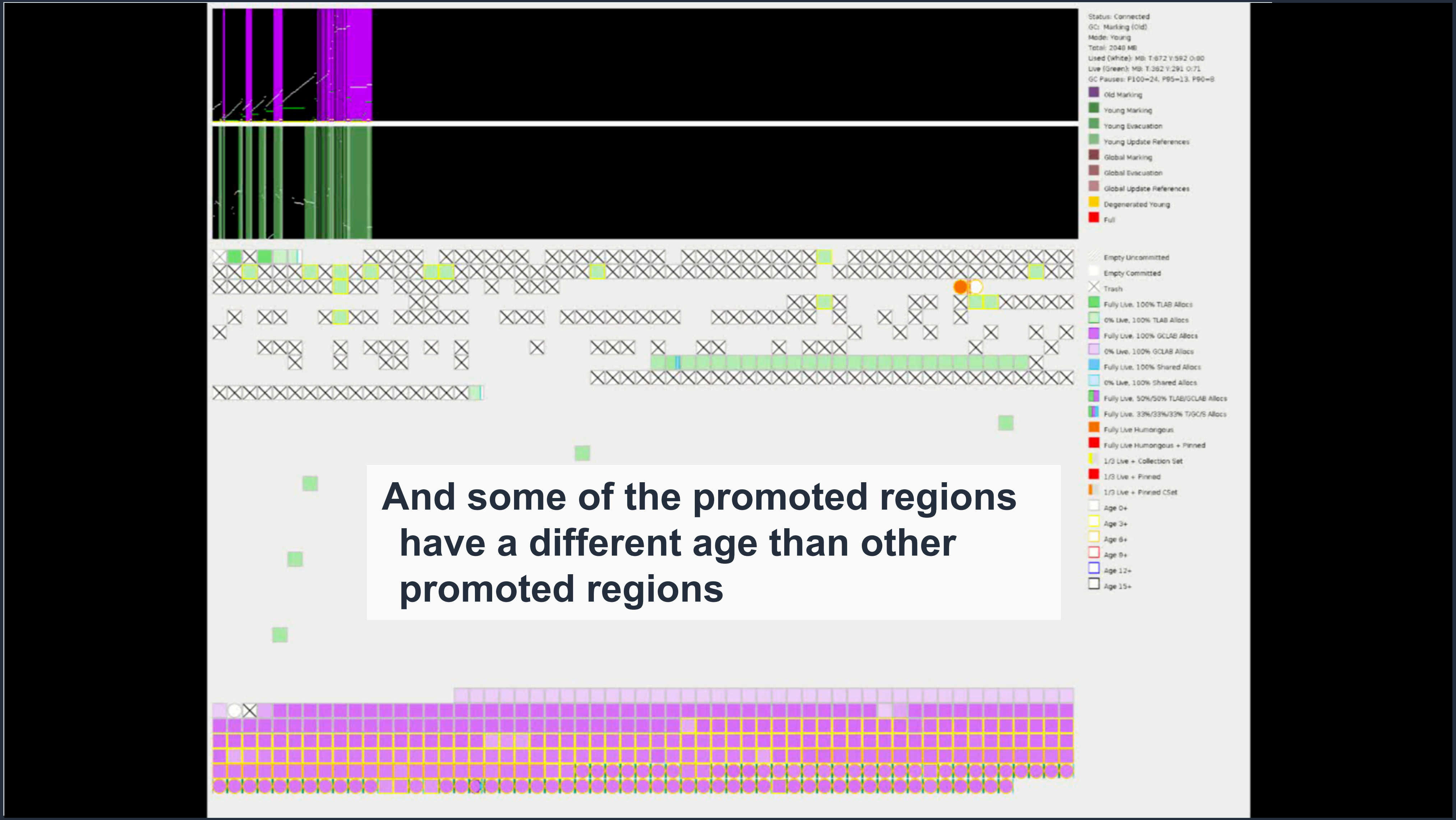


```
Status: Connected
GC:  Marking (Old)
Mode: Young
Total: 2048 MB
Used (White): MB: T:625 Y:586 O:39
Live (Green): MB: T:393 Y:358 O:34
GC Pauses: P100=18, P95=13, P90=3
```

🟪 Old Marking
🟩 Young Marking
🟩 Young Evacuation
🟩 Young Update References
🟥 Global Marking
🟥 Global Evacuation
🟥 Global Update References
🟨 Degenerated Young
🟥 Full
▨ Empty Uncommitted
⬜ Empty Committed
✕ Trash
🟩 Fully Live, 100% TLAB Allocs
🟩 0% Live, 100% TLAB Allocs
🟪 Fully Live, 100% GCLAB Allocs
🟪 0% Live, 100% GCLAB Allocs
🟦 Fully Live, 100% Shared Allocs
🟦 0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
🟧 Fully Live Humongous
🟥 Fully Live Humongous + Pinned
1/3 Live + Collection Set
🟥 1/3 Live + Pinned
1/3 Live + Pinned CSet
⬜ Age 0+
⬜ Age 3+
⬜ Age 6+
⬜ Age 9+
⬜ Age 12+
⬜ Age 15+

Note the circled regions below – these represent old gen (mostly above)

Visualization of Concurrent Old- and Young-Gen GC

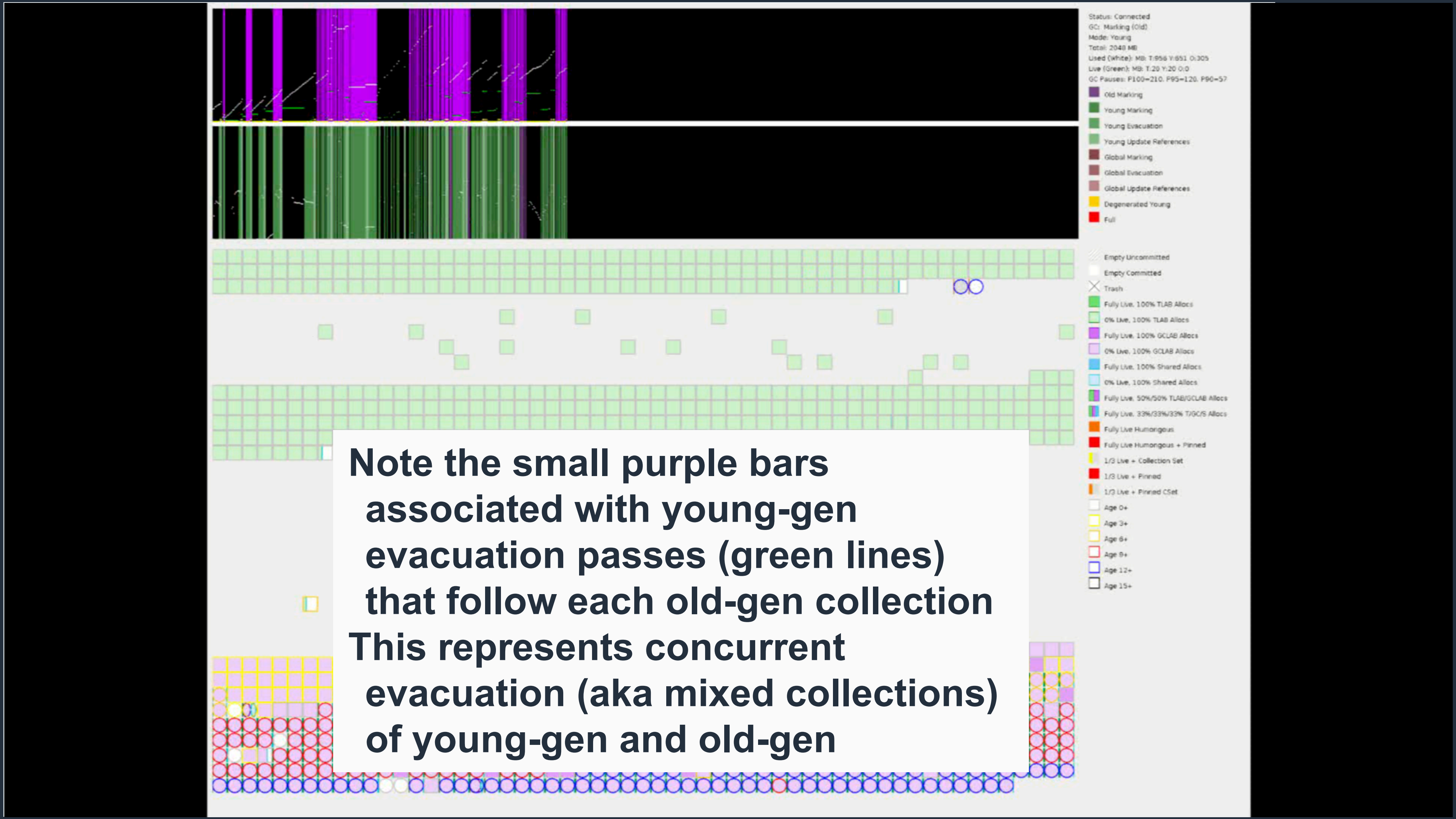


```
Status: Connected
GC:  Marking (Old)
Mode: Young
Total: 2048 MB
Used (White): MB: T:672 Y:592 O:80
Live (Green): MB: T:362 Y:291 O:71
GC Pauses: P100=24, P95=13, P90=8
```

Old Marking
Young Marking
Young Evacuation
Young Update References
Global Marking
Global Evacuation
Global Update References
Degenerated Young
Full
Empty Uncommitted
Empty Committed
Trash
Fully Live, 100% TLAB Allocs
0% Live, 100% TLAB Allocs
Fully Live, 100% GCLAB Allocs
0% Live, 100% GCLAB Allocs
Fully Live, 100% Shared Allocs
0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
Fully Live Humongous
Fully Live Humongous + Pinned
1/3 Live + Collection Set
1/3 Live + Pinned
1/3 Live + Pinned CSet
Age 0+
Age 3+
Age 6+
Age 9+
Age 12+
Age 15+

And some of the promoted regions have a different age than other promoted regions

Visualization of Concurrent Old- and Young-Gen GC



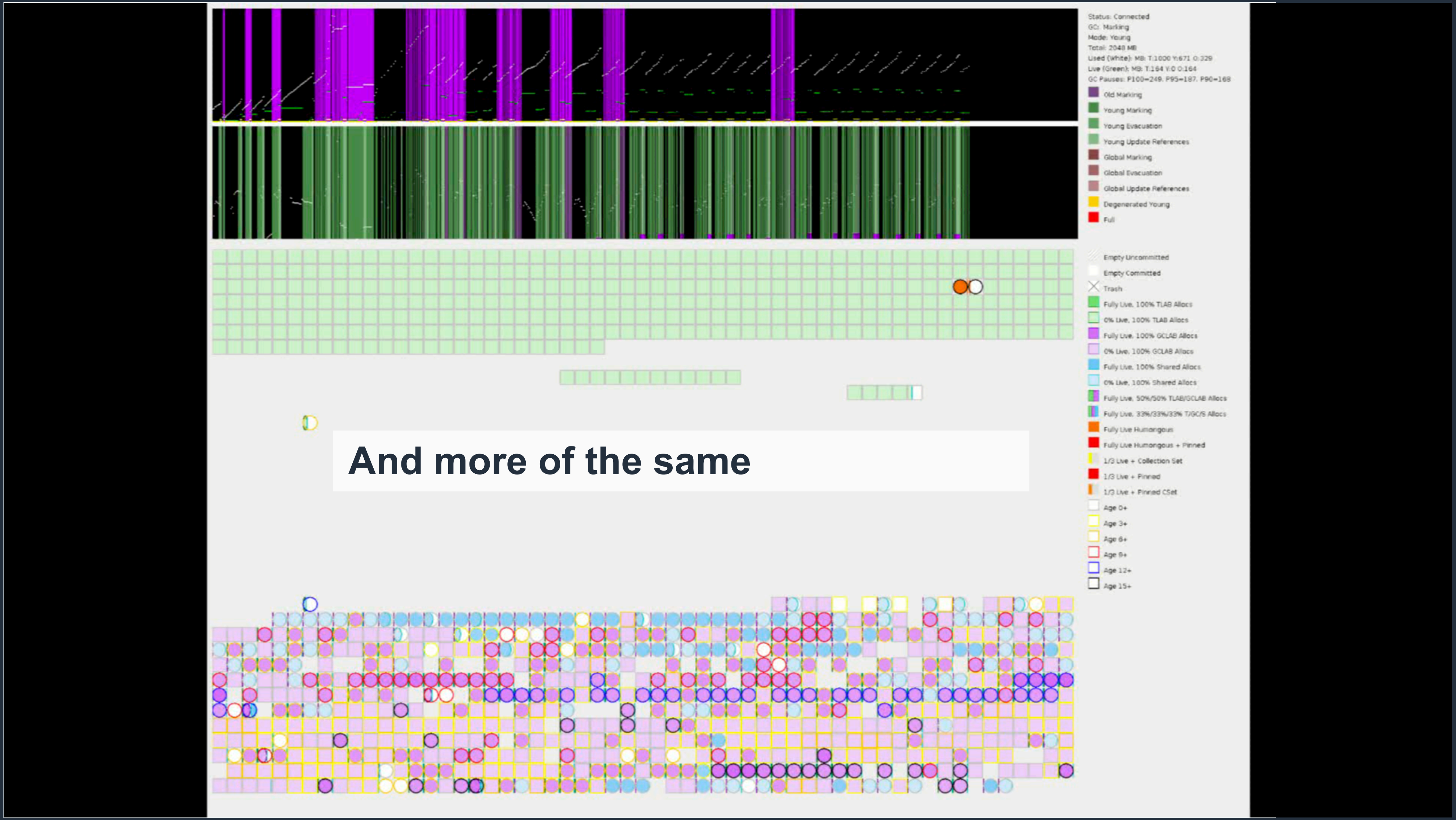
```
Status: Connected
GC:  Marking (Old)
Mode: Young
Total: 2048 MB
Used (White): MB: T:956 Y:651 O:305
Live (Green): MB: T:20 Y:20 O:0
GC Pauses: P100=210, P95=120, P90=57
```

Old Marking
Young Marking
Young Evacuation
Young Update References
Global Marking
Global Evacuation
Global Update References
Degenerated Young
Full
Empty Uncommitted
Empty Committed
Trash
Fully Live, 100% TLAB Allocs
0% Live, 100% TLAB Allocs
Fully Live, 100% GCLAB Allocs
0% Live, 100% GCLAB Allocs
Fully Live, 100% Shared Allocs
0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
Fully Live Humongous
Fully Live Humongous + Pinned
1/3 Live + Collection Set
1/3 Live + Pinned
1/3 Live + Pinned CSet
Age 0+
Age 3+
Age 6+
Age 9+
Age 12+
Age 15+

Note the small purple bars associated with young-gen evacuation passes (green lines) that follow each old-gen collection

This represents concurrent evacuation (aka mixed collections) of young-gen and old-gen

Visualization of Concurrent Old- and Young-Gen GC



Status: Connected
GC: Marking
Mode: Young
Total: 2048 MB
Used (White): MB: T:1000 Y:671 O:329
Live (Green): MB: T:164 Y:0 O:164
GC Pauses: P100=249, P95=187, P90=168

🟪 Old Marking
🟩 Young Marking
🟩 Young Evacuation
🟩 Young Update References
🟥 Global Marking
🟥 Global Evacuation
🟥 Global Update References
🟨 Degenerated Young
🟥 Full
▨ Empty Uncommitted
⬜ Empty Committed
✕ Trash
🟩 Fully Live, 100% TLAB Allocs
🟩 0% Live, 100% TLAB Allocs
🟪 Fully Live, 100% GCLAB Allocs
🟪 0% Live, 100% GCLAB Allocs
🟦 Fully Live, 100% Shared Allocs
🟦 0% Live, 100% Shared Allocs
Fully Live, 50%/50% TLAB/GCLAB Allocs
Fully Live, 33%/33%/33% T/GC/S Allocs
🟧 Fully Live Humongous
🟥 Fully Live Humongous + Pinned
1/3 Live + Collection Set
🟥 1/3 Live + Pinned
1/3 Live + Pinned CSet
⬜ Age 0+
⬜ Age 3+
⬜ Age 6+
⬜ Age 9+
⬜ Age 12+
⬜ Age 15+

And more of the same

Milestones for Early Access Release

- Milestone 9: Performance optimizations
  - Concurrent remembered set scanning
  - More efficient coalesce and fill of dead objects in old-gen
  - Old-gen allocation from PLABs
  - Perform maintenance upon retiring old-gen PLABs
- Milestone 10: Functional Improvements
  - Reconstitute Full GC
  - May add support for incremental-update mode of operation
- Milestone 11: Improved Metrics
  - Average and burst allocation and promotion rates
  - Ages at which objects become garbage within young-gen and old-gen regions
  - Distribution by age of live objects within young-gen and old-gen regions
  - Total live memory usage
  - Fragmentation within young-gen and old-gen memory regions
  - Performance counters: objects and bytes copied by mutators, frequency of copy collisions, objects and bytes abandoned following copy collisions

Further Enhancements Anticipated for Generally Available Release

- Existing Shenandoah Heuristics Need to be Enhanced:
  - Numbers of processor cores dedicated to concurrent GC
  - Sizes of young-gen and old-gen memory regions
  - Triggering frequencies for young-gen and old-gen collections
  - Frequency at which object ages are incremented
  - Age at which to promote young-gen objects
  - Bounds on young-gen and old-gen memory collected by each evacuation pass
  - Memory utilization thresholds at which old-gen regions become candidates for evacuation
  - Memory utilization thresholds at which young-gen regions become candidates for evacuation
  - Bias old-gen collection set selection towards regions that are more fragmented or have not been “recently” collected
  - Triggering frequencies for class unloading, reference processing, string deduplication, and heuristic refinement background tasks

Possible Far-Future Improvements after Generally Available Release

- Adaptive degenerated GC modes: subset of all mutator threads take on GC activities (rather than all mutator threads)
- Augment compacting old-gen GC with compact-in-place by repurposing dead areas within heap regions not evacuated
- Use virtual memory mapping of heap regions to reduce fragmentation of humongous allocations
- Pack allocations into remnant memory within humongous regions
- Replace direct card marking with thread-local buffer of reference-writes for remembered set maintenance
- Combine update-references of GC pass N with concurrent mark of GC pass N+1
- Enhance old-gen remembered set to reduce update-refs effort following old-gen evacuation
- Explore use of from-space invariant instead of to-space invariant
- Optimize for NUMA deployments

Summary

- Generational Shenandoah sustains higher allocation rates and higher memory utilization than non-generational Shenandoah
- An EA release offers these benefits when configured by manual methods
- A subsequent GA release reduces manual tuning efforts by providing heuristics to auto-tune configuration and adapt to evolving workloads
- Further contemplated improvements expect to offer improvements in throughput and latency in subsequent production releases of Generational Shenandoah

- Thank you to my colleagues on the Corretto JVM team who join with me in implementing these capabilities and have contributed to this presentation