G1: One Garbage Collector To Rule Them All

Many articles describe how a poorly tuned garbage collector can bring an application's Service Level Agreement (SLA) commitments to its knees. For example, an unpredictably protracted garbage collection pause can easily exceed the response-time requirements of an otherwise performant application. Moreover, the irregularity increases when you have a non-compacting Garbage Collector (GC) such as Concurrent Mark and Sweep (CMS) that tries to reclaim its fragmented heap with a serial (single-threaded) full garbage collection that is stop-the-world (STW).

Let us now expand on the above paragraph: Suppose an allocation failure in the young generation triggers a young collection, leading to promotions to the old generation. Further, suppose that the fragmented old generation has insufficient space for the newly promoted objects. Such conditions would trigger a full garbage collection cycle, which will perform compaction of the heap.

With CMS GC, the full collection is serial and STW, hence your application threads are stopped for the entire duration while the heap space is reclaimed and then compacted. The duration for the STW pause depends on your heap size and the surviving objects.

Alternatively, even if you do have parallel (multi-threaded) compaction to combat fragmentation, you still end up with a full garbage collection (that involves all the generations of the Java heap), when it might have been sufficient to just reclaim some of the free space from the old generation.

This is a common scenario with Parallel Old GC. With Parallel Old, the reclamation of old generation is with a parallel STW full garbage collection pause. This full garbage collection is not incremental; it is one big STW pause and does not interleave with the application execution.

Note: You can read more about HotSpot GCs h
ere.

With the above information, we would like to consider one solution in the form of the "Garbage First” (G1) collector, HotSpot's latest GC (introduced in JDK7 update 4).

G1 GC is an incremental parallel compacting GC that provides more predictable pause times compared to CMS GC and Parallel Old GC. By introducing a parallel, concurrent and multi-phased marking cycle, G1 GC can work with much larger heaps while providing reasonable worst-case pause times. The basic idea with G1 GC is to set your heap ranges (using
`-Xms`
for min heap size and
`-Xmx`
for the max size) and a realistic (soft real time) pause time goal (using
`-XX:MaxGCPauseMillis)`
and then let the GC do its job.

With the introduction of G1 GC, HotSpot moves away from its conventional GC layout where a contiguous Java heap splits into (contiguous) young and old generations. In G1 GC, HotSpot introduces the concept of “regions”. A single large contiguous Java heap space divides into multiple fixed-sized heap regions. A list of “free” regions maintains these regions. As the need arises, the free regions are assigned to either the young or the old generation. These regions can span from 1MB to 32MB in size depending on your total Java heap size. The goal is to have around 2048 regions for the total heap. Once a region frees up, it goes back to the "free" regions list. The principle of G1 GC is to reclaim the Java heap as much as possible (while trying its best to meet the pause time goal) by collecting the regions with the least amount of live data i.e. the ones with most garbage, first; hence the name
Garbage First
.

Fig. 1: Conventional GC Layout

One thing to note is that for G1 GC, neither the young nor the old generation has to be contiguous. This is a handy feature since the sizing of the generation is now more dynamic.

Adaptive sized GC algorithms like the Parallel Old GC, end up reserving the extra space that may be required by each generation so that they can fit in their contiguous space constraint. In case of CMS, a full garbage collection is required to resize the Java heap and the generations.

In contrast, G1 GC uses logical generations (a collection of non-contiguous regions of the young generation and a remainder in the old generation), so there is not much wastage in space or time.

To be sure, the G1 GC algorithm does utilize some of HotSpot's basic concepts. For example, the concepts of allocation, copying to survivor space and promotion to old generation are similar to previous HotSpot GC implementations. Eden regions and survivor regions still make up the young generation. Most allocations happen in eden except for "humongous" allocations. (Note: For G1 GC, objects that span more than half a region size are considered "Humongous objects" and are directly allocated into "humongous" regions out of the old generation.) G1 GC selects an adaptive young generation size based on your pause time goal. The young generation can range anywhere from the preset min to the preset max sizes, that are a function of the Java heap size. When eden reaches capacity, a "young garbage collection", also known as an "evacuation pause", will occur. This is a STW pause that copies (evacuates) the live objects from the regions that make up the eden, to the 'to-space' survivor regions.

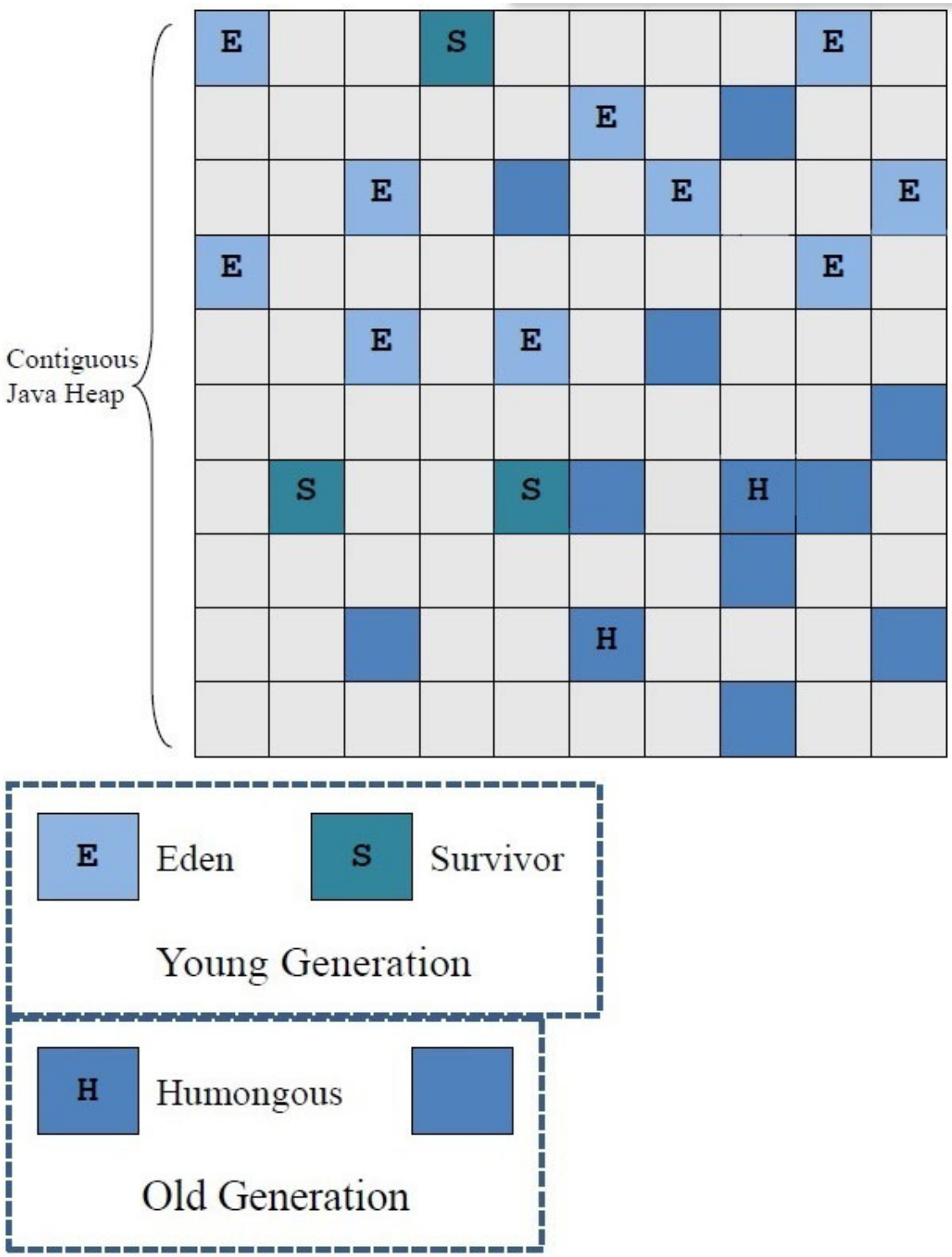

Fig. 2: Garbage First GC Layout

In addition, live objects from the 'from-space' survivor regions will be either copied to the 'to-space' survivor regions or, based on the object's age and the 'tenuring threshold', will be promoted to region(s) from the old generation space.

Every young collection involves parallel worker time and sequential/serial time. To explain this further, I will use a log output from the latest Java 7 update release, which at the time of this publication is 7u25. (We also have an Early Access (EA) for 7u40. Please feel free to try out the EA bundles for your platform. With 7u40 EA, you may see a difference in the log format, but the basic premise remains the same.)

The following command line options generated the GC log output thereafter –

java -Xmx1G -Xms1G -XX:+UseG1GC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps GCTestBench

Note: I went with the default pause time goal of 200ms.

```
0.189: [GC pause (young), 0.00080776 secs]
   [Parallel Time: 0.4 ms]
      [GC Worker Start (ms): 188.7 188.7 188.8 188.8
       Avg: 188.8, Min: 188.7, Max: 188.8, Diff: 0.1]
      [Ext Root Scanning (ms): 0.2 0.2 0.2 0.1
       Avg: 0.2, Min: 0.1, Max: 0.2, Diff: 0.1]
      [Update RS (ms): 0.0 0.0 0.0 0.0
       Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]
         [Processed Buffers : 0 0 0 1
          Sum: 1, Avg: 0, Min: 0, Max: 1, Diff: 1]
      [Scan RS (ms): 0.0 0.0 0.0 0.0
       Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]
      [Object Copy (ms): 0.2 0.2 0.1 0.2
       Avg: 0.2, Min: 0.1, Max: 0.2, Diff: 0.0]
      [Termination (ms): 0.0 0.0 0.0 0.0
       Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]
         [Termination Attempts : 1 2 1 2
          Sum: 6, Avg: 1, Min: 1, Max: 2, Diff: 1]
      [GC Worker End (ms): 189.1 189.1 189.1 189.1
       Avg: 189.1, Min: 189.1, Max: 189.1, Diff: 0.0]
      [GC Worker (ms): 0.4 0.4 0.3 0.3
       Avg: 0.4, Min: 0.3, Max: 0.4, Diff: 0.1]
      [GC Worker Other (ms): 0.0 0.0 0.1 0.1
       Avg: 0.1, Min: 0.0, Max: 0.1, Diff: 0.1]
   [Clear CT: 0.2 ms]
   [Other: 0.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.2 ms]
      [Ref Enq: 0.0 ms]
      [Free CSet: 0.0 ms]
```

The indentation demarcates the parallel and the **sequential** work groups. The parallel worker time is further split into -

1. External Root Scanning:
   The time spent by the parallel GC worker threads in scanning the external roots such as registers, thread stacks, etc that point into the Collection Set.
2. Update Remembered Sets (RSets):
   RSets aid G1 GC in tracking reference that point into a region. The time shown here is the amount of time the parallel worker threads spent in updating the RSets.
3. Processed Buffers:
   The count shows how many 'Update Buffers' were processed by the worker threads.
4. Scan RSets:
   The time spent in Scanning the RSets for references into a region. This time will depend on the "coarseness" of the RSet data structures.
5. Object Copy:
   During every young collection, the GC copies all live data from the eden and 'from-space' survivor, either to the regions in the 'to-space' survivor or to the old generation regions. The amount of time it takes the worker threads to complete this task is listed here.
6. Termination:
   After completing their particular work (e.g. object scan and copy), each worker thread enters its 'termination protocol'. Prior to terminating, the worker thread looks for work from the other threads to steal and terminates when there is none. The time listed here indicates the time spent by the worker threads offering to terminate.
7. Parallel worker 'Other' time:
   Time spent by the worker threads that was not accounted in any of the parallel activities listed above.

The **sequential work** (which could be parallelized, individually) is divided into -

1. **Clear CT**: Time spent by the GC worker threads in clearing the Card Table of RSet scanning meta-data.
2. And a few others clubbed under the '**Other**' time, comprised of:

- **Choose Collection Set (CSet)**: A garbage collection cycle collects the set of regions in the CSet. The collection pause collects/evacuates all the live data in a particular CSet. The time listed here is the time spent in finalizing the set of regions added to the CSet.
- **Reference Processing**: The time spent in processing the deferred references (soft, weak, final and phantom) from the prior garbage collection phases.
- **Reference En-queuing**: The time spent in placing the references on to the pending list.
- **Free CSet**: Time spent in freeing the just collected set of regions. This includes the time spent in freeing their RSets as well.

I have just skimmed the surface with respect to many things like the RSets, its coarsening, the update buffers, the CSet, and in the next few paragraphs there will be a few more things like the Snapshot-At-The-Beginning (SATB) algorithm and barriers, etc. However, in-order to learn more about them, we would have to "deep dive" into the internals of G1 GC, an interesting topic that is outside the scope of this article.

Now that we understand how the young collections start filling up the old generation, we need to introduce (and understand) the concept of a 'marking threshold'. When the occupancy of the total heap crosses this threshold, G1 GC will trigger a multi-phased concurrent marking cycle. The command line option that sets the threshold is `-XX:InitiatingHeapOccupancyPercent` and it defaults to 45 percent of the total Java heap size. G1 GC uses a marking algorithm called Snapshot-At-The-Beginning (SATB) that takes a logical snapshot of the set of live objects in the heap at the 'beginning' of the marking cycle. This algorithm uses a pre-write barrier to record and mark the objects that are a part of the logical snapshot. Now let us spend some time discussing the individual phases of the multi-phased concurrent marking and first a look at the output from the GC log:

```
0.078: [GC pause (young) (initial-mark), 0.00262460 secs]
   [Parallel Time: 2.3 ms]
      [GC Worker Start (ms): 78.1 78.2 78.2 78.2
       Avg: 78.2, Min: 78.1, Max: 78.2, Diff: 0.1]
      [Ext Root Scanning (ms): 0.2 0.1 0.2 0.1
       Avg: 0.2, Min: 0.1, Max: 0.2, Diff: 0.1]
      [Update RS (ms): 0.2 0.2 0.2 0.2
       Avg: 0.2, Min: 0.2, Max: 0.2, Diff: 0.0]
         [Processed Buffers : 2 3 2 2
          Sum: 9, Avg: 2, Min: 2, Max: 3, Diff: 1]
      [Scan RS (ms): 0.0 0.0 0.0 0.0
       Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]
      [Object Copy (ms): 1.8 1.8 1.8 1.8
       Avg: 1.8, Min: 1.8, Max: 1.8, Diff: 0.0]
```

```
      [Termination (ms): 0.0 0.0 0.0 0.0
       Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]
         [Termination Attempts : 1 1 1 1
          Sum: 4, Avg: 1, Min: 1, Max: 1, Diff: 0]
      [GC Worker End (ms): 80.4 80.4 80.4 80.4
       Avg: 80.4, Min: 80.4, Max: 80.4, Diff: 0.0]
      [GC Worker (ms): 2.2 2.2 2.2 2.2
       Avg: 2.2, Min: 2.2, Max: 2.2, Diff: 0.1]
      [GC Worker Other (ms): 0.0 0.1 0.1 0.1
       Avg: 0.1, Min: 0.0, Max: 0.1, Diff: 0.1]
   [Clear CT: 0.2 ms]
   [Other: 0.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.1 ms]
      [Ref Enq: 0.0 ms]
      [Free CSet: 0.0 ms]
   [Eden: 3072K(5120K)->0B(5120K) Survivors: 1024K->1024K Heap: 16M(32M)->16M(32M)]
  [Times: user=0.06 sys=0.00, real=0.00 secs]
   0.081: [GC concurrent-root-region-scan-start]
   0.082: [GC concurrent-root-region-scan-end, 0.0009122]
   0.082: [GC concurrent-mark-start]
<snip>
[Zero or more embedded young garbage collections are possible here,
          but removed for brevity.]

   0.094: [GC concurrent-mark-end, 0.0115579 sec]
   0.094: [GC remark 0.094: [GC ref-proc, 0.0000033 secs], 0.0004374 secs]
          [Times: user=0.00 sys=0.00, real=0.00 secs]
   0.094: [
GC cleanup 22M->10M(32M), 0.0003031 secs
]
          [
Times: user=0.00 sys=0.00, real=0.00 secs
]
   0.095: [
GC concurrent-cleanup-start
]
   0.095: [
GC concurrent-cleanup-end, 0.0000350
]
```

In addition, here are the details:

- The Initial Mark Phase
  – G1 GC marks the roots during the initial-mark phase. This is what the first line of output above is telling us. The initial-mark phase is piggy backed (done at the same time) on a normal (STW) young garbage collection. Hence, the output is similar to what you see during a young evacuation pause.

- The Root Region Scanning Phase
  – During this phase, G1 GC scans survivor regions of the initial mark phase for references into the old generation and marks the referenced objects. This phase runs concurrently (not STW) with the application. It is important that this phase complete before the next young garbage collection happens.
- The Concurrent Marking Phase
  – During this phase, G1 GC looks for reachable (live) objects across the entire Java heap. This phase happens concurrently with the application and a young garbage collection can interrupt the concurrent marking phase (
  shown above
).
- The Remark Phase
  – The remark phase helps the completion of marking. During this STW phase, G1 GC drains any remaining SATB buffers and traces any as-yet unvisited live objects. G1 GC also does reference processing during the remark phase.
- The Cleanup Phase
  – This is the final phase of the multi-phase marking cycle. It is
  partly STW
  when G1 GC does live-ness accounting (to identify completely free regions and mixed garbage collection candidate regions) and when G1 GC scrubs the RSets. It is
  partly concurrent
  when G1 GC resets and returns the empty regions to the free list.

Once G1 GC successfully completes the concurrent marking cycle, it has the information that it needs to start the old generation collection. Up until now, the collection of the old regions was not possible since G1 GC did not have any marking information associated with those regions. A collection that facilitates the compaction and evacuation of old generation is appropriately called a 'mixed' collection since G1 GC not only collects the eden and the survivor regions, but also (optionally) adds old regions to the mix. Let us now discuss some details that are important to understand a mixed collection.

A mixed collection can (and usually does) happen over multiple mixed garbage collection cycles. When a sufficient number of old regions are collected, G1 GC reverts to performing the young garbage collections until the next marking cycle completes. A number of flags listed and defined here control the exact number of old regions added to the CSets:

`-XX:G1MixedGCLiveThresholdPercent`
: The occupancy threshold of live objects in the old region to be included in the mixed collection.

`-XX:G1HeapWastePercent`
: The threshold of garbage that you can tolerate in the heap.

`-XX:G1MixedGCCountTarget`
: The target number of mixed garbage collections within which the regions with at most G1MixedGCLiveThresholdPercent live data should be collected.

`-XX:G1OldCSetRegionThresholdPercent`
: A limit on the max number of old regions that can be collected during a mixed collection.

Let us look at a mixed collection cycle output from a G1 GC log:

```
1.269: [GC pause (mixed), 0.00373874 secs]
   [Parallel Time: 3.0 ms]
      [GC Worker Start (ms): 1268.9 1268.9 1268.9 1268.9
    Avg: 1268.9, Min: 1268.9, Max: 1268.9, Diff: 0.0]
   [Ext Root Scanning (ms): 0.2 0.2 0.2 0.1
    Avg: 0.2, Min: 0.1, Max: 0.2, Diff: 0.1]
   [Update RS (ms): 0.0 0.0 0.0 0.0
    Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]
      [Processed Buffers : 0 0 0 1
       Sum: 1, Avg: 0, Min: 0, Max: 1, Diff: 1]
   [Scan RS (ms): 0.1 0.0 0.0 0.1
    Avg: 0.1, Min: 0.0, Max: 0.1, Diff: 0.1]
   [Object Copy (ms): 2.6 2.7 2.7 2.6
    Avg: 2.7, Min: 2.6, Max: 2.7, Diff: 0.1]
   [Termination (ms): 0.1 0.1 0.0 0.1
    Avg: 0.0, Min: 0.0, Max: 0.1, Diff: 0.1]
      [Termination Attempts : 2 1 2 2
       Sum: 7, Avg: 1, Min: 1, Max: 2, Diff: 1]
   [GC Worker End (ms): 1271.9 1271.9 1271.9 1271.9
    Avg: 1271.9, Min: 1271.9, Max: 1271.9, Diff: 0.0]
   [GC Worker (ms): 3.0 3.0 3.0 2.9
    Avg: 3.0, Min: 2.9, Max: 3.0, Diff: 0.0]
   [GC Worker Other (ms): 0.1 0.1 0.1 0.1
    Avg: 0.1, Min: 0.1, Max: 0.1, Diff: 0.0]
  [Clear CT: 0.1 ms]
  [Other: 0.6 ms]
   [Choose CSet: 0.0 ms]
   [Ref Proc: 0.1 ms]
   [Ref Enq: 0.0 ms]
   [Free CSet: 0.3 ms]
```

In summary, G1 improves upon its predecessor GCs by introducing the concept of regions that make up a logical generation. The regions help provide finer granularity for an incremental collection of the old generation. G1 does most of its reclamation through copying of the live data, thus achieving compaction. This is definitely a step up from in-space de-allocation without compaction, which lends the old generation looking like Swiss cheese!
☺

The first level of reclamation happens during the Cleanup phase (of the multi-phased marking cycle) when the *completely* free (i.e. full of garbage) regions are reclaimed and returned to the free list. The next level happens during the incremental mixed garbage collections. If all else fails, the entire Java heap is collected. This is the well-known fail-safe full garbage collection.

All of the above makes the reclamation of the old generation a lot easier and in a way tiered.

I hope this article helped in painting a basic picture of the differences and the makeup of G1 GC. Thank you for tuning in!

Editor's note: Please stay tuned for part 2, coming in September 2013, where Monica will discuss some advanced topics and offer some advice about how to use these metrics to tune your application performance.

About the Author

Monica Beckwith
is a Java Performance Consultant. Her past experiences include working with Oracle/Sun and AMD; optimizing the JVM for server class systems. Monica was voted a Rock Star speaker @JavaOne 2013 and was the performance lead for Garbage First Garbage Collector (G1 GC). You can follow Monica on twitter @mon_beck