Garbage-First Garbage Collector

Migration to, Expectations and Advanced Tuning
[CON3754]

Charlie Hunt
Monica Beckwith
John Cuthbertson

What to Expect

Learn what you need to know to experience nirvana in the evaluation of G1 GC even if your are migrating from Parallel GC to G1, or CMS GC to G1 GC

You also get a walk through of some case study data

Who are these guys?

About us

- Charlie Hunt
  - Architect, Performance Engineering, Salesforce.com
- Monica Beckwith
  - Performance Architect, Servergy
- John Cuthbertson
  - GC Engineer, Azul Systems

Agenda

Three Topics

- Migration / Evaluation Prep
- Parallel GC to G1 GC
- CMS GC to G1 GC

And, at the end

- Summary of things to watch for during your G1 evaluation

Prep Work

Prep Work

- Want less stress in your performance testing?

Prep Work

- Want less stress in your performance testing?
  - Define success in terms of throughput, latency, footprint and capacity
- Approach
  - Get your application stakeholders together and start asking questions!

Throughput

- What's the expected throughput?
  - Can you fall below that expected throughput?
    - How long can you fall below that expected throughput?
  - What throughput can you never fall below?
- How is throughput measured?
  - Txns per sec, messages per sec, one metric, multiple metrics?
- Where is throughput measured?
  - Application server? At the client?

Latency

- What's the expected latency?
  - Can you exceed the expected latency?
    - By how much, and for how long?
    - What latency should you never exceed?
- How is latency measured?
  - Response time, percentile(s), one metric, multiple metrics?
- Where is latency measured?
  - Application server? At the client?

Footprint

- How much RAM / memory can you use?
- What amount of RAM / memory should you never exceed?
- How is memory consumption measured?
  - What tools?
  - At what time during application execution?

Capacity

- What's the expected load?
  - i.e. concurrent users, number of concurrent txns, injection rate
- How much additional (load) capacity do you need?
  - i.e. concurrent users, number of concurrent txns, injection rate
- Capacity metrics are often some combination of
  - # of concurrent users, an injection rate, number of concurrent txns in flight and/or CPU utilization

One last one …. Power Consumption

- Power consumption may also be a requirement
- What's the expected power consumption?
- FYI - power consumption BIOS settings can impact throughput, latency and capacity
  - Can observe as much as 15% difference in latency and CPU utilization

Experimental Design

- Now you've got the inputs to begin experimental design
- Do you have a workload that be configured to test the performance requirements?
  - You may have to develop or enhance a workload
- Do you have an environment that can test the requirements you've gathered?

Share the Experiment Design(s)

- Once you have the set of experiments to test defined
  - Share those plans with your stakeholders
    - It's a great way to validate your testing for the right things in an appropriate manner
    - You'll likely get feedback and tweak your experiment's design(s)
  - Most of all, you get agreement on what's being tested, and how
    - Saves you a lot of hassle!

Migrating from Parallel GC to G1 GC

When to consider migration

- When Parallel GC can't be tuned to meet latency goals
  - Usually due to Full GC pauses
    - Duration and/or frequency
- When Parallel GC can't meet footprint and latency goals
  - G1 will likely be able to run with smaller Java heap and shorter pauses than Parallel GC's full GCs
- Max load capacity will likely be lower with G1
  - G1 has more overhead than Parallel GC

When to consider migration

- If you can tune Parallel GC to meet your throughput, latency, footprint and capacity goals
  - Stay with Parallel GC!
    - If you can tune Parallel GC to meet latency, footprint and capacity goals, it offers the best throughput of the HotSpot GCs

Parallel GC Heap Sizing

- Suppose an application using Parallel GC and these heap sizing options

  -Xms5g -Xmx5g -Xmn2g

  -XX:PermSize=700m -XX:MaxPermSize=700m

  -XX:-UseAdaptiveSizePolicy -XX:SurvivorRatio=4

  -XX:+UseParallelOldGC

  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps

Does this make sense?

- Updated for G1

-Xms5g -Xm -Xmn2g

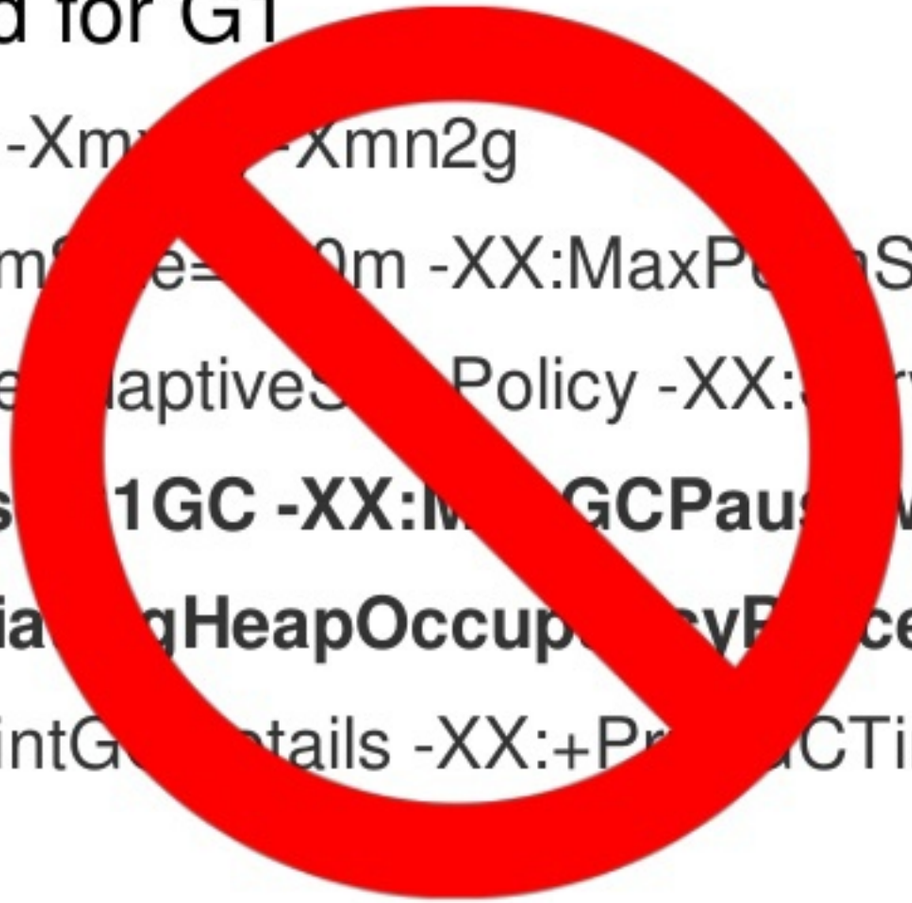
-XX:Perm e= m -XX:MaxP Size=700m

-XX:-Use aptive Policy -XX: rvivorRatio=4

-XX:+Us 1GC -XX:M GCPau Millis=250

-XX:Initia gHeapOccup cy cent=45 (default is 45)

-XX:+PrintG tails -XX:+Pr CTimeStamps



Remove all young gen size refining options

- Updated for G1

  -Xms5g -Xmx5g ~~-Xmn2g~~

  -XX:PermSize=700m -XX:MaxPermSize=700m

  ~~-XX:-UseAdaptiveSizePolicy -XX:SurvivorRatio=4~~

  -XX:+UseG1GC -XX:MaxGCPauseMillis=250

  -XX:InitiatingHeapOccupancyPercent=45 (default is 45)

  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps

Example Application Requirements

- At expected load
  - 2 second response time at 99th percentile
  - No response time above 10 seconds
- 2x expected load
  - 2 second response time at 99th percentile
- 6 GB RAM available to JVM

Current Observations

- With Parallel GC at expected capacity / load
  - Observing 2.5 second latency at 99th percentile
    - Above 2 second 99th percentile requirement
  - ~ 29% CPU utilization
- 2x expected load not evaluated
  - Why bother? Can't meet 99th percentile requirement for expected capacity / load!

Again, our Parallel GC heap sizing

-Xms5g -Xmx5g -Xmn2g

-XX:PermSize=700m -XX:MaxPermSize=700m

-XX:-UseAdaptiveSizePolicy -XX:SurvivorRatio=4

-XX:+UseParallelOldGC

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps

Again, our starting place for G1 GC

-Xms5g -Xmx5g

-XX:PermSize=700m -XX:MaxPermSize=700m

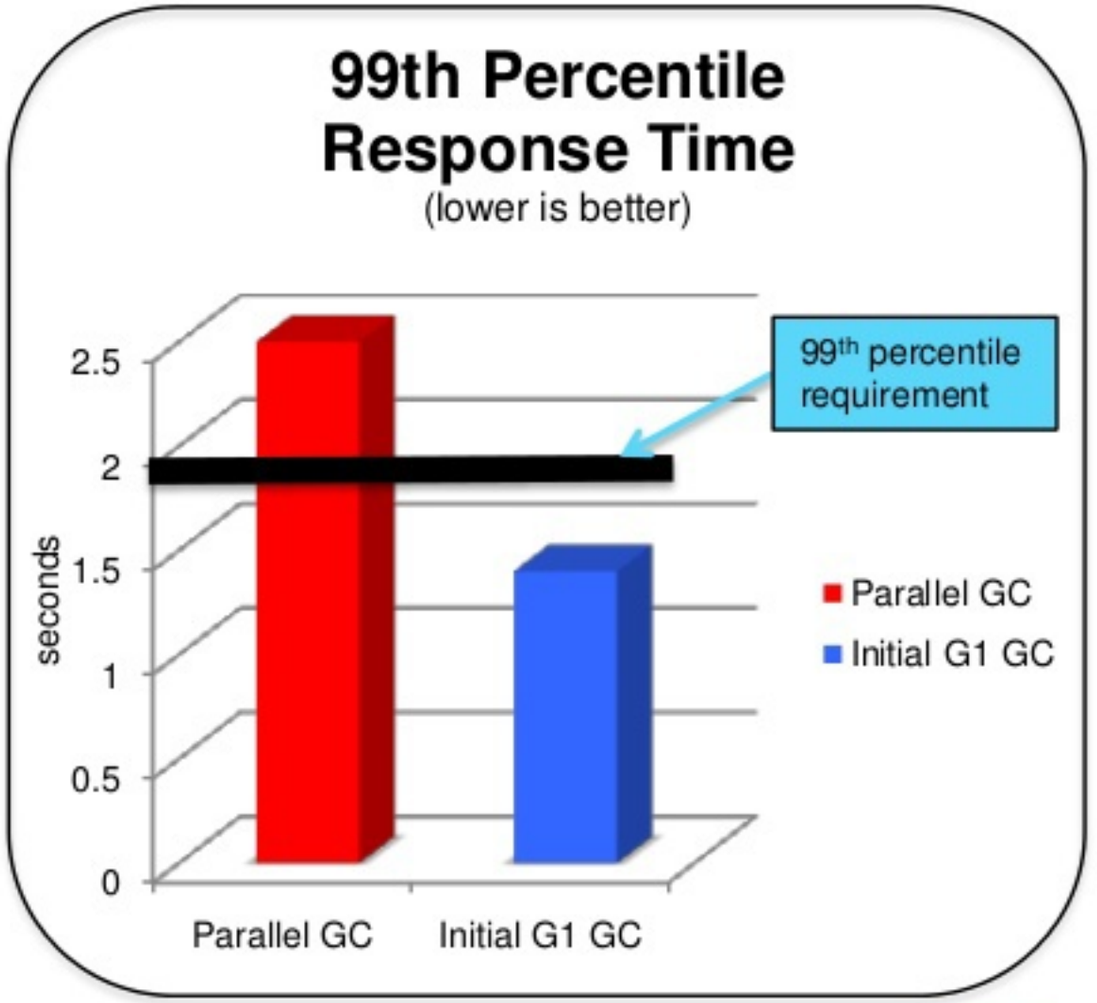
-XX:+UseG1GC -XX:MaxGCPauseMillis=250

-XX:InitiatingHeapOccupancyPercent=45

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps

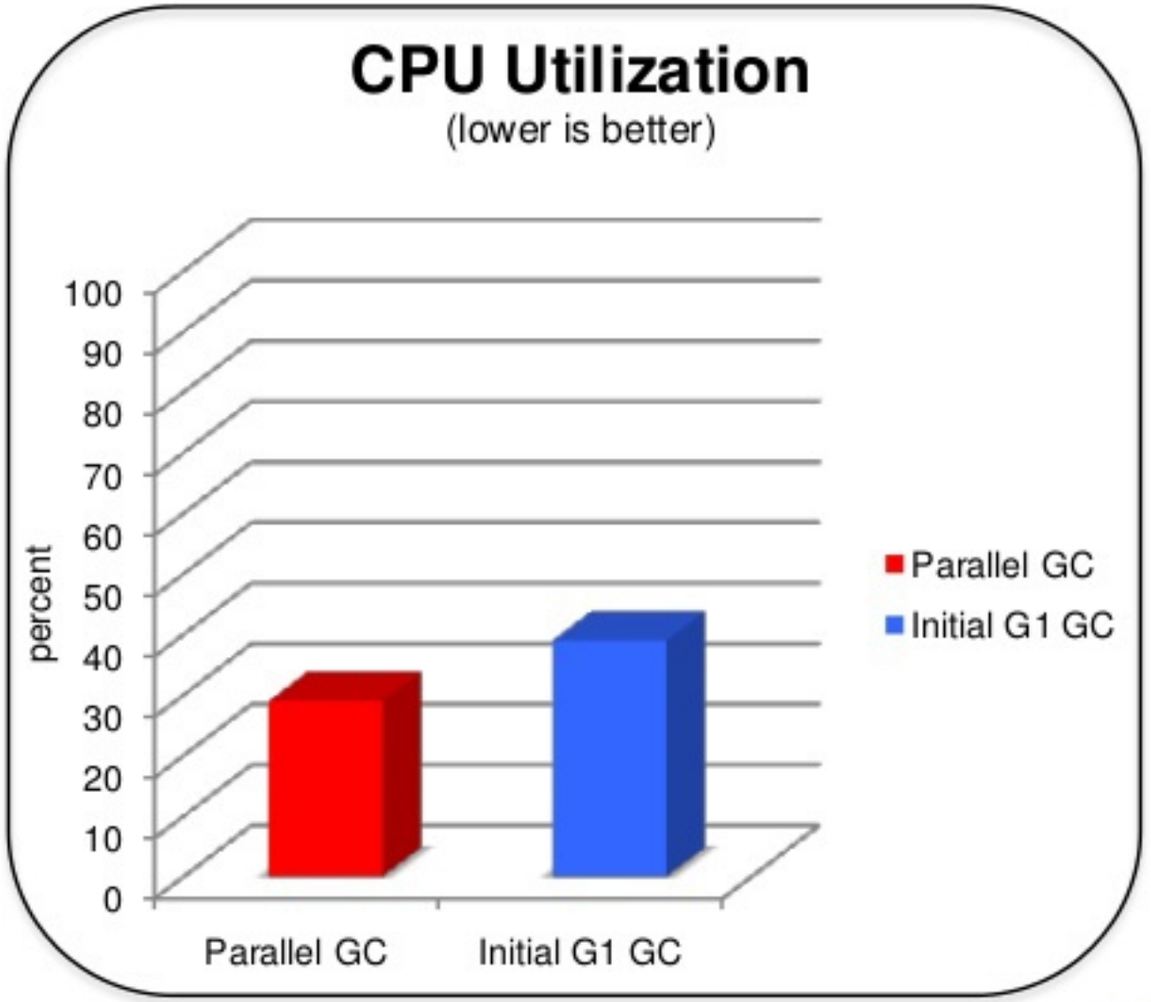
First G1 GC results

- Initial observations
  - 99th percentile response time
    - G1 GC - 1.4 seconds
    - Parallel GC - 2.5 seconds
      - ~ 44% reduction



First G1 GC results

- Initial observations
  - CPU utilization
    - G1 GC - 39%
    - Parallel GC – 29%



Analysis

- Are we done?
  - Should we next test for 2x capacity?
  - No, go look at the raw G1 GC logs for potential improvement!!!
  - Additionally, that jump in CPU utilization is a bit concerning, 29% to 39%
    - How will the app behave as the injection rate increases towards our capacity goals of 2x expected load?

G1 GC Analysis

- Starting point
  - Look for “to-space overflow”, “to-space exhausted” and “Full GCs” in the G1 GC logs
    - #1 goal is to avoid these with G1 GC!

Evacuation failures (i)

- to-space overflow and to-space exhausted
  - These are G1 region evacuation failures
  - Situations where no G1 regions are available to evacuate live objects
    - Note, the region being evacuated could be an eden region, survivor region or an old region – i.e. “to-space overflow” is not limited to a survivor space overflow situation like what can occur with other HotSpot GCs

Evacuation failures (ii)

- to-space overflow and to-space exhausted
  - (Very) expensive events to deal with
    - Any object that has already been evacuated, its object references must be updated, and the region must be tenured
    - Any object that has not been evacuated, its object references must be self-forwarded and the region tenured in place
  - Evacuation failures are usually followed by a Full GC
    - Full GC collects all regions, and is single threaded

G1 GC Analysis

- What did we find?

About every 25 minutes …

2216.206: [GC pause (young) (**to-space overflow**), 0.62894300 secs]

[Parallel Time: 239.4 ms] …

2217.958: [**Full GC** 4802M->1571M(5120M), 3.0889520 secs]

[Times: user=4.34 sys=0.01, real=3.09 secs]

This is not good!

G1 GC Analysis

- What next?
  - Enable -XX:+PrintAdaptiveSizePolicy to see what adaptive decisions G1 is making
  - Look at GC logs for those areas leading up to to-space overflows, to-space exhausted and Full GC

G1 GC Analysis

- What did we find?

  2219.546: [G1Ergonomics (Concurrent Cycles) ***request concurrent cycle initiation***, reason: occupancy higher than threshold, occupancy: 2311061504 bytes, ***allocation request: 6440816 bytes***, threshold: 2312110080 bytes (45.00 %), ***source: concurrent humongous allocation***]

  +250 of these in two hours!

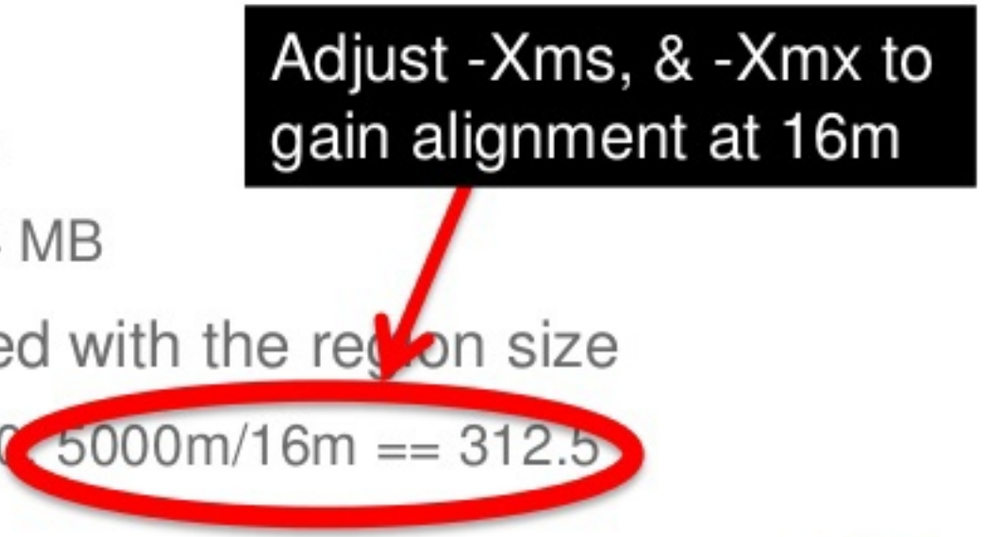
- Resolution: avoid humongous allocations

G1 GC Analysis

- What next?
  - Object allocations greater than or equal to 50% of G1's region size are considered a humongous allocation
    - Humongous object allocation & collection are not optimized in G1
  - G1 region size is determined automatically at JVM launch time based on the size of the Java heap (-Xms in 7u40 & later)
    - But you can override it with -XX:G1HeapRegionSize=#
      - Size must be a power of 2 ranging from 1 MB to 32 MB

G1 GC Analysis

- What next (continued)?
  - Object allocation from GC log, 6440816 bytes, or about 6.4 MB
  - Need a region size such that 6.4 MB is less than 50% of the region size, and where the region size is a power of 2 between 1 MB and 32 MB
    - Set -XX:G1HeapRegionSize=16m
      - 16 MB * 50% = 8 MB, 8 MB > 6.4 MB
    - Note: -Xms & -Xmx must be aligned with the region size
      - 5g == 5120m, 5120m/16m == 320, 5000m/16m == 312.5



Updated G1 GC Options

-Xms5g -Xmx5g

-XX:PermSize=700m -XX:MaxPermSize=700m

-XX:+UseG1GC -XX:MaxGCPauseMillis=250

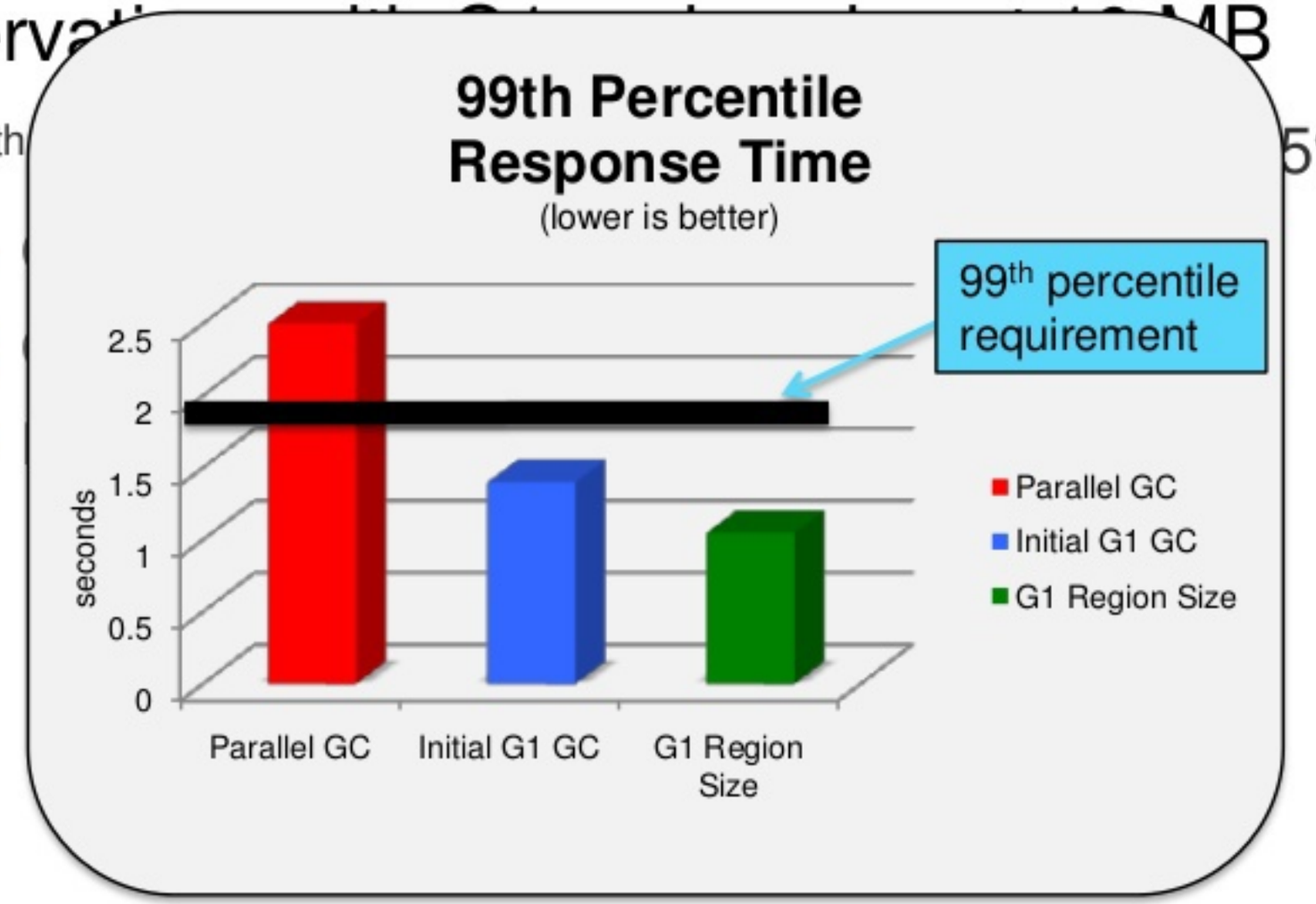
-XX:InitiatingHeapOccupancyPercent=45

-XX:G1HeapRegionSize=16m

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps

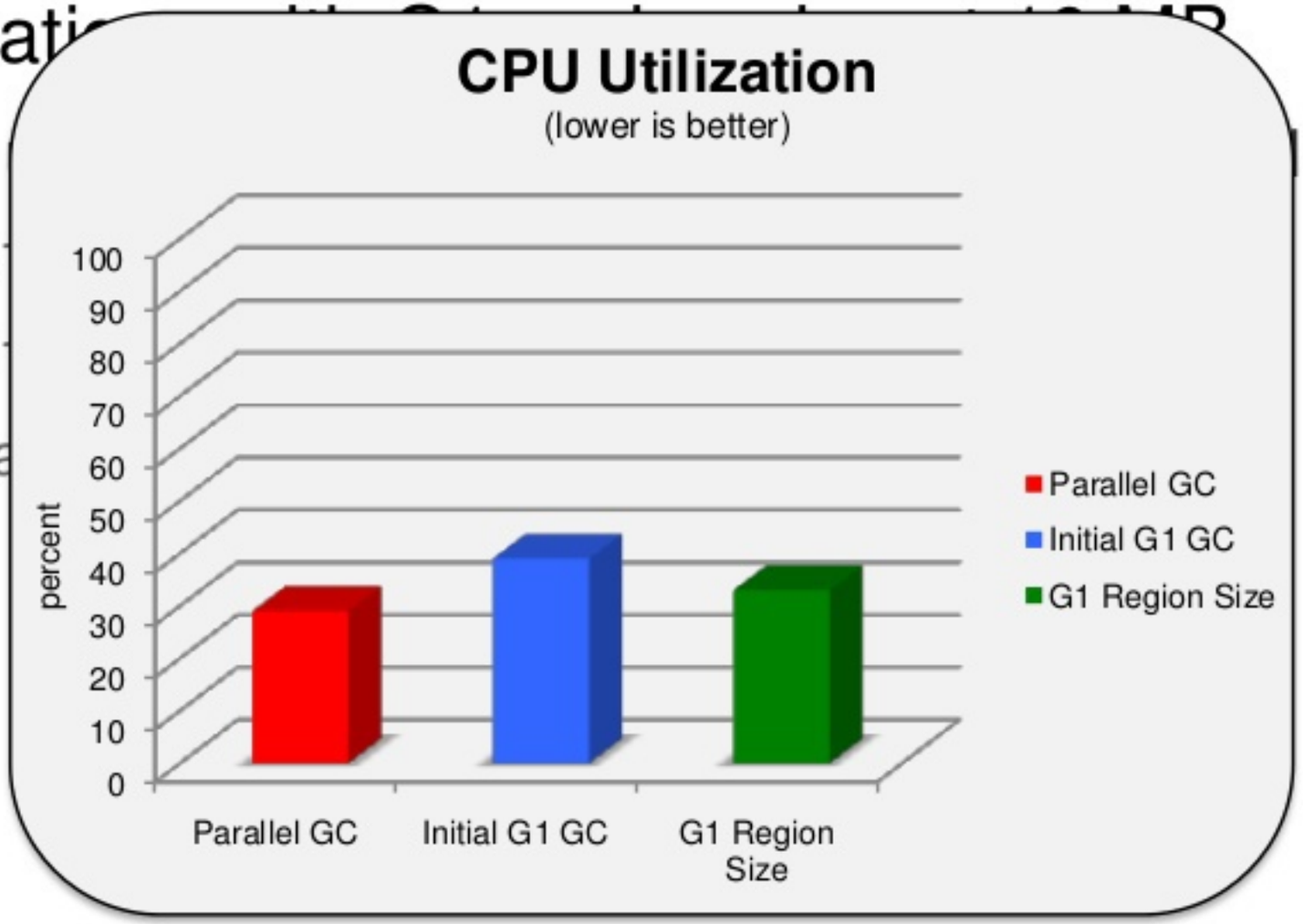
16 MB Region Size

- Observa
  - 99th
    - 5%
    -
    -
    -



16 MB Region Size

- Observati
  - CPU
    - G
    - G
    - Pa



el GC

Analysis

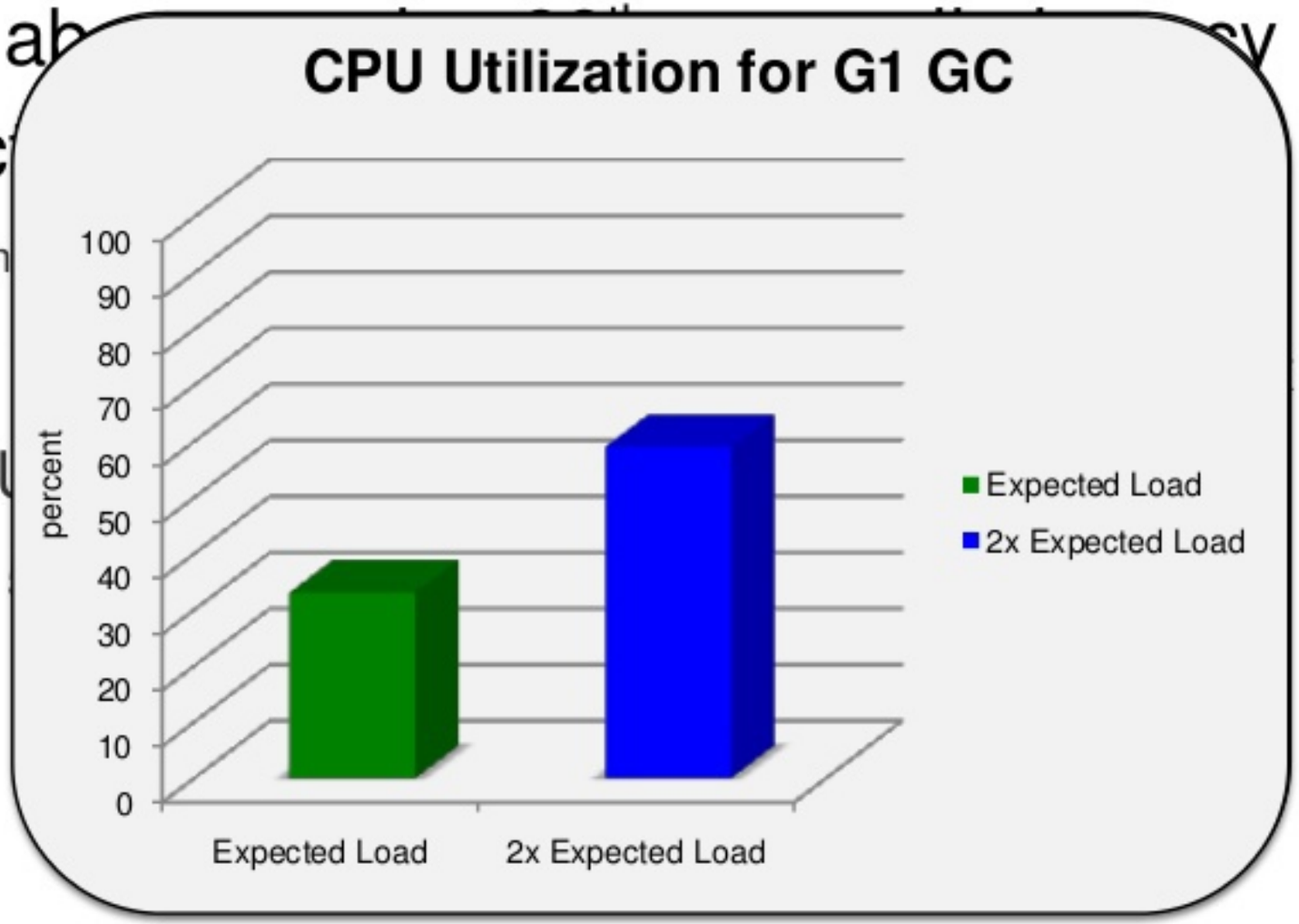
- Are we done?
  - No, go look at the raw G1 GC logs for some additional potential improvement!!!
  - CPU utilization still a little higher, 29% vs. 33%
  - Still need to know how the app will behave at 2x expected load capacity goal?

G1 GC Analysis

- What did we find?
  - No humongous allocations, no adaptive / ergonomic issues
  - Looking at the amount of space reclaimed per marking & mixed GC cycle
    - 300 MB to 900 MB per cycle, ~ 6% - 18% of the Java heap
    - ~ 95% heap occupancy just prior to mixed GC cycles
    - Seems pretty good
      - Delicate balance between starting the marking cycle too early and wasting CPU cycles at the expense of capacity vs starting too late and experiencing to-space overflow or to-space exhausted events

Analysis

- What ab… …y at 2x expec…
  - 99th …
    - …
  - CPU …
    - …



Analysis

- Are we done?
  - Yes, for now ;-)
    - Requirements met
    - Some additional performance realized via GC analysis

Analysis

- Are we done?
  - Yes, for now ;-)
    - Requirements met
    - Some additional performance realized via GC analysis
  - Stakeholders happy!

Migrating from CMS GC to G1 GC

When to consider migration

- When CMS GC cannot meet your latency goals
  - Usually due to fragmented old gen space, or not enough available Java heap
- When CMS GC has difficulties meeting footprint goals
  - G1 GC will likely run with a smaller Java heap
- G1 GC may use more CPU
  - G1 has more overhead than CMS GC
    - Though not as much of a delta as G1 GC to Parallel GC

Example Application Requirements

- Expected load
  - 1500 concurrent users, 2 Txns/user per second
- 99th percentile – 250 ms
  - Desire all response times to be less than 1 second
- 12 GB RAM per JVM
- 2x load capacity
  - 3000 users, 2 Txns/user per second

Current CMS GC Configuration

-Xms10g -Xmx10g -Xmn2g

-XX:PermSize=64m -XX:MaxPermSize=64m

-XX:InitialSurvivorRatio=5 -XX:SurvivorRatio=5

-XX:InitialTenuringThreshold=15 -XX:MaxTenuringThreshold=15

-XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:CMSInitiatingOccupancyFraction=60

-XX:+UseCMSInitiatingOccupancyOnly

-XX:+CMSClassUnloadingEnabled

-XX:ParallelGCThreads=12 -XX:ConcGCThreads=4

Does this make sense?

- Updated for G1

-Xms10g -Xm[illegible]g -Xmn2g

-XX:Perm[illegible]e=[illegible]m -XX:MaxPe[illegible]Size=64m

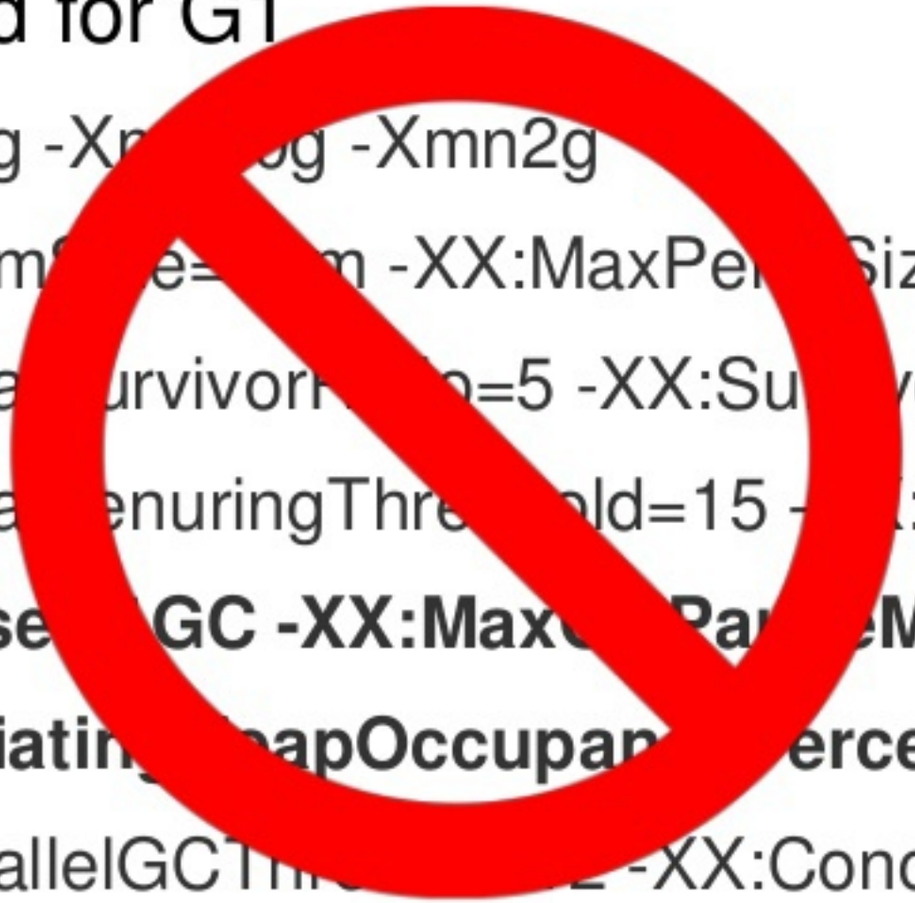
-XX:Initia[illegible]urvivorR[illegible]o=5 -XX:Su[illegible]vorRatio=5

-XX:Initia[illegible]enuringThre[illegible]old=15 -[illegible]:MaxTenuringThreshold=15

-XX:+Use[illegible]GC -XX:MaxG[illegible]Pa[illegible]eMillis=200

-XX:Initiatin[illegible]eapOccupan[illegible]ercent=50

-XX:ParallelGCThr[illegible]2 -XX:ConcGCThreads=4



CMS Configuration Updated for G1

-Xms10g -Xmx10g ~~-Xmn2g~~

-XX:PermSize=64m -XX:MaxPermSize=64m

~~-XX:InitialSurvivorRatio=5 -XX:SurvivorRatio=5~~

~~-XX:InitialTenuringThreshold=15 -XX:MaxTenuringThreshold=15~~

-XX:+UseG1GC -XX:MaxGCPauseMillis=200 (default is 200)

-XX:InitiatingHeapOccupancyPercent=50 (default is 45)

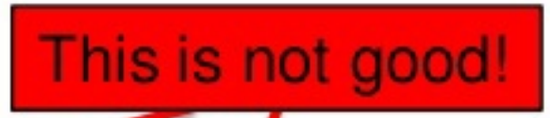
~~-XX:ParallelGCThreads=12 -XX:ConcGCThreads=4~~

Current Observations

- With CMS GC
  - At expected capacity
    - Meeting 99th percentile response time of 250 ms
      - 240 ms
    - Not meeting max response time less than 1 second
      - ~ 5 seconds
  - At 2x expected capacity
    - Not meeting 99th percentile (250 ms), or max response time < 1 sec
      - 99th percentile – 310 ms
      - Max response time ~ 8 seconds

Current Observations

- What did we find?
  - CMS GC promotion failures
  - Here's an example



[ParNew (promotion failed) 1797568K->1775476K(1797568K), 0.7051600 secs] [CMS: 3182978K->1773645K(8388608K), 5.2594980 secs] 4736752K->1773645K(10186176K), [CMS Perm : 52858K->52843K(65536K)] 5.9649620 secs] [Times: user=7.43 sys=0.00, real=5.96 secs]

Again, CMS GC Heap Sizing

-Xms10g -Xmx10g -Xmn2g

-XX:PermSize=64m -XX:MaxPermSize=64m

-XX:InitialSurvivorRatio=5 -XX:SurvivorRatio=5

-XX:InitialTenuringThreshold=15 -XX:MaxTenuringThreshold=15

-XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:CMSInitiatingOccupancyFraction=60

-XX:+UseCMSInitiatingOccupancyOnly

-XX:+CMSClassUnloadingEnabled

-XX:ParallelGCThreads=12 -XX:ConcGCThreads=4

Again, our starting place for G1 GC

-Xms10g -Xmx10g

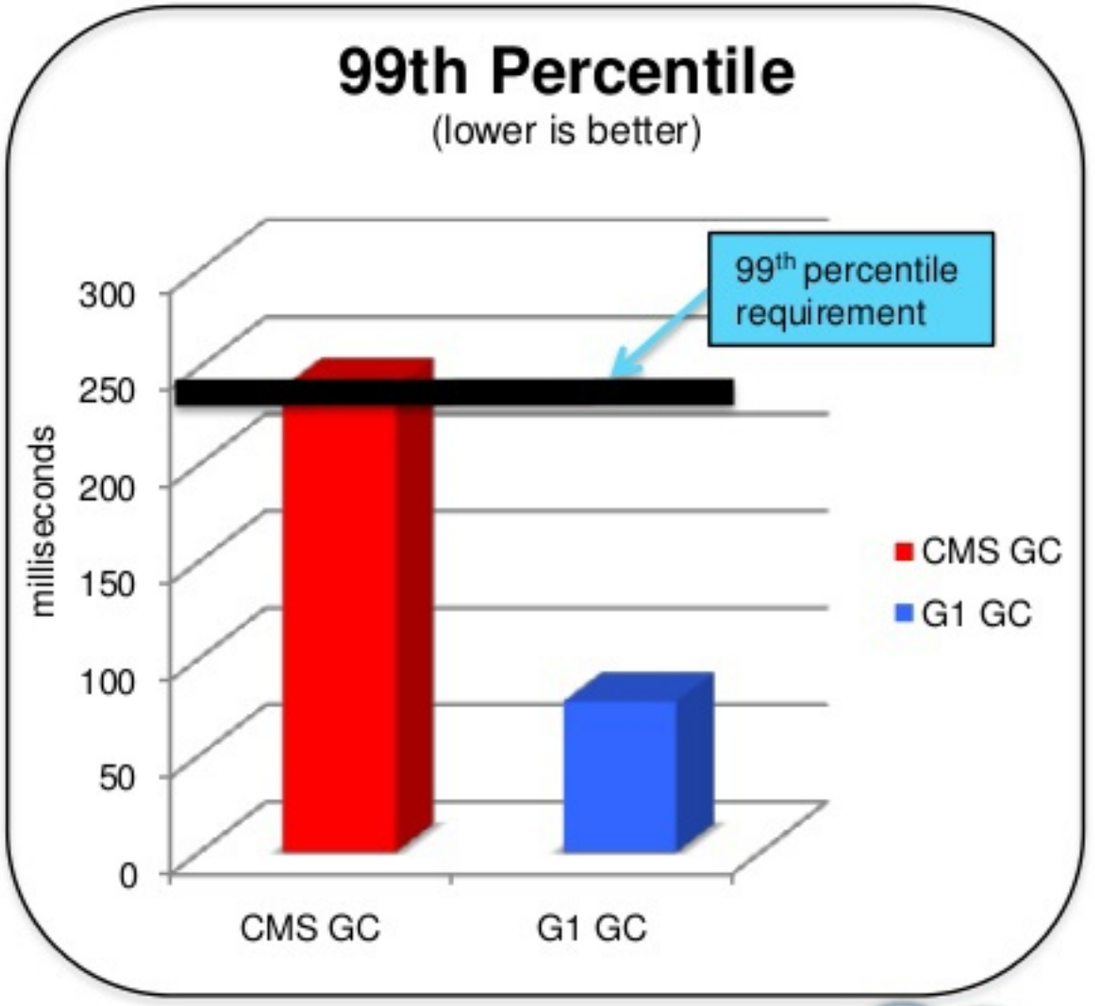
-XX:PermSize=64m -XX:MaxPermSize=64m

-XX:+UseG1GC -XX:MaxGCPauseMillis=200 (default is 200)

-XX:InitiatingHeapOccupancyPercent=50 (default is 45)

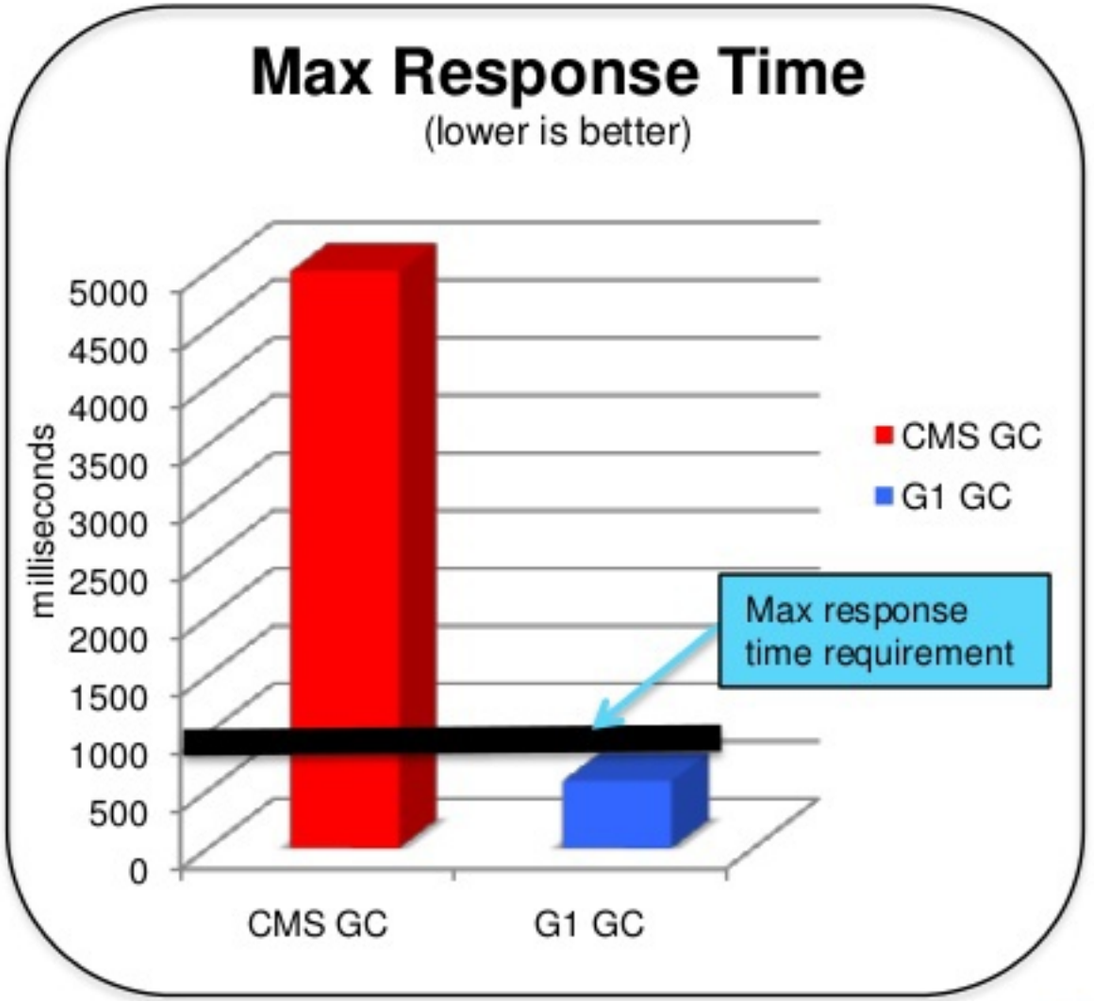
First G1 GC results (response times)

- Initial observations
  - 99th percentile at expected load
    - CMS – 240 ms
    - G1 – 78 ms



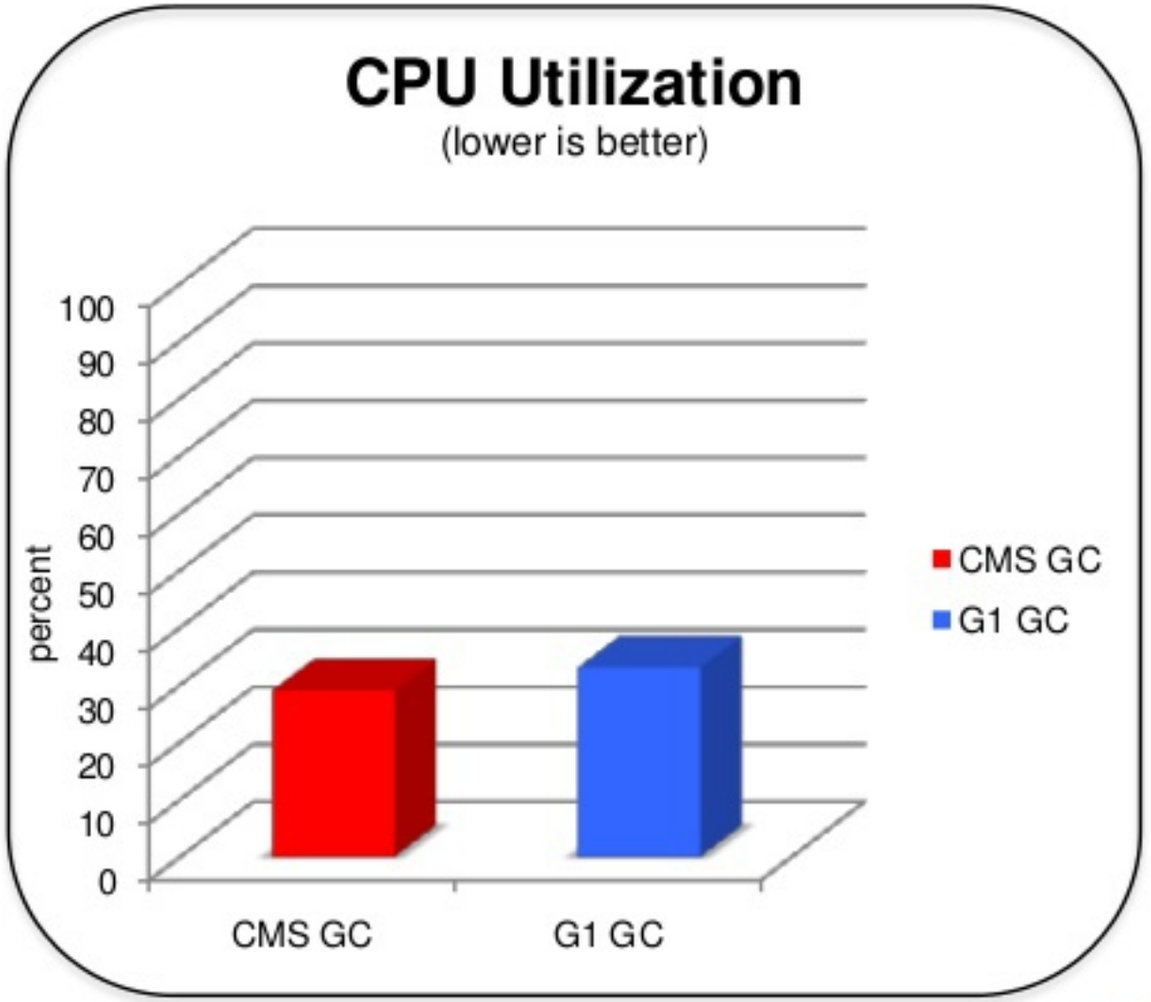
First G1 GC results (response times)

- Initial observations
  - Max at expected load
    - CMS – 5000 ms
    - G1 – 582 ms



First G1 GC results (CPU utilization)

- Initial observations
  - At expected load
    - CMS – 29%
    - G1 – 33%

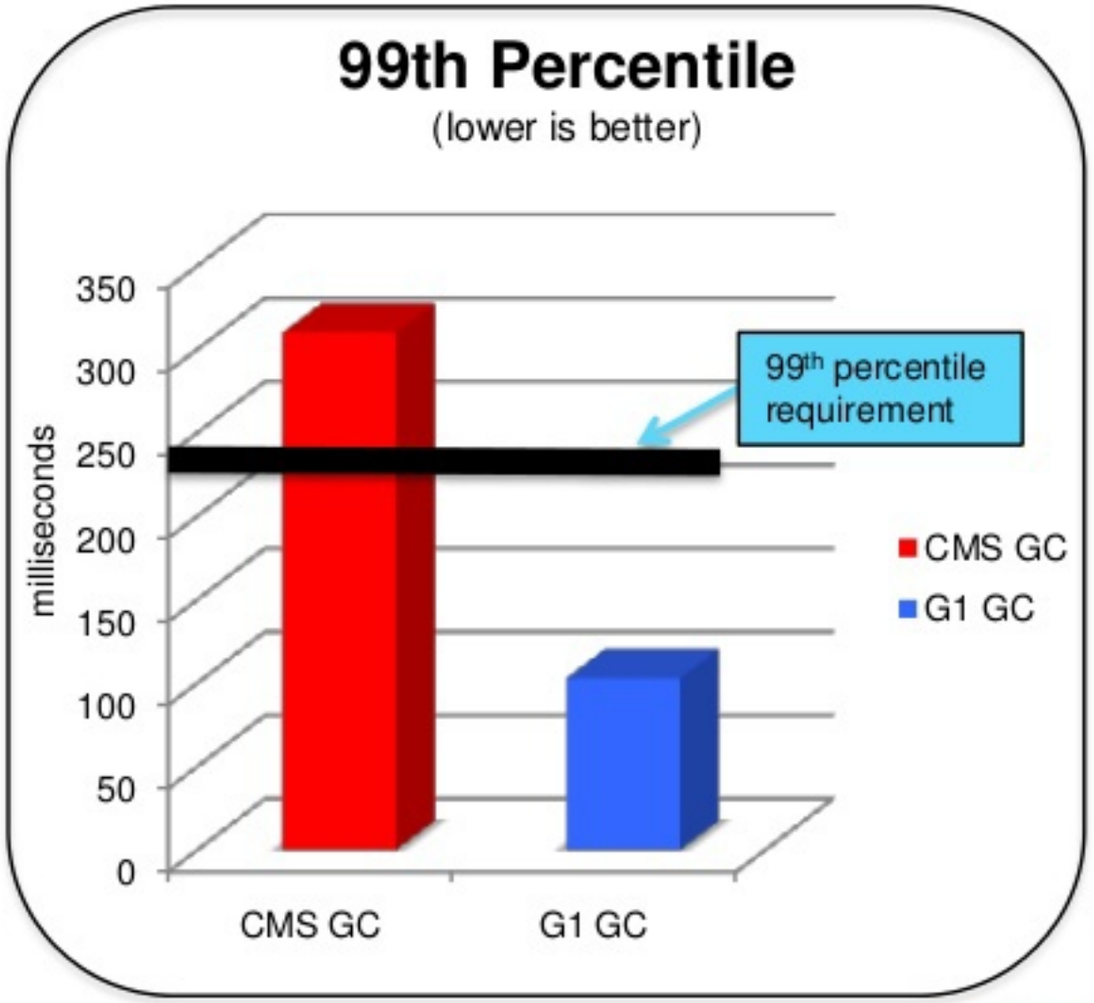


Analysis

- Are we done?
  - No, go look at the raw G1 GC logs for potential improvement!!!
  - Additionally, the increase in CPU utilization, (29% to 33%) may be a little concerning
  - How will the app behave at 2x load capacity?

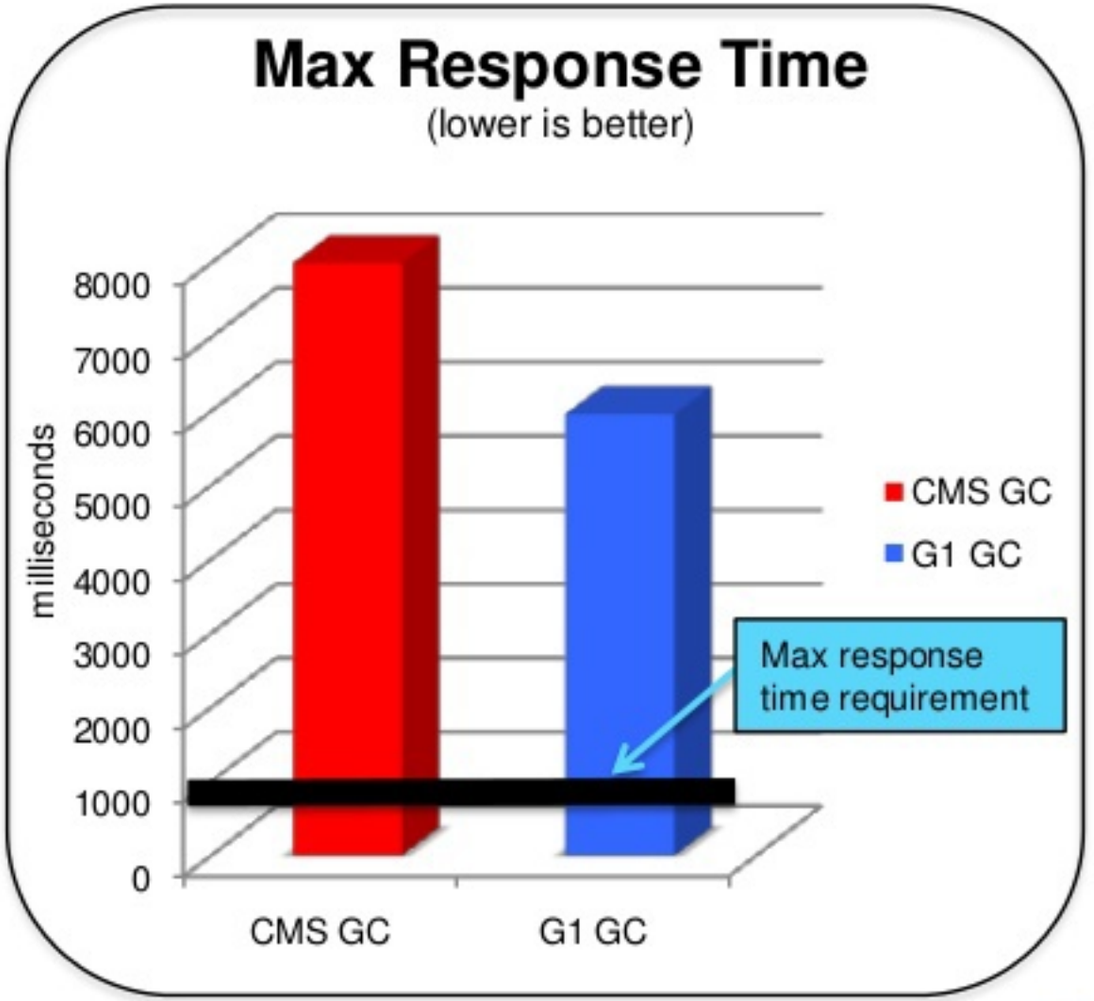
G1 2x Capacity (response times)

- Observations
  - 99th percentile at 2x capacity
    - CMS – 310 ms
    - G1 – 103 ms



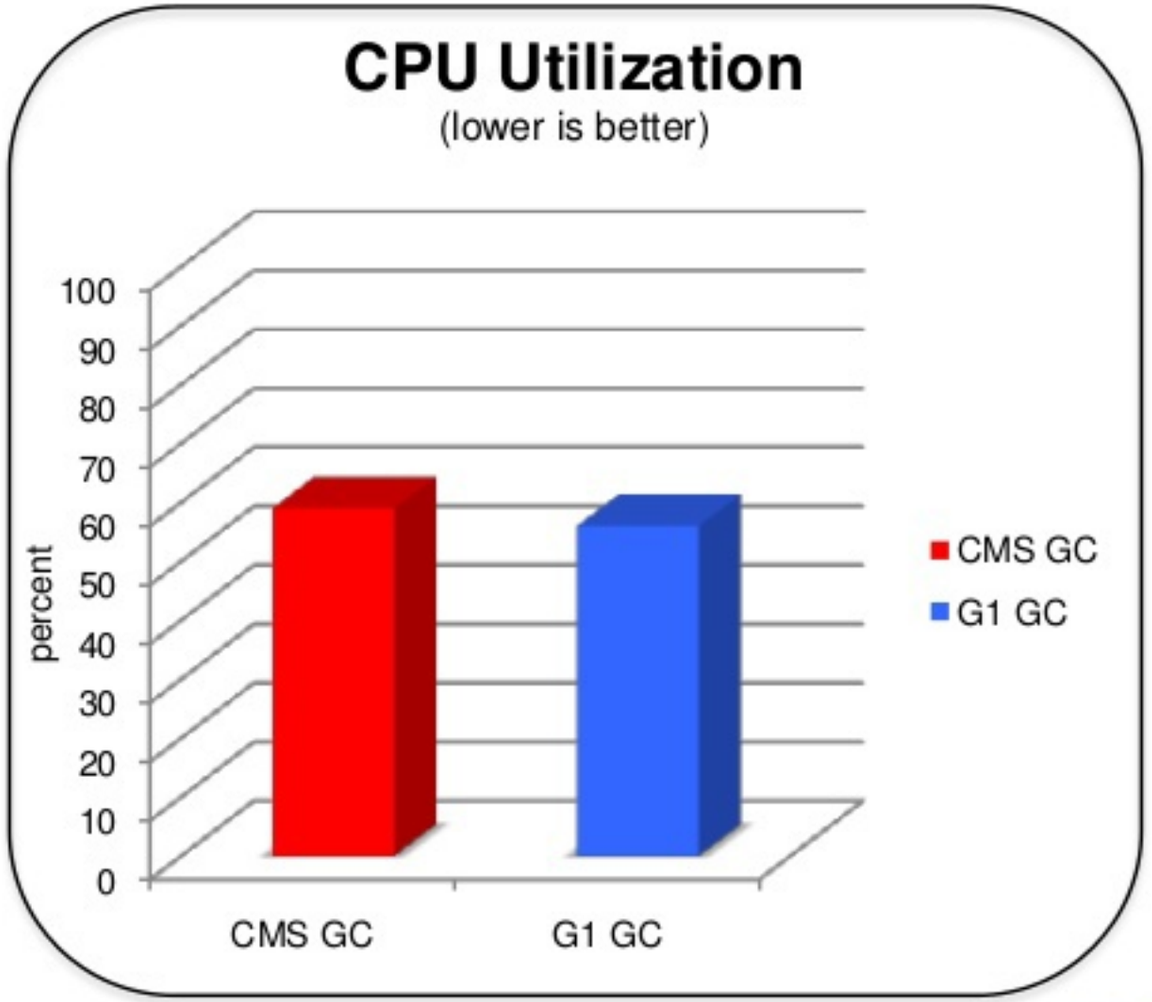
G1 2x Capacity (response times)

- Observations
  - Max at 2x capacity
    - CMS – 8000 ms
    - G1 – 5960 ms



G1 2x Capacity (CPU utilization)

- Observations
  - CPU at 2x capacity
    - CMS – 59%
    - G1 – 56%



G1 GC Analysis

- What next?
  - Go look at the GC logs!!!

G1 GC Analysis

- What did we find?
  - Excessive RSet update times

[GC pause (young), 0.9929170 secs] [Parallel Time: 988.0 ms, GC Workers: 10]

…. <other GC info removed> ….

[Update RS (ms): Min: 154.5, Avg: 237.4, Max: 980.2, Diff: 825.7, Sum: 2373.8]

Very high pause time for 200 ms pause time goal!

Very high RSet update time!

G1 GC Analysis

- What next?
  - When RSet update times are high
    - Tune -XX:G1RSetUpdatingPauseTimePercent (default is 10)
      - Target time to spend updating RSets during GC evacuation pause
      - Lower value pushes more work onto G1 Concurrent Refinement threads
      - Higher value pushes more to be done during GC (stop the world) pause
    - Tune -XX:G1ConcRefinementThreads
      - Defaults to value of -XX:ParallelGCThreads

G1 GC Analysis

- What next? (continued)
  - Test system has 12 virtual processors
  - Currently using default -XX:ParallelGCThreads=10
  - Change -XX:ParallelGCThreads=12
    - This will also set -XX:G1ConcRefinementThreads=12
  - Decrease -XX:G1RSetUpdatingPauseTimePercent=5
    - default is 10
  - Watch for change / increase in CPU usage

Updated G1 GC JVM Options

-Xms10g -Xmx10g

-XX:PermSize=64m -XX:MaxPermSize=64m

-XX:+UseG1GC -XX:MaxGCPauseMillis=200

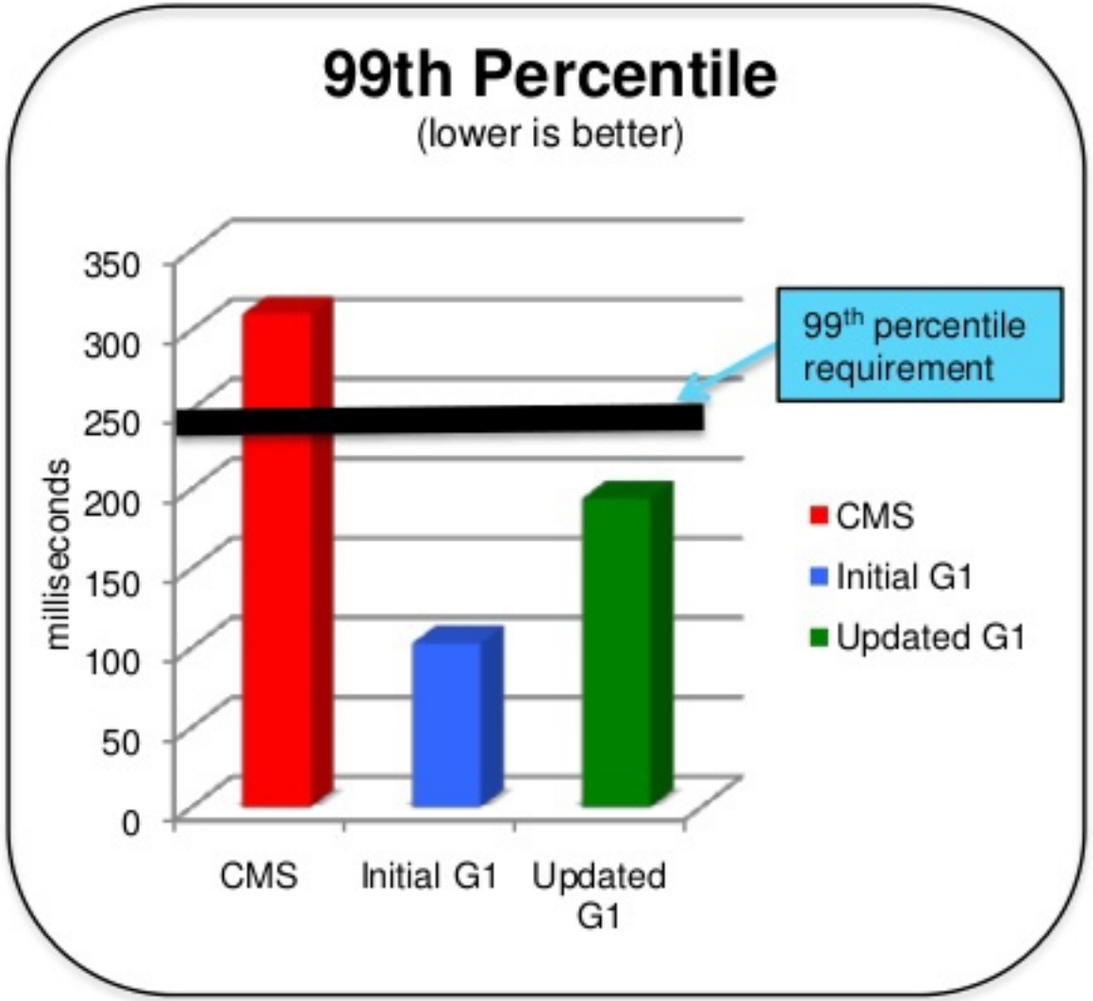
-XX:InitiatingHeapOccupancyPercent=50

-XX:ParallelGCThreads=12

-XX:G1RSetUpdatingPauseTimePercent=5

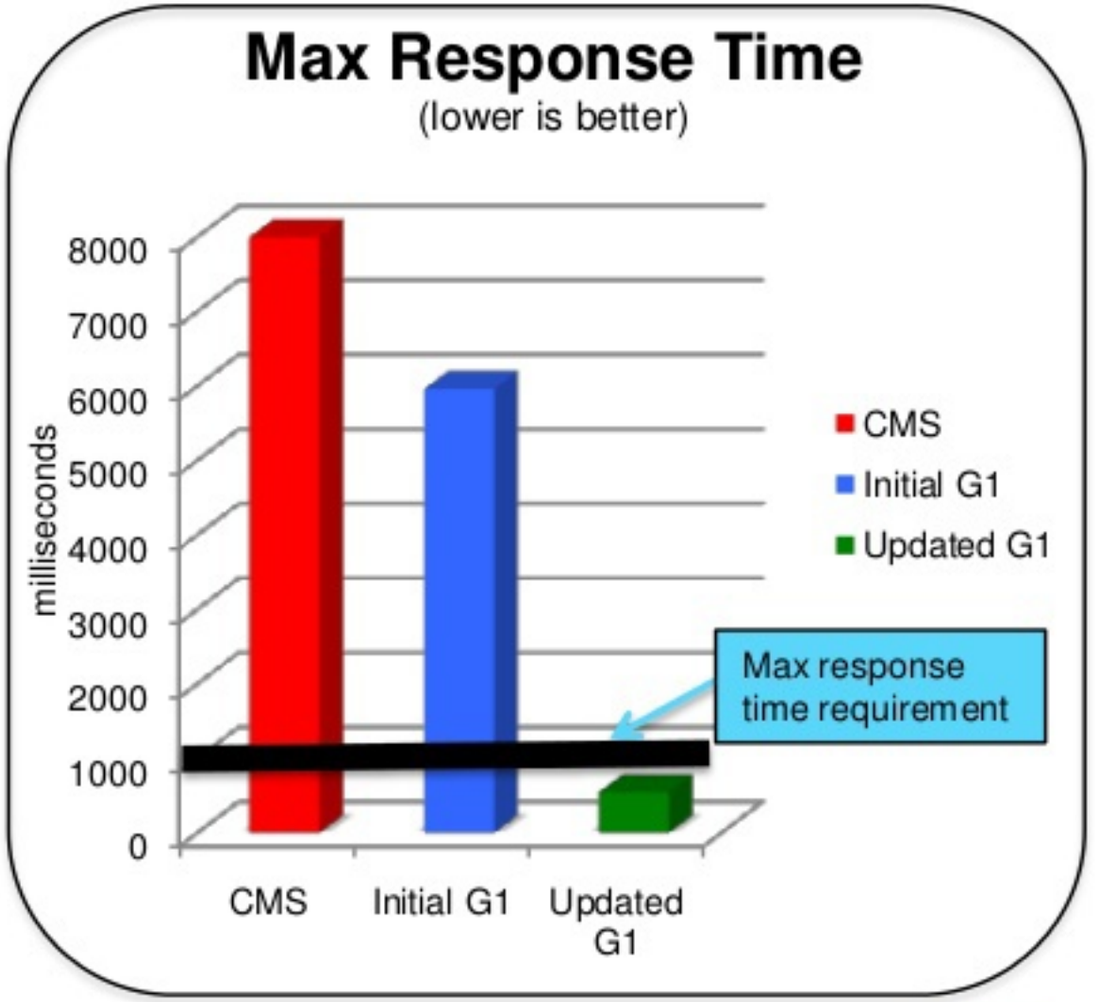
Updated G1 GC Config (response times)

- Observations at 2x capacity
  - 99th percentile
    - CMS – 310 ms
    - Initial G1 – 103 ms
    - Updated G1 - 194 ms



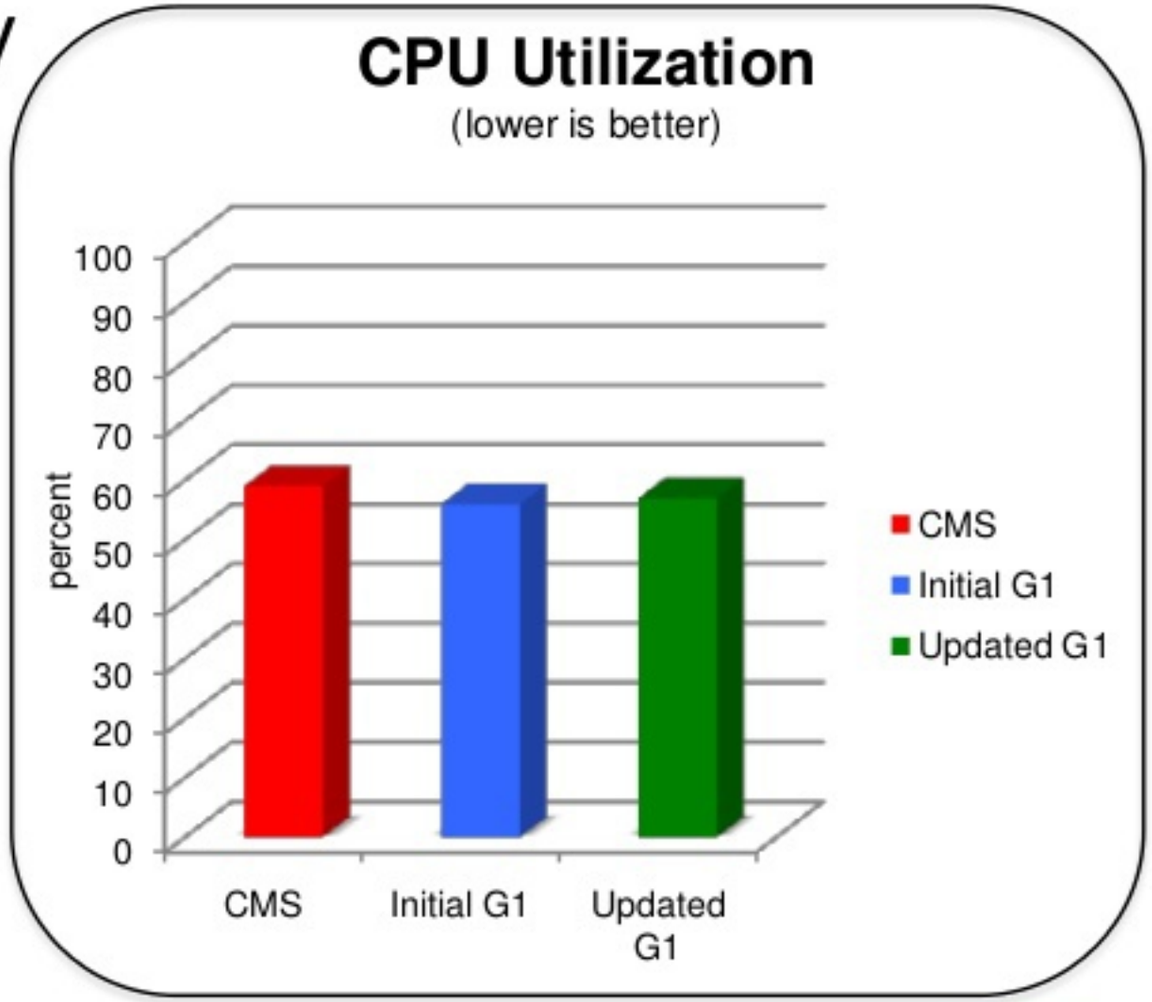
Updated G1 GC Config (response times)

- Observations at 2x capacity
  - Max response time
    - CMS – 8000 ms
    - Initial G1 – 5960 ms
    - Updated G1 – 548 ms



Updated G1 GC Config (CPU utilization)

- Observations at 2x capacity
  - CPU Utilization
    - CMS – 59%
    - Initial G1 – 56%
    - Updated G1 – 57%



Analysis

- Are we done?
  - No, go look at the raw G1 GC logs for some additional potential improvement!!!

G1 GC Analysis

- What did we find?
  - No humongous allocations, no adaptive / ergonomic issues
  - Looking at the amount of space reclaimed per marking & mixed GC cycle
    - 1 GB to 2 GB per cycle, ~ 10% - 20% of the Java heap
    - ~ 90% heap occupancy just prior to mixed GC cycle
    - Seems pretty good
      - Delicate balance between starting the marking cycle too early and wasting CPU cycles at the expense of capacity vs starting too late and experiencing to-space exhausted or to-space overflow events

Analysis

- Are we done?
  - Yes, for now ;-)
    - Requirements met
      - 99th percentile
      - Max response time
      - Both expected load and 2x expected load
    - With lower CPU utilization!

Analysis

- Are we done?
  - Yes, for now ;-)
    - Requirements met
      - 99th percentile
      - Max response time
      - Both expected load and 2x expected load
    - With lower CPU utilization!
  - Stakeholders happy!

Summary

Call to Action (i)

- Understand the “success criteria”
- Understand when it makes sense to transition and evaluate G1 GC
- Don’t over-tune!
  - Start G1 evaluation with minimal settings
    - Initial and max heap size, pause time target and when to start marking cycle
    - Don’t use young gen sizing, i.e. -Xmn, -XX:SurvivorRatio, etc

Call to Action (ii)

- Do GC analysis
  - Use tools for a high level view and identify troublesome time periods --- then look at the raw GC logs
    - Look at several GC events prior and after to see how things go into an undesirable state, and the action G1 took and the recovery

Call to Action (iii)

- Do GC analysis
  - If you see evacuation failures (to-space overflow, to-space exhausted or Full GC)
    - Enable -XX:+PrintAdaptiveSizePolicy
    - If you see humongous object allocations
      - Increase -XX:G1HeapRegionSize
    - If you see to-space exhausted, i.e. marking cycle starting too late
      - Remove young gen sizing constraints if they exist
      - Decrease -XX:InitiatingHeapOccupancyPercent, (default is 45)
      - Increase Java heap size -Xms and -Xmx

Call to Action (iv)

- Do GC analysis
  - If you see high RSet update times
    - Increase -XX:ParallelGCThreads if not at the max budget of virtual processors per JVM
      - This will increase -XX:G1ConcRefinementThreads
        - Can set independently of -XX:ParallelGCThreads
    - Decrease -XX:G1RSetUpdatingPauseTimePercent
      - Lowers time spent in RSet updating in GC evacuation pause
    - Note: May observe increase in CPU utilization

Call to Action (iv) continued …

- To observe RSet update and RSet scan information
  - Add these three JVM command line options
    - -XX:+G1SummarizeRSetStats
    - -XX:G1SummarizeRSetStatsPeriod=1
    - -XX:+UnlockDiagnosticVMOptions
  - Can use these to identify if some of the updating and coarsening of RSets is being pushed onto the mutator (application) threads

Call to Action (v)

- If time from marking cycle start until mixed GCs start take a long time
  - Increase -XX:ConcGCThreads (default ParallelGCThreads / 4)
    - Note: may observe an increase in CPU utilization

Call to Action (vi)

- If marking cycles are running too frequently
  - Look at space reclaimed in the mixed GC cycles after a marking cycle completes
  - If you're reclaiming (very) little space per marking cycle & mixed GC cycle
    - Increase -XX:InitiatingHeapOccupancyPercent
  - Note: The more frequent the marking cycles & mixed GC cycles, you may observe higher CPU utilization

Call to Action …. Last One

- Parallel Reference Processing
  - Look for high “[Ref Proc:” times or high reference processing times in remark pauses --- should be a fraction of the overall GC pause time

Call to Action …. Last One

- Parallel Reference Processing
  - Look for high "[Ref Proc:" times or high reference processing times in remark pauses --- should be a fraction of the overall GC pause time

[Other 24.8 ms]

[Choose CSet: 0.0 ms]

[Ref Proc: 16.9 ms]

Or,

[GC remark 17212.791 [GC ref-proc, 7.8638730 secs] 8.4352460 secs]

Very high reference processing times

Call to Action …. Last One

- Parallel Reference Processing
  - Look for high "[Ref Proc:" times or high reference processing times in remark pauses --- should be a fraction of the overall GC pause time

[Other 24.8 ms]

[Choose CSet: 0.0 ms]

[Ref Proc: 16.9 ms]

Or,

[GC remark 17212.791 [GC ref-proc, 7.8638730 secs] 8.4352460 secs]

Very high reference processing times

  - Enable -XX:+ParallelRefProcEnabled to address this

Additional Info

- JavaOne Sessions
  - G1 Collector: Current and Future Adaptability and Ergonomics
    - If you missed it, get a recording!
  - Tuning Low-Pause Garbage Collection (Hands on Lab)
    - If you missed it, get the materials!

Additional Info

- GC Analysis Tools
  - JClarity has some very good GC analysis tools
    - Currently adding G1 analysis
    - http://www.jclarity.com
  - We most often use awk, perl, specialized JFreeChart and spreadsheets :-O

If you can,
share observations and GC logs at
hotspot-gc-use@openjdk.java.net

Happy G1 GC Trails!

Thank You