Java / Technical Details /
Technical Article

Garbage First Garbage Collector Tuning

by Monica Beckwith
Published August 2013

Learn about how to adapt and tune the G1 GC for evaluation, analysis and performance.

The Garbage First Garbage Collector (G1 GC) is the low-pause, server-style generational garbage collector for Java HotSpot VM. The G1 GC uses concurrent and parallel phases to achieve its target pause time and to maintain good throughput. When G1 GC determines that a garbage collection is necessary, it collects the regions with the least live data first (garbage first).

A garbage collector (GC) is a memory management tool. The G1 GC achieves automatic memory management through the following operations:

- Allocating objects to a young generation and promoting aged objects into an old generation.

- Finding live objects in the old generation through a concurrent (parallel) marking phase. The Java HotSpot VM triggers the marking phase when the total Java heap occupancy exceeds the default threshold.
- Recovering free memory by compacting live objects through parallel copying.

Here, we look at how to adapt and tune the G1 GC for evaluation, analysis and performance—we assume a basic understanding of Java garbage collection.

The G1 GC is a regionalized and generational garbage collector, which means that the Java object heap (heap) is divided into a number of equally sized regions. Upon startup, the Java Virtual Machine (JVM) sets the region size. The region sizes can vary from 1 MB to 32 MB depending on the heap size. The goal is to have no more than 2048 regions. The eden, survivor, and old generations are logical sets of these regions and are not contiguous.

The G1 GC has a pause time-target that it tries to meet (soft real time). During young collections, the G1 GC adjusts its young generation (eden and survivor sizes) to meet the soft real-time target. During mixed collections, the G1 GC adjusts the number of old regions that are collected based on a target number of mixed garbage collections, the percentage of live objects in each region of the heap, and the overall acceptable heap waste percentage.

The G1 GC reduces heap fragmentation by incremental parallel copying of live objects from one or more sets of regions (called Collection Set (CSet)) into different new region(s) to achieve compaction. The goal is to reclaim as much heap space as possible, starting with those regions that contain the most reclaimable space, while attempting to not exceed the pause time goal (garbage first).

The G1 GC uses independent Remembered Sets (RSets) to track references into regions. Independent RSets enable parallel and independent collection of regions because only a region's RSet must be scanned for references into that region, instead of the whole heap. The G1 GC uses a post-write barrier to record changes to the heap and update the RSets.

Garbage Collection Phases

Apart from evacuation pauses (described below) that compose the stop-the-world (STW) young and mixed garbage collections, the G1 GC also has parallel, concurrent, and multiphase marking cycles. G1 GC uses the Snapshot-At-The-Beginning (SATB) algorithm, which takes a snapshot of the set of live objects in the heap at the start of a marking cycle. The set of live objects is composed of the live

objects in the snapshot, and the objects allocated since the start of the marking cycle. The G1 GC marking algorithm uses a pre-write barrier to record and mark objects that are part of the logical snapshot.

Young Garbage Collections

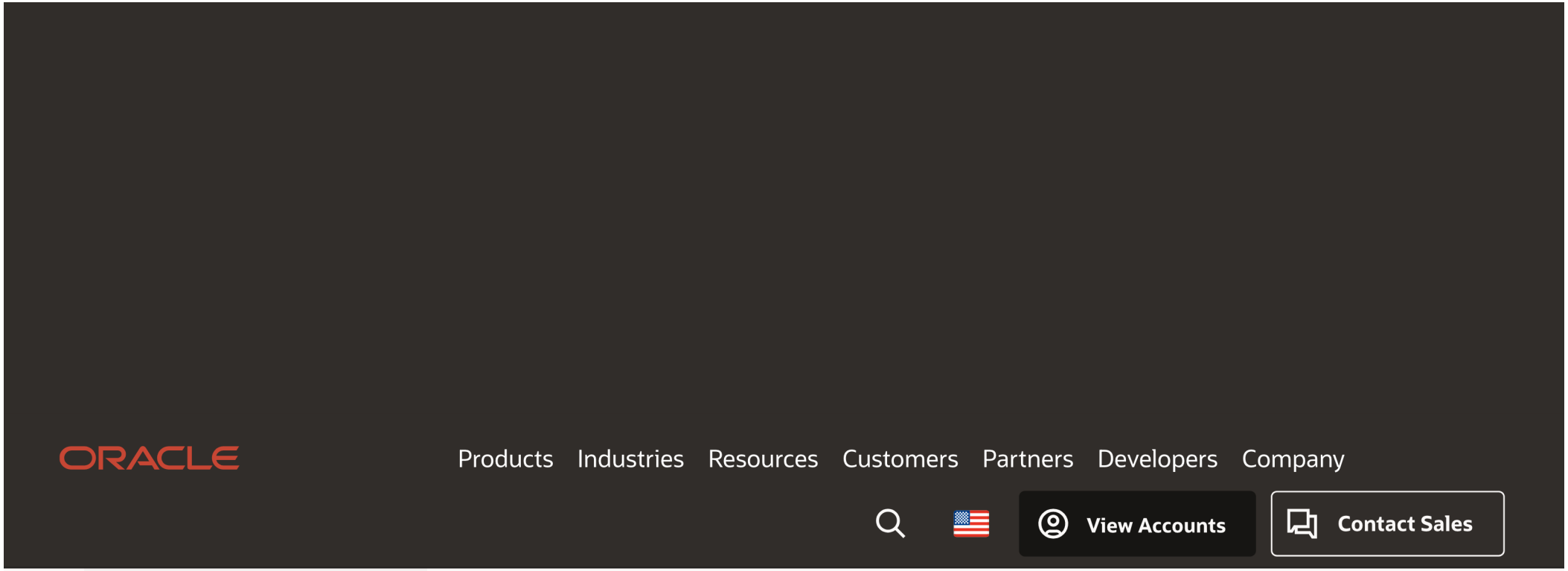
The G1 GC satisfies most allocation requests from regions added to the eden set of regions. During a young garbage collection, the G1 GC collects both the eden regions and the survivor regions from the previous garbage collection. The live objects from the eden and survivor regions are copied, or evacuated, to a new set of regions. The destination region for a particular object depends upon the object's age; an object that has aged sufficiently evacuates to an old generation region (that is, promoted); otherwise, the object evacuates to a survivor region and will be included in the CSet of the next young or mixed garbage collection.

Mixed Garbage Collections

Upon successful completion of a concurrent marking cycle, the G1 GC switches from performing young garbage collections to performing mixed garbage collections. In a mixed garbage collection, the G1 GC optionally adds some old regions to the set of eden and survivor regions that will be collected. The exact number of old regions added is controlled by a number of flags that will be discussed later (see "Taming Mixed GCs"). After the G1 GC collects a sufficient number of old regions (over multiple mixed garbage collections), G1 reverts to performing young garbage collections until the next marking cycle completes.

Phases of the Marking Cycle

The marking cycle has the following phases:



Sets a target value for desired maximum pause time. The default value is 200 milliseconds. The specified value does not adapt to your heap size.

- `-XX:G1NewSizePercent=5`
  Sets the percentage of the heap to use as the minimum for the young generation size. The default value is 5 percent of your Java heap. This is an experimental flag. See "How to unlock experimental VM flags" for an example. This setting replaces the `-XX:DefaultMinNewGenPercent` setting. This setting is not available in Java HotSpot VM, build 23.

- `-XX:G1MaxNewSizePercent=60`
  Sets the percentage of the heap size to use as the maximum for young generation size. The default value is 60 percent of your Java heap. This is an experimental flag. See "How to unlock experimental VM flags" for an example. This setting replaces the `-XX:DefaultMaxNewGenPercent` setting. This setting is not available in Java HotSpot VM, build 23.

- `-XX:ParallelGCThreads=n`

Sets the value of the STW worker threads. Sets the value of n to the number of logical processors. The value of `n` is the same as the number of logical processors up to a value of 8.

If there are more than eight logical processors, sets the value of `n` to approximately 5/8 of the logical processors. This works in most cases except for larger SPARC systems where the value of `n` can be approximately 5/16 of the logical processors.

- `-XX:ConcGCThreads=n`
  Sets the number of parallel marking threads. Sets `n` to approximately 1/4 of the number of parallel garbage collection threads (`ParallelGCThreads`).

- `-XX:InitiatingHeapOccupancyPercent=45`
  Sets the Java heap occupancy threshold that triggers a marking cycle. The default occupancy is 45 percent of the entire Java heap.

- `-XX:G1MixedGCLiveThresholdPercent=65`
  Sets the occupancy threshold for an old region to be included in a mixed garbage collection cycle. The default occupancy is 65 percent. This is an experimental flag. See "How to unlock experimental VM flags" for an example. This setting replaces the `-XX:G1OldCSetRegionLiveThresholdPercent` setting. This setting is not available in Java HotSpot VM, build 23.

- `-XX:G1HeapWastePercent=10`
  Sets the percentage of heap that you are willing to waste. The Java HotSpot VM does not initiate the mixed garbage collection cycle when the reclaimable percentage is less than the heap waste percentage. The default is 10 percent. This setting is not available in Java HotSpot VM, build 23.

- `-XX:G1MixedGCCountTarget=8`
  Sets the target number of mixed garbage collections after a marking cycle to collect old regions with at most `G1MixedGCLIveThresholdPercent` live data. The default is 8 mixed garbage collections. The goal for mixed collections is to be within this target number. This setting is not available in Java HotSpot VM, build 23.

- `-XX:G1OldCSetRegionThresholdPercent=10`
  Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle. The default is 10 percent of the Java heap. This setting is not available in Java HotSpot VM, build 23.

- `-XX:G1ReservePercent=10`
  Sets the percentage of reserve memory to keep free so as to reduce the risk of to-space overflows. The default is 10 percent. When you increase or decrease the percentage, make sure to adjust the total Java heap by the same amount. This setting is not available in Java HotSpot VM, build 23.

How to Unlock Experimental VM Flags

To change the value of experimental flags, you must unlock them first. You can do this by setting `-XX:+UnlockExperimentalVMOptions` explicitly on the command line before any experimental flags. For example:

```
> java -XX:+UnlockExperimentalVMOptions -XX:G1NewSizePercent=10 -XX:G1MaxNewSizePercent=75 G1test.jar
```

Recommendations

When you evaluate and fine-tune G1 GC, keep the following recommendations in mind:

- **Young Generation Size**: Avoid explicitly setting young generation size with the `-Xmn` option or any or other related option such as `-XX:NewRatio`. Fixing the size of the young generation overrides the target pause-time goal.
- **Pause Time Goals**: When you evaluate or tune any garbage collection, there is always a latency versus throughput trade-off. The G1 GC is an incremental garbage collector with uniform pauses, but also more overhead on the application threads. The throughput goal for the G1 GC is 90 percent application time and 10 percent garbage collection time. When you compare this to Java HotSpot VM's throughput collector, the goal there is 99 percent application time and 1 percent garbage collection time. Therefore, when you evaluate the G1 GC for throughput, relax your pause-time target. Setting too aggressive a goal indicates that you are willing to bear an increase in garbage collection overhead, which has a direct impact on throughput. When you evaluate the G1 GC for latency, you set your desired (soft) real-time goal, and the G1 GC will try to meet it. As a side effect, throughput may suffer.
- **Taming Mixed Garbage Collections**: Experiment with the following options when you tune mixed garbage collections. See "Important Defaults" for information about these options:

  - `-XX:InitiatingHeapOccupancyPercent`
    For changing the marking threshold.

- `-XX:G1MixedGCLiveThresholdPercent` and `-XX:G1HeapWastePercent`
  When you want to change the mixed garbage collections decisions.
- `-XX:G1MixedGCCountTarget` and `-XX:G1OldCSetRegionThresholdPercent`
  When you want to adjust the CSet for old regions.

Overflow and Exhausted Log Messages

When you see to-space overflow/exhausted messages in your logs, the G1 GC does not have enough memory for either survivor or promoted objects, or for both. The Java heap cannot expand since it is already at its max. Example messages:

```
924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space exhausted), 0.1957310 secs]
```

OR

```
924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space overflow), 0.1957310 secs]
```

To alleviate the problem, try the following adjustments:

Increase the value of the `-XX:G1ReservePercent` option (and the total heap accordingly) to increase the amount of reserve memory for "to-space".

Start the marking cycle earlier by reducing the `-XX:InitiatingHeapOccupancyPercent`.

You can also increase the value of the `-XX:ConcGCThreads` option to increase the number of parallel marking threads.

See "Important Defaults" for a description of these options.

Humongous Objects and Humongous Allocations

For G1 GC, any object that is more than half a region size is considered a "Humongous object". Such an object is allocated directly in the old generation into "Humongous regions". These Humongous regions are a contiguous set of regions. `StartsHumongous` marks the

start of the contiguous set and `ContinuesHumongous` marks the continuation of the set.

Before allocating any Humongous region, the marking threshold is checked, initiating a concurrent cycle, if necessary.

Dead Humongous objects are freed at the end of the marking cycle during the cleanup phase also during a full garbage collection cycle.

In-order to reduce copying overhead, the Humongous objects are not included in any evacuation pause. A full garbage collection cycle compacts Humongous objects in place.

Since each individual set of StartsHumongous and ContinuesHumongous regions contains just one humongous object, the space between the end of the humongous object and the end of the last region spanned by the object is unused. For objects that are just slightly larger than a multiple of the heap region size, this unused space can cause the heap to become fragmented.

If you see back-to-back concurrent cycles initiated due to Humongous allocations and if such allocations are fragmenting your old generation, please increase your `-XX:G1HeapRegionSize` such that previous Humongous objects are no longer Humongous and will follow the regular allocation path.

Conclusion

G1 GC is a regionalized, parallel-concurrent, incremental garbage collector that provides more predictable pauses compared to other HotSpot GCs. The incremental nature lets G1 GC work with larger heaps and still provide reasonable worst-case response times. The adaptive nature of G1 GC just needs a maximum soft-real time pause-time goal along-with the desired maximum and minimum size for the Java heap on the JVM command line.

See Also

The Garbage-First Garbage Collector

Java HotSpot Garbage Collection

About the Author

Monica Beckwith, Principal Member of Technical Staff at Oracle, is the performance lead for the Java HotSpot VM's Garbage First Garbage Collector. She has worked in the performance and architecture industry for over 10 years. Prior to Oracle and Sun Microsystems, Monica lead the performance effort at Spansion Inc. Monica has worked with many industry standard Java based benchmarks with a constant goal of finding opportunities for improvement in the Java HotSpot VM.

Join the Conversation

Join the Java community conversation on Facebook, Twitter, and the Oracle Java Blog!