Tips for Tuning the Garbage First Garbage Collector

This is the second article in a two-part series on the G1 garbage collector. You can find part one on InfoQ July 15, 2013: G1: One Garbage Collector To Rule Them All.

Before we can understand how to tune the Garbage First garbage collector (G1 GC), we must first understand the key concepts that define G1. In this article, I will first introduce the concept and then talk about how to tune G1 (where appropriate) with respect to that concept.

Remembered Sets

Recall from the previous article: Remembered Sets (RSets in short) are per-region entries that aid G1 GC in tracking outside references that point into a heap region. So now, instead of scanning the entire heap for references into a region, G1 just needs to scan its RSet.

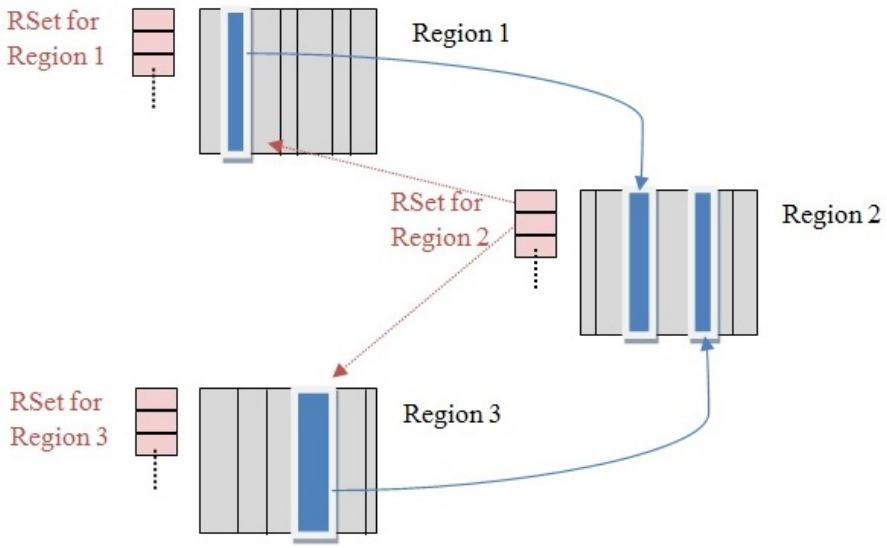

Figure 1: Remembered Sets

Let us look at an example. In figure 1 above, we show three regions (in gray): Region 1, Region 2 and Region 3 and their associated RSets (in pink) that represent a set of cards. Both Region 1 and Region 3 happen to be referencing objects in Region 2. Therefore, the RSet for Region 2 tracks the two references to Region 2, the "owning region".

There are two concepts that help maintain RSets:

1. Post-write barriers
2. Concurrent refinement threads

The barrier code steps in after a write (hence the name "post-write barrier") and helps track cross-region updates. Update log buffers are responsible for logging the cards that contain the updated reference field. Once these buffers are full, they are retired. Concurrent refinement threads process these full buffers.

Note that the concurrent refinement threads help maintain RSets by updating them concurrently (when the application is also running). The deployment of the concurrent refinement threads is tiered, where initially only a small number of threads are deployed and then more are eventually added depending on the amount of filled update buffers to be processed. The max number of concurrent refinement threads can be controlled by **-XX:G1ConcRefinementThreads** or even -**XX:ParallelGCThreads**. If the concurrent refinement threads cannot keep up with the amount of filled buffers, then the mutator threads own and handle the processing of the buffers - usually something that you should strive to avoid.

OK, so coming back to RSets - There is one RSet per region. There are three levels of granularity for RSets - Sparse, Fine and Coarse. A Per-Region-Table (PRT) is an abstraction that houses the granularity level for RSet. A sparse PRT is a hash table that contains card indices. G1 GC internally maintains these cards. The card may contain references from the region that spans the address associated with the card to the owning region. A fine-grain PRT is an open hash table where each entry represents a region with a reference into the owning region. The card indices, within the region, are held in a bitmap. When reaching the max capacity of the fine-grain PRT, a corresponding coarse-grained bit is set in the coarse grain bitmap and the corresponding entry is deleted from the fine grain PRT. A coarse bitmap has one bit for each region. A set bit in the coarse grain map means that the associated region may contain references to the owning region.

A Collection Set (CSet) is a set of regions to be collected during a garbage collection. For a young collection, the CSet only includes young regions, for a mixed collection, the CSet includes young and old regions.

If the CSet includes many regions with coarsened RSets (Note that "coarsening of RSets" is defined as the transitioning of RSets through different levels of granularity), then you will see an increase in scan time for the RSets. These scan times are represented in the GC pause as "Scan RS (ms)" in the GC logs. If the Scan RS times seem high relative to the overall GC pause time, or they appear high for your application, then please look for the text string "**Did xyz coarsenings**" in your GC log output when using the diagnostic option **-XX:+G1SummarizeRSetStats** (you can also specify the reporting frequency period (in number of GCs) by setting **-XX:G1SummarizeRSetStatsPeriod=period**).

If you recall from the previous article, an "Update RS (ms)" in the GC pause output shows the time spent in updating RSets, and the "Processed Buffers" show the count of the update buffers process during the GC pause. If you spot issues in those in your GC logs then use the above-mentioned options to "deep-dive" into the issue even further.

Those options can also help identify potential issues with the update log buffers and the concurrent refinement threads.

A sample output of **-XX:+G1SummarizeRSetStats** with the period set to one **-XX:G1SummarizeRSetStatsPeriod=1:**
Concurrent RS processed 784125 cards

Of 4870 completed buffers:

4870 **(100.0%) by concurrent RS threads.**

0 (**0.0%) by mutator threads**.

Concurrent RS threads times (s)

0.64 0.30 0.26 0.18 0.17 0.16 0.17 0.15 0.15 0.12 0.13 0.08 0.13 0.13 0.12 0.13 0.12 0.11 0.12 0.11 0.12 0.13 0.11

Concurrent sampling threads times (s)

0.00

Total heap region rem set sizes = 199140K. Max = 661K.

Static structures = 660K, free_lists = 15052K.

1009422114 occupied cards represented.

Max size region = 313:(O)[0x000000054e400000,0x000000054e800000,0x000000054e800000], size = 662K, occupied = 1214K.

Did 2759 coarsenings.

The above output shows the count of processed cards and completed buffers. It also shows that the **concurrent refinement threads did 100% of the work** and **mutator threads did none**(which as we said is a good sign!). It then lists the concurrent refinement thread times for each thread involved in the work.

The segment in brown shows the cumulative stats since the start of the HotSpot VM. The cumulative stats include the total RSet sizes and max RSet size, total number of occupied cards and max region size information. It also shows the total number of coarsenings done since the start of the VM.

At this point, I think it is safe to introduce another option flag **-XX:G1RSetUpdatingPauseTimePercent=10**. This flag sets a percent target amount (defaults to 10 percent of the pause time goal) that G1 GC should spend in updating RSets during a GC evacuation pause. You can increase or decrease the percent value, so as to spend more or less (respectively) time in updating the RSets during the stop-the-world (STW) GC pause and let the concurrent refinement threads deal with the update buffers accordingly.

Keep in mind that by decreasing the percent value, you are pushing off the work to the concurrent refinement threads; hence, you will see an increase in concurrent work.

Reference Processing

G1 GC processes references during an evacuation pause and during a remark pause (a part of the multi-phased concurrent marking).

During an evacuation pause, the reference objects are discovered during the object scanning and copying phase and are processed after that. In the GC log, you can see the reference processing (Ref proc) time clubbed under the sequential work group called "Other" -

```
[Other: 0.2 ms]
  [Choose CSet: 0.0 ms]
  [Ref Proc: 0.2 ms]
  [Ref Enq: 0.0 ms]
  [Free CSet: 0.0 ms]
```

Note: References with dead referents are added to the pending list and that time is shown in the GC log as reference enqueing time (Ref Enq).

During a remark pause, the discovery happens during the earlier phase of concurrent marking. (Note: Both are a part of the multi-phase concurrent marking cycle. Please refer to the previous article for more information.) The remark phase deals with the processing of the discovered references. In the GC log, you can see the reference processing (GC ref-proc) time shown in the GC remark section -

```
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]
```

If you see high times during reference processing then please turn on parallel reference processing by enabling the following option on the command line **-XX:+ParallelRefProcEnabled**.

Evacuation Failure

You might have come across some of the terms "evacuation failure", "to-space exhausted", "to-space overflow" or maybe something like "promotion failure" during your GC travels. These terms all refer to the same thing, and the concept is similar in G1 GC as well. When there are no more free regions to promote to the old generation or to copy to the survivor space, and the heap cannot expand since it is already at its maximum, an evacuation failure occurs.

For G1 GC, an evacuation failure is very expensive -

1. For successfully copied objects, G1 needs to update the references and the regions have to be tenured.
2. For unsuccessfully copied objects, G1 will self-forward them and tenure the regions in place.

So what should you do when you encounter an evacuation failure in your G1 GC logs? -

1. Find out if the failures are a side effect of over-tuning - Get a simple baseline with min and max heap and a realistic pause time goal: Remove any additional heap sizing such as **-Xmn, -XX:NewSize, -XX:MaxNewSize, -XX:SurvivorRatio**, etc. Use only **-Xms, -Xmx** and a pause time goal **-XX:MaxGCPauseMillis**.
2. If the problem persists even with the baseline run and if humongous allocations (see next section below) are not the issue - the corrective action is to increase your Java heap size, if you can, of course
3. If increasing the heap size is not an option and if you notice that the marking cycle is not starting early enough for G1 GC to be able to reclaim the old generation then drop your **-XX:InitiatingHeapOccupancyPercent**. The default for this is 45% of your total Java heap. Dropping the value will help start the marking cycle earlier. Conversely, if the marking cycle is starting early and not reclaiming much, you should increase the threshold above the default value to make sure that you are accommodating for the live data set for your application.
4. If concurrent marking cycles are starting on time, but are taking a lot of time to finish; and hence are delaying the mixed garbage collection cycles which will eventually lead to an evacuation failure since old generation is not timely reclaimed; increase the number of concurrent marking threads using the command line option: **-XX:ConcGCThreads**.
5. If "to-space" survivor is the issue, then increase the **-XX:G1ReservePercent**. The default is 10% of the Java heap. G1 GC creates a false ceiling and reserves the memory, in case there is a need for more "to-space". Of course, G1 GC caps it off at 50%, since we do not want the end-user to set it to a very large value.

To help explain the cause of evacuation failure, I want to introduce a very useful option: **-XX:+PrintAdaptiveSizePolicy**. This option will provide many ergonomic details that are purposefully kept out of the **-XX:+PrintGCDetails** option.

Let us look at a log snippet with **-XX:+PrintAdaptiveSizePolicy** enabled:

```
6062.121: [GC pause (G1 Evacuation Pause) (mixed) 6062.121: [G1Ergonomics

6062.121: [G1Ergonomics (CSet Construction) add young regions to CSet, ed

6062.122: [G1Ergonomics (CSet Construction) finish adding old regions to

6062.281: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: req

6062.281: [G1Ergonomics (Heap Sizing) expand the heap, requested expansio

6062.281: [G1Ergonomics (Heap Sizing) did not expand the heap, reason: he
```

```
6062.902: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: rec

6062.902: [G1Ergonomics (Concurrent Cycles) do not request concurrent cyc

6062.902: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate

(to-space exhausted), 0.7805160 secs]
```

The above snippet carries a lot of information - first off let me highlight a few things from the set of command line options that were used for the above GC log: -server **-Xms12g -Xmx12g** -XX:+UseG1GC **-XX:NewSize=4g -XX:MaxNewSize=5g**

The switches in **red** show that the user has restricted the nursery in the 4G to 5G range and thus restricting the adaptability of G1 GC. If G1 needs to drop nursery to a smaller value, it cannot; if G1 needs to increase the nursery spaces, beyond its distribution, it cannot!

This is evident from the heap utilization information printed at the end of this evacuation pause:

```
[Eden: 3648.0M(3648.0M)->0.0B(3696.0M) Survivors: 448.0M->400.0M Heap: 11
```

After the cycle, G1 has to adhere to 4096M as the minimum nursery (-XX:NewSize=4g), out of which, based on G1's calculations, 3696M should be for Eden space and 400M for the survivor space. However, post-collection the data in the Java heap is already at 9537.9M. So, G1 ran out of "to-space"! The next two evacuation pauses also result in evacuation failures with the following heap information:

Next mixed evacuation pause 1:

```
   [Eden: 2736.0M(3696.0M)->0.0B(4096.0M) Survivors: 400.0M->0.0B Heap: 1
```

Next mixed evacuation pause 2:

```
   [Eden: 0.0B(4096.0M)->0.0B(4096.0M) Survivors: 0.0B->0.0B Heap: 12.0G(
```

Finally leading to a full GC -

```
6086.564: [Full GC (Allocation Failure) 11G->3795M(12G), 15.0980440 secs]

   [Eden: 0.0B(4096.0M)->0.0B(4096.0M) Survivors: 0.0B->0.0B Heap: 12.0G(
```

The Full GC could have been avoided by letting the nursery / young gen shrink to the default minimum (5% of the total Java heap). As you may be able to tell the old generation was big enough to accommodate the live data set (LDS) of 3795M. However, LDS coupled with the explicitly set minimum young generation of 4Gs, pushed the occupancy to above 7891M. Since the marking threshold was at its default value of 45% of the heap (i.e. around 5529M), the marking cycle started earlier and reclaimed very little during the mixed collections. The heap occupancy kept increasing and another marking cycle started, but by the time, the marking cycle was done and the mixed GCs kicked in, the occupancy was already at **11.3Gs (as seen in the first heap utilization information).** This collection also encountered evacuation failures. Therefore, this issue falls in the over-tuning and “starting marking cycle too early” categories.

Humongous Allocations

One last thing that I would like to cover in this article is a concept that may be new to many end-users - humongous objects (H-objs) and G1 GC's handling of H-objs.

So, why do we need a different allocation path for H-objs? -

For G1 GC, objects are considered humongous if they span 50% or more of G1's region size. A humongous allocation needs a contiguous set of regions. As you can imagine, if G1 would allocate H-objs in the young generation and if they would stay alive for a long enough time, then there would be a lot of unnecessary and expensive (remember H-objs need contiguous regions) copying of these H-objs to survivor space and then eventually these H-objs will get promoted to the old generation. Hence, to avoid this overhead, H-objs are directly allocated out of the old generation and then categorized, or mapped as humongous regions.

By allocating H-objs directly in the old generation, G1 avoids including them in any of the evacuation pauses, and thus they are never moved. During a full garbage collection cycle, G1 GC compacts around the live H-objs. Outside of a full GC, dead H-objs are reclaimed during the cleanup phase of the multi-phased concurrent marking cycle. In other words, H-objs are collected either during the cleanup phase, or they are collected during a full GC.

Before allocating H-obj, G1 GC will check if the initiating heap occupancy percent, the marking threshold, will be crossed because of the allocation. If it will, then G1 GC will initiate a G1 concurrent marking cycle. This is done in this manner since we want to avoid evacuation failures and full garbage collection cycles as much as possible. As a result we check as early as possible so as to give the G1 concurrent cycle as much time as possible to complete before there are no more available regions for live object evacuations.

With G1 GC, the basic premise with H-objs is the hope that there are not too many of those and that they are long lived. However, since G1 GC's region size is dependent on your minimum heap size, it could happen that based on the assigned region size, your "normal" allocation may look humongous to G1 GC. This would then lead to lots of H-obj allocations taking regions from old generation, which would eventually lead to an evacuation failure since G1 would not be able to keep up with those humongous allocations.

Now you may be thinking how to find out if humongous allocations are leading to evacuation failures. - Here, once again, -XX:+PrintAdaptiveSizePolicy will come to your rescue.

In your GC log, you will see something like this:

```
1361.680: [G1Ergonomics (Concurrent Cycles) request concurrent cycle ini
```

Therefore, now you can tell that a **concurrent cycle was requested** since there was a humongous **allocation request for 4194320 bytes**.

This information is helpful, since you not only can tell how many humongous allocations were made by your application (and whether they were excessive or not), but you can also tell the size of the allocation. Moreover, if you deem that there were excessive humongous allocations, all you have to do is increase the G1 region size to fit the H-objs as regular ones. Therefore, for example, the allocation size is just above 4M bytes. Hence, to make this allocation a regular allocation, we need a region size of 16M. So, the recommendation here was to set that explicitly on the command line: -XX:G1HeapRegionSize=16M

Note: Recall from my last article, G1 regions can span from 1MB to 32MB in powers of two, and the allocation request is slightly over 4 MB. Hence, an 8MB region size is not quite large enough to avoid the humongous allocations. We need to go to the next power of 2, 16 MB.

OK. I think I have covered most of the important issues and G1 GC concepts here. Once again, thanks for tuning in!

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