Performance gain with G1 garbage collection algorithm in vertically scaled J2EE Infrastructure deployment

Prateek Khanna
Sr. Principal CoE Engineer – Fusion Middleware CoE

aioug
ALL INDIA ORACLE USERS GROUP
THE POWER TO CONNECT & CONQUER

Overview

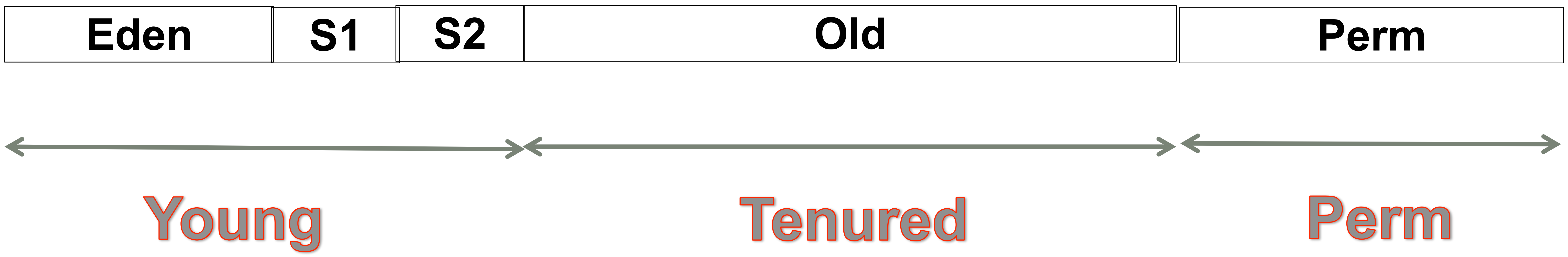
Garbage Collection

- Garbage = memory occupied by dead objects(unreachable)
- Garbage collection = Reclamation of garbage
- Invented by John McCarthy in 1958 as part of Lisp
- Available in Java, Modula3, Prolog , SmallTalk etc.

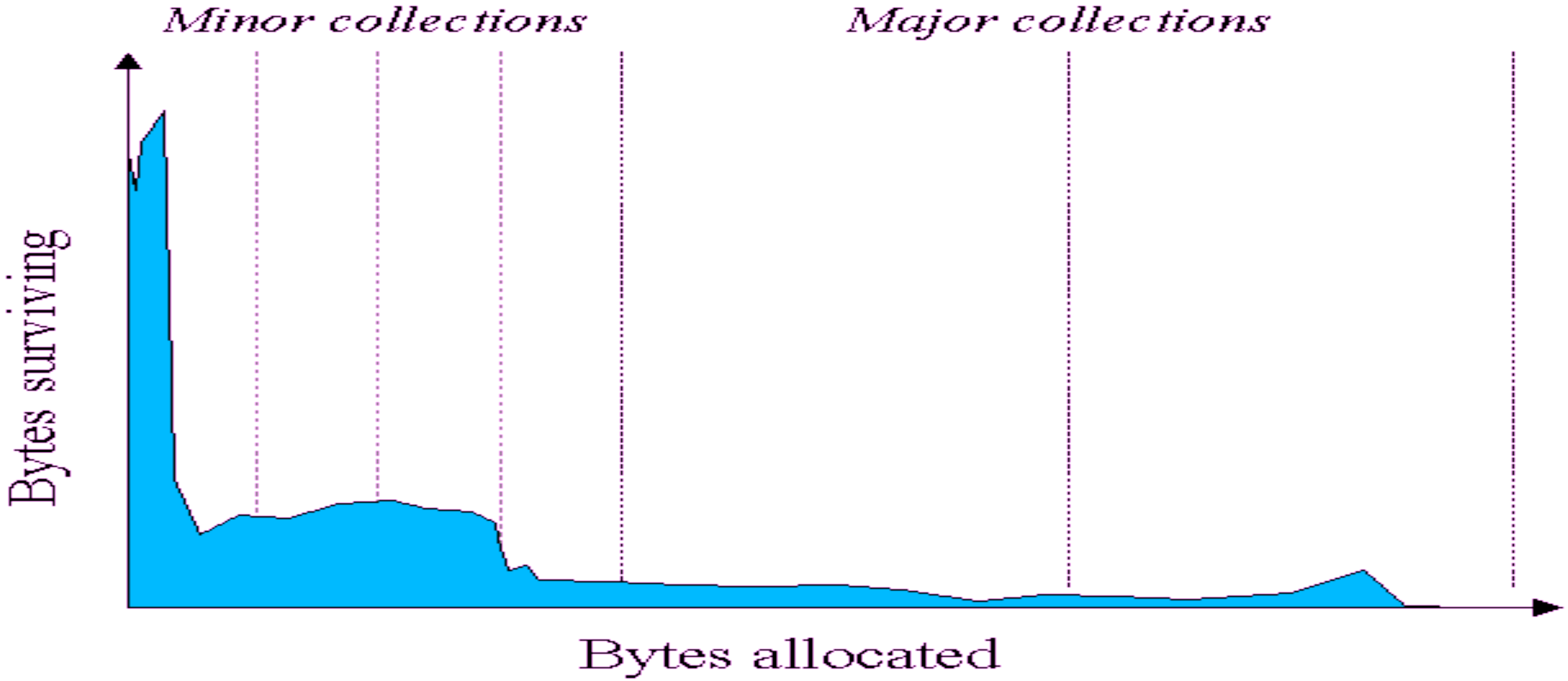
Garbage Collection

- Generational Hypothesis – Young objects are more likely to die than old objects (Infant mortality)
- Basis for generational GC algorithms which divide the heap into New and Old segments based on object age.
- Entire heap does not need to be garbage collected at every instance.

Garbage Collection



Garbage Collection



Garbage Collection - Classifications

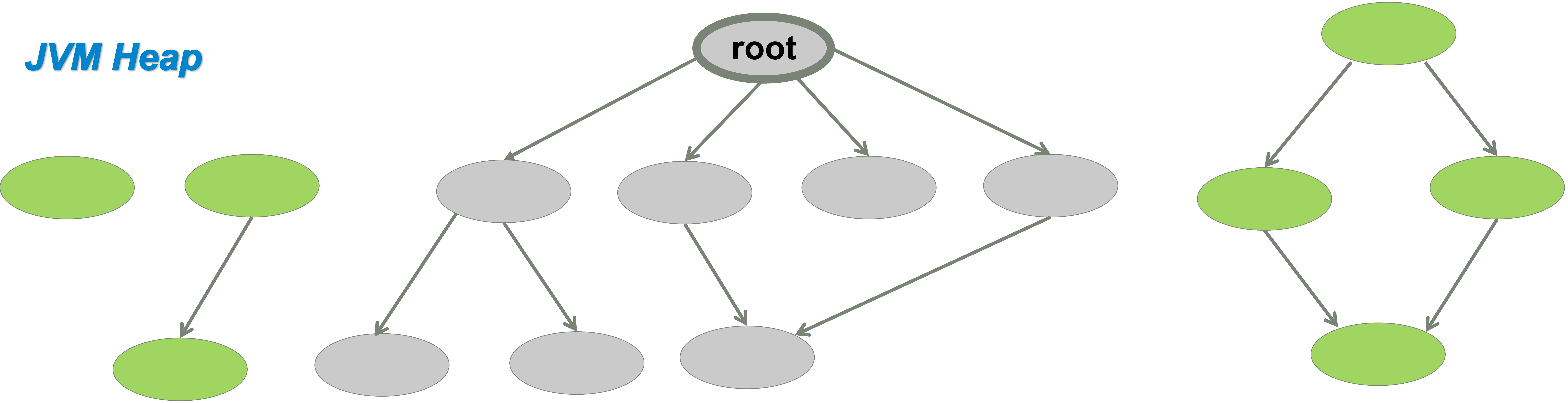
- Generational
  - Young (Minor)
  - Old (Major)

- Concurrency
  - Serial
  - Concurrent

- G1 – region oriented

Mark and Sweep

JVM Heap



Reachable object (live)

UnReachable object (dead) – Ready for GC

Concurrent Mark and Sweep

- Mark phase marks up all the unreferenced objects.
- Sweep phase removes these unreferenced objects
- The CMS collector performs much of the GC activity **concurrently** with the application execution threads
- Separate GC threads are used for this purpose.

Concurrent Mark and Sweep

- The pause times involved are shorter.
- But this is a non compacting collector.
- It leaves the heap fragmented.
- This will eventually lead to a high pause compaction cycle.

Concurrent Mark and Sweep

- The CMS GC cycle includes the following phases:
  - Initial marking phase (Serial)
  - Marking phase, Preclean phase (Concurrent)
  - Remarking phase (Parallel)
  - Sweep phase (Concurrent)

Concurrent Mark and Sweep

Concurrent Mark and Sweep - Problems

- 64 – bit environments
- Larger heap sizes (Xmx > 6 GB)
- GC pauses grow beyond acceptable levels in live enterprise environments due to fragmentation.
- Unpredictable GC pause times

Garbage First GC

- This is meant to be a long term replacement for CMS.
- Fully supported since JDK 7 update 4.
- It is a compacting collector.
- It allows specifying desired pause times.

Garbage First GC

- It attempts to meet these specified pause times with a high probability.
- The maximum pause time can be spcified using -XX:MaxGCPauseMillis.
- e.g., -XX:MaxGCPauseMillis=100 will set the request for maximum GC pause time to 100 ms.

Garbage First GC

- The JVM heap is partitioned into a set of uniformly sized heap regions.
- The default size of each region is determined based on the actual heap size. The region size varies between 1 Mb – 32 Mb
- The size can also be specified explicitly using -XX:G1HeapRegionSize=<size in Mb>.

Garbage First GC

- G1 performs a concurrent marking phase across all regions to determine the liveness of objects

- G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, hence the name **Garbage First**.

- G1 selects the number of regions to collect based on the specified pause time target.

Garbage First GC

- Concurrent mark phases:
  - Initial mark
  - Concurrent root region scan
  - Concurrent mark
  - Remark
  - Cleanup

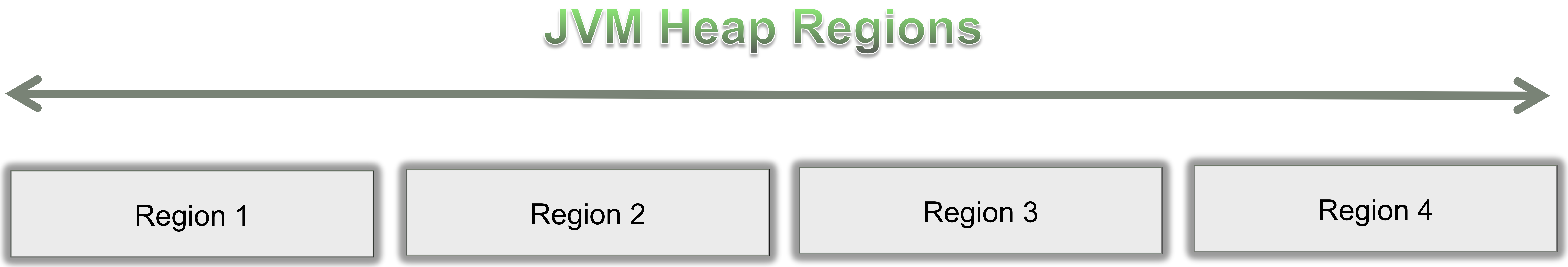
Garbage First GC

- The regions in the **collection set** are garbage collected using **evacuation**.
- G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory.
- The evacuation is performed in parallel on multi-processor environments.

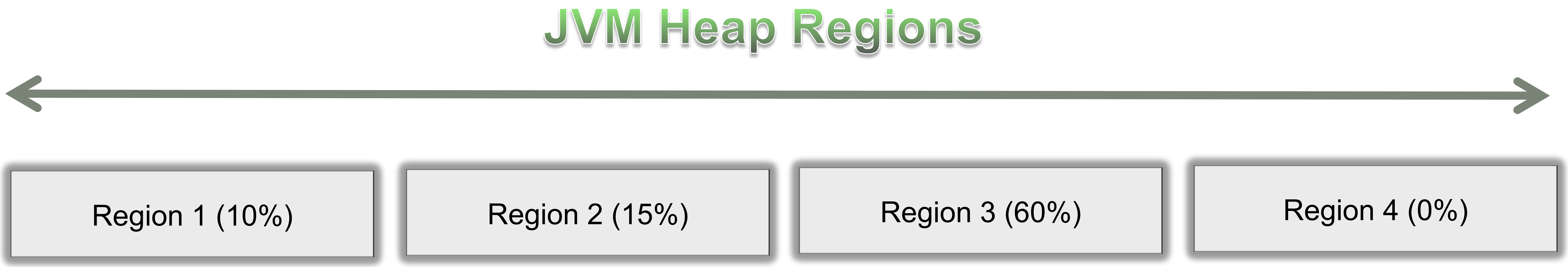
Garbage First GC

- During evacuation, all application threads are stopped.
- Parallel GC threads copy live objects from the identified **collection set** of regions to the target region.
- Application threads resume after the evacuation pause.

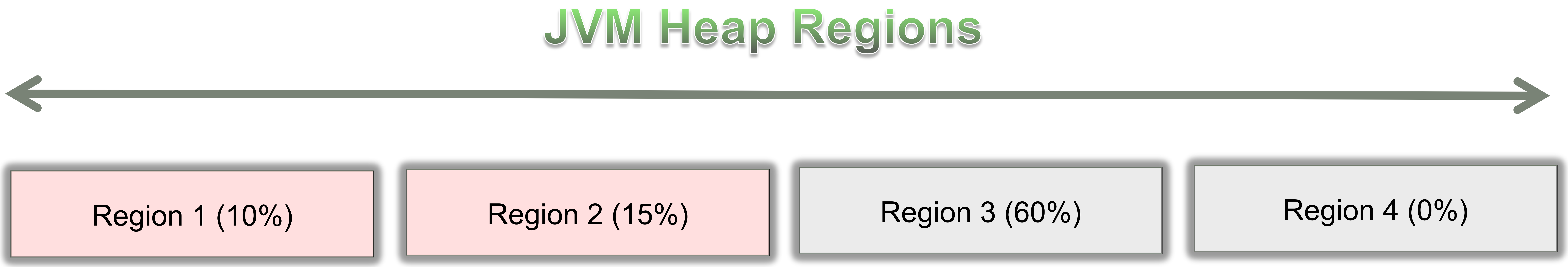
Garbage First GC



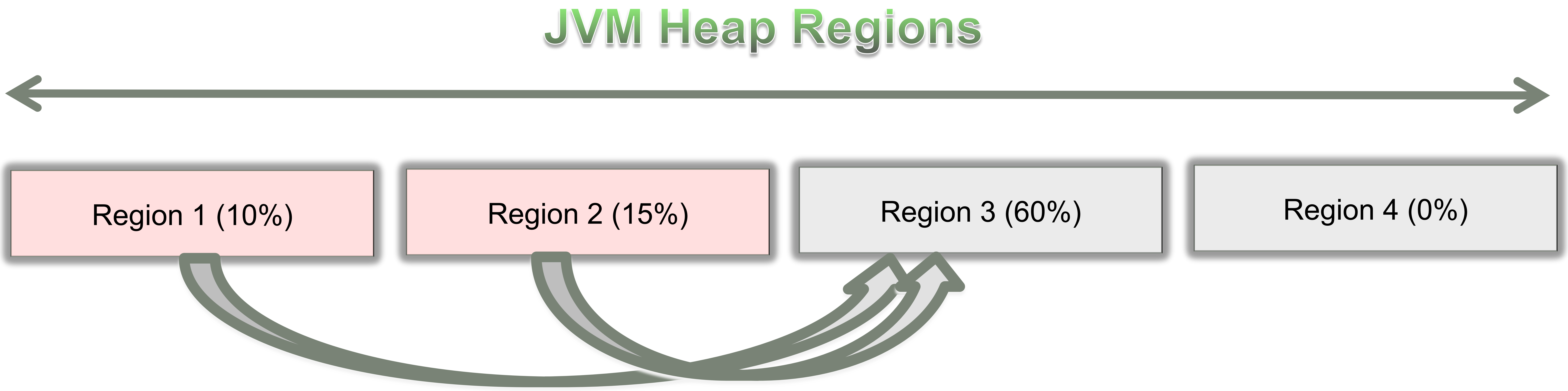
Garbage First GC – After Live Object Markup phase



Garbage First GC – Identify *Collection Set*



Garbage First GC – Evacuation

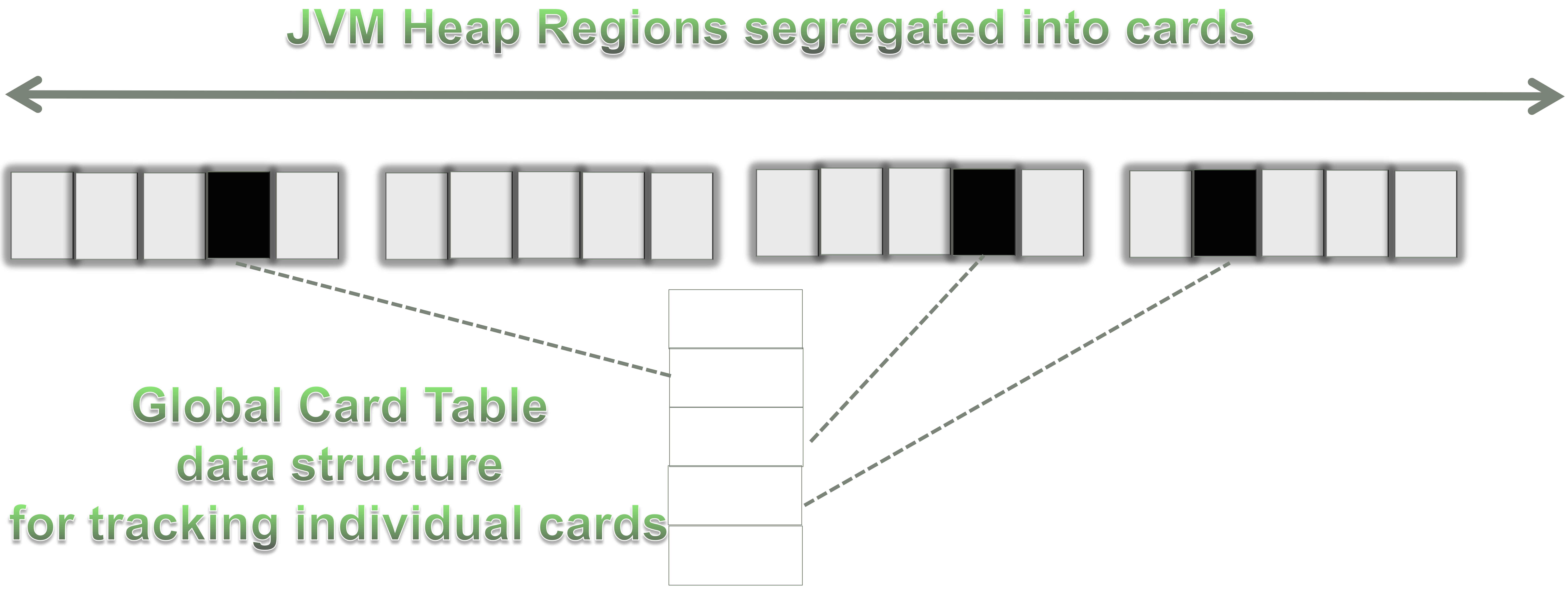


Garbage First GC – Free Regions after collection

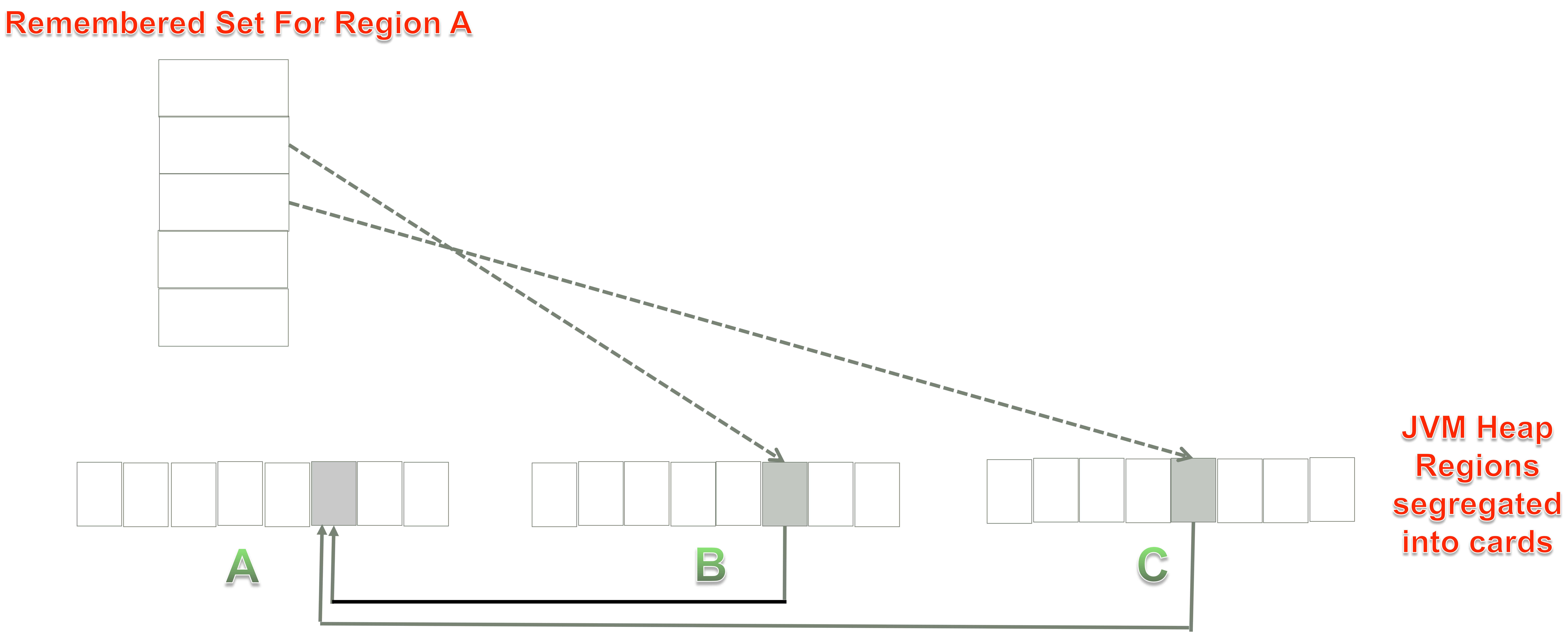
Garbage First GC

- Each region is further divided into 512 byte section called **card**.
- Each region has an associated **remembered set** in the **global card table** which contains a 1 byte entry per card
- Cards that contain pointers from other regions to this region's objects form part of the current region's logical remembered set.

Garbage First GC



Garbage First GC



G1 versus CMS

- G1 GC is best suited for runtimes with large heap sizes (> 6 GB) which require limited GC pause times (~ 0.5 sec).

- Note that it is not suitable for hard real time applications.

- CMS on the other hand does not scale well with larger heap sizes leading to longer pause times.

G1 versus CMS

G1 is intended to provide a good fit for large-scale server applications which have big size live heap data and a multi-threaded implementation running on multi-core platforms.

GC Performance Results

GC Performance Metrics Terminology

- Throughput : Percentage of runtime spent on actual work (not GC)
- GC pause times : min/max/cumulative
- GC pause interval : min/max
- Heap size : used/ max available

GC Performance Results

- An experimental evaluation was carried out of the behavior of G1 and CMS across various runtime scenarios.

- The DaCapo 9.12 suite is an open source Java benchmarking tool consisting of a set of real world applications with nontrivial memory loads. We have used tests from this suite for determining the GC behavioral characteristics under different scenarios including simulating application server work profile.

GC Performance Results

- Verbose GC logging was turned on for capturing GC related information.

- Sample GC log information obtained during runtime are depicted in the subsequent slides.

GC Performance Results

- Sample CMS Log entries:

0.689: [Full GC (System) 0.692: [**CMS**: 0K->300K(63872K), 0.0473210 secs] 4882K->300K(83008K), [CMS Perm : 3820K->3819K(21248K)], 0.0478150 secs] [Times: user=0.01 sys=0.03, real=0.04 secs]

9.681: [GC 9.681: [**ParNew**: 17024K->1298K(19136K), 0.0230910 secs] 17324K->1598K(83008K), 0.0231860 secs] [Times: user=0.04 sys=0.02, real=0.03 secs]

GC Performance Results

- Sample G1 Log entries:

```
0.450: [GC pause (young), 0.00399600 secs]
           [Parallel Time:   3.9 ms]
              [GC Worker Start Time (ms):  450.4  450.4]
              [Update RS (ms):  0.0  0.0
               Avg:   0.0, Min:   0.0, Max:   0.0]
                 [Processed Buffers : 0 3
                  Sum: 3, Avg: 1, Min: 0, Max: 3]
              [Ext Root Scanning (ms):  1.0  0.9
               Avg:   1.0, Min:   0.9, Max:   1.0]
              [Mark Stack Scanning (ms):  0.0  0.0
               Avg:   0.0, Min:   0.0, Max:   0.0]
              [Scan RS (ms):  0.0  0.0
               Avg:   0.0, Min:   0.0, Max:   0.0]
              [Object Copy (ms):  2.9  2.9
               Avg:   2.9, Min:   2.9, Max:   2.9]
              [Termination (ms):  0.0  0.0
               Avg:   0.0, Min:   0.0, Max:   0.0]
                 [Termination Attempts : 1 1
                  Sum: 2, Avg: 1, Min: 1, Max: 1]
              [GC Worker End Time (ms):  454.2  454.2]
              [Other:   0.0 ms]
           [Clear CT:   0.0 ms]
           [Other:   0.1 ms]
              [Choose CSet:   0.0 ms]
           [ 4096K->857K(32M)]
         [Times: user=0.01 sys=0.00, real=0.00 secs]
```

GC Performance Results

Test environment:

- *Processor:* Intel(R) Xeon(R) CPU X5675 @ 3.07GHz
- *Operating System:*   Linux 2.6.18-53.el5 #1 SMP Wed Oct 10 16:34:19 EDT 2007 x86_64 x86_64 x86_64 GNU/Linux
- *JVM :* Java(TM) SE Runtime Environment (build 1.7.0_06-b24) / Java HotSpot(TM) 64-Bit Server VM (build 23.2-b09, mixed mode)
- *JVM Heap Size :* 7168 MB

GC Performance Results – Test A

Performance Comparison of G1 with CMS using a set of broad based DaCapo 9.12 tests:

DaCapo benchmark tests:

avrora batik eclipse h2 luindex lusearch pmd sunflow xalan

No. of test iterations: 5

GC Performance Results – Test A

DaCapo benchmark test details:

- **Avrora** - simulates a number of programs run on a grid of AVR microcontrollers
- **Batik** - produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
- **Eclipse** - executes some of the (non-gui) jdt performance tests for the Eclipse IDE
- **H2** - executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark

GC Performance Results – Test A

- **Luindex** - Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
- **Lusearch** - Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
- **Pmd** - analyzes a set of Java classes for a range of source code problems
- **Sunflow** - renders a set of images using ray tracing
- **Xalan** - transforms XML documents into HTML

GC Performance Results – Test A

	CMS	*G1*
Total Pause Time	111.51 s	59.07 s
Throughput	85.29%	93%
Total Heap Usage	411.87M/7159.7M(5.8%)	4798.67M/7168M(66.95%)
Total GC Pauses	1153	80
Min/Max GC Pause	0.027 s/3.385 s	0.035 s/5.857s
Min/Max pause interval	0.054s/19.101s	0.893s/88.411s

GC Performance Results – Test B

- Validation of Test A using a set of DaCapo 9.12 tests over extended run times(20 iterations):

- *DaCapo benchmark tests:*

avrora batik eclipse h2 luindex lusearch pmd sunflow xalan

- *No. of test iterations: 20*

GC Performance Results – Test B

	CMS	*G1*
Total Pause Time	431.02 s	194.47s
Throughput	82.16%	91.86%
Total Heap Usage	416.6M/7159.7M(5.8%)	5230M/7168M(73%)
Total GC Pauses	4475	240
Min/Max GC Pause	0.026 s/3.951 s	0.096/5.734s
Min/Max pause interval	0.042 s/13.92 s	1.067s/95.822s

GC Performance Results – Test C

- Testing the GC behavior on a vertically scaled J2EE Infrastructure setup (Tomcat) over multiple iterations.
- *DaCapo benchmark tests:* tomcat
- This test uses the Apache Tomcat servlet container to run a set of sample web applications and validates it.

GC Performance Results – Test C

No. of iterations	*20*		*50*		*100*	
	CMS	*G1*	*CMS*	*G1*	*CMS*	*G1*
Total Pause Time	12.25 s	9.78 s	28.52s	26.95s	65.61s	54.58s
Throughput	93.1%	94.44 %	92.96%	94.71%	91.71%	93.74%
Total Heap Usage	80.8M/ 7159.7M(1.1%)	595M/ 7168M(8.3%)	80.7M/ 7159.7M(1.1%)	596M/ 7168M(8.3%)	80.8M/ 7159.7M(1.1%)	597M/ 7168M(8.3%)
Total GC Pauses	181	20	451	50	901	100
Min/Max GC Pause	0.031 s/0.401 s	0.327s/1.068s	0.031s/0.641s	0.346s/1.246s	0.031s/0.596s	0.335s/3.664s
Min/Max pause interval	0.213 s/3.932 s	5.826s/29.686s	0.314s/9.334s	5.373s/63.24s	0.224s/4.25s	6.356s/86.425s

Conclusion

- In general, G1 shows lower number of GC pauses and lower cumulative pause time than CMS. The same behavior has been independently verified for vertically scaled J2EE deployment as well.

- The difference in behavior is more pronounced over larger load conditions.

Conclusion

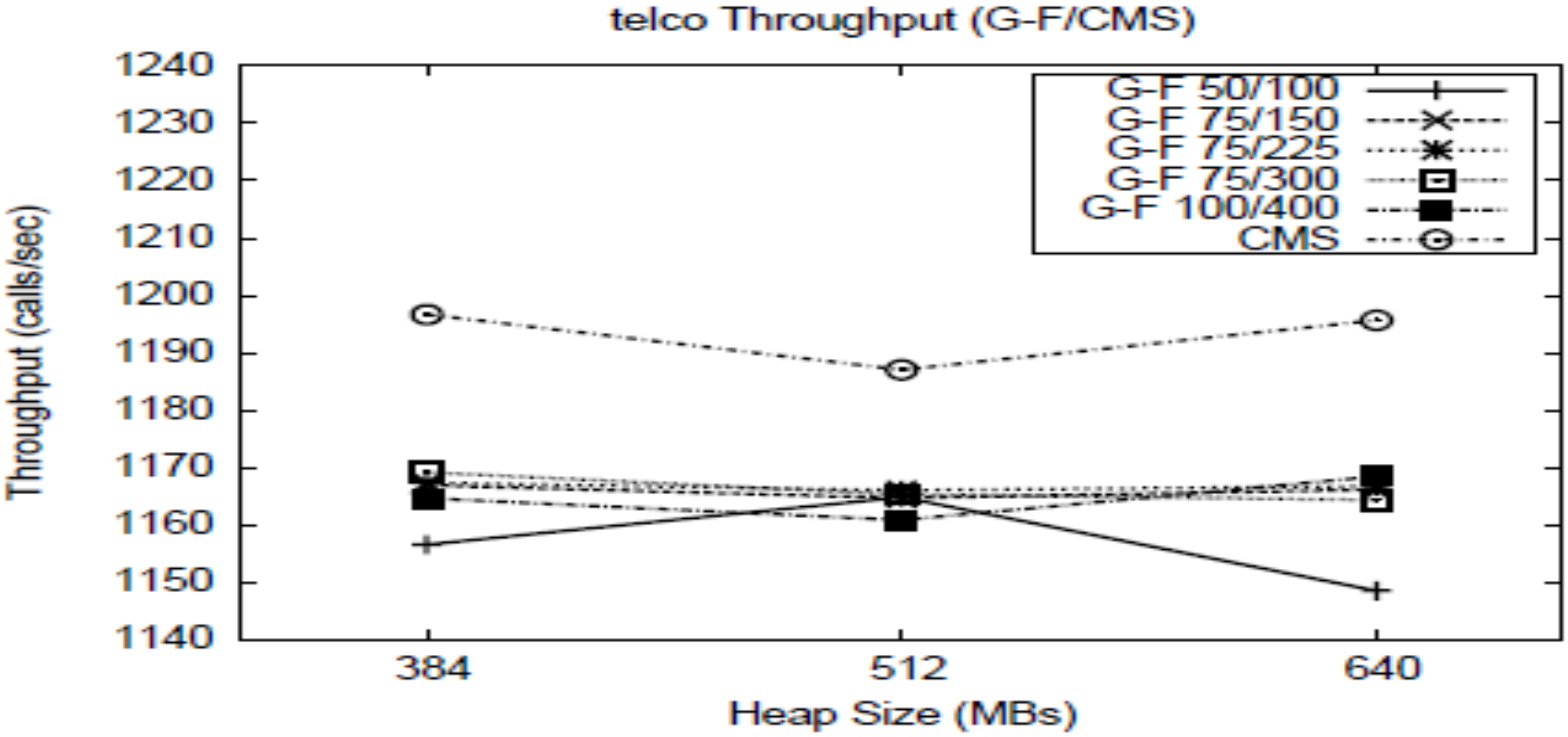
- G1 GC is found to make use of a higher proportion of available heap during runtime.
- Number of Young GC pauses is found to decrease markedly with increase in available heap for G1 garbage collector.

References

- http://labs.oracle.com/jtech/pubs/04-g1-paper-ismm.pdf
- http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html
- http://sigops.org/sosp/sosp11/workshops/plos/07-gidra.pdf
- http://dacapobench.org/

Appendix

3rd party test results



aioug

ALL INDIA ORACLE USERS GROUP

THE POWER TO CONNECT & CONQUER