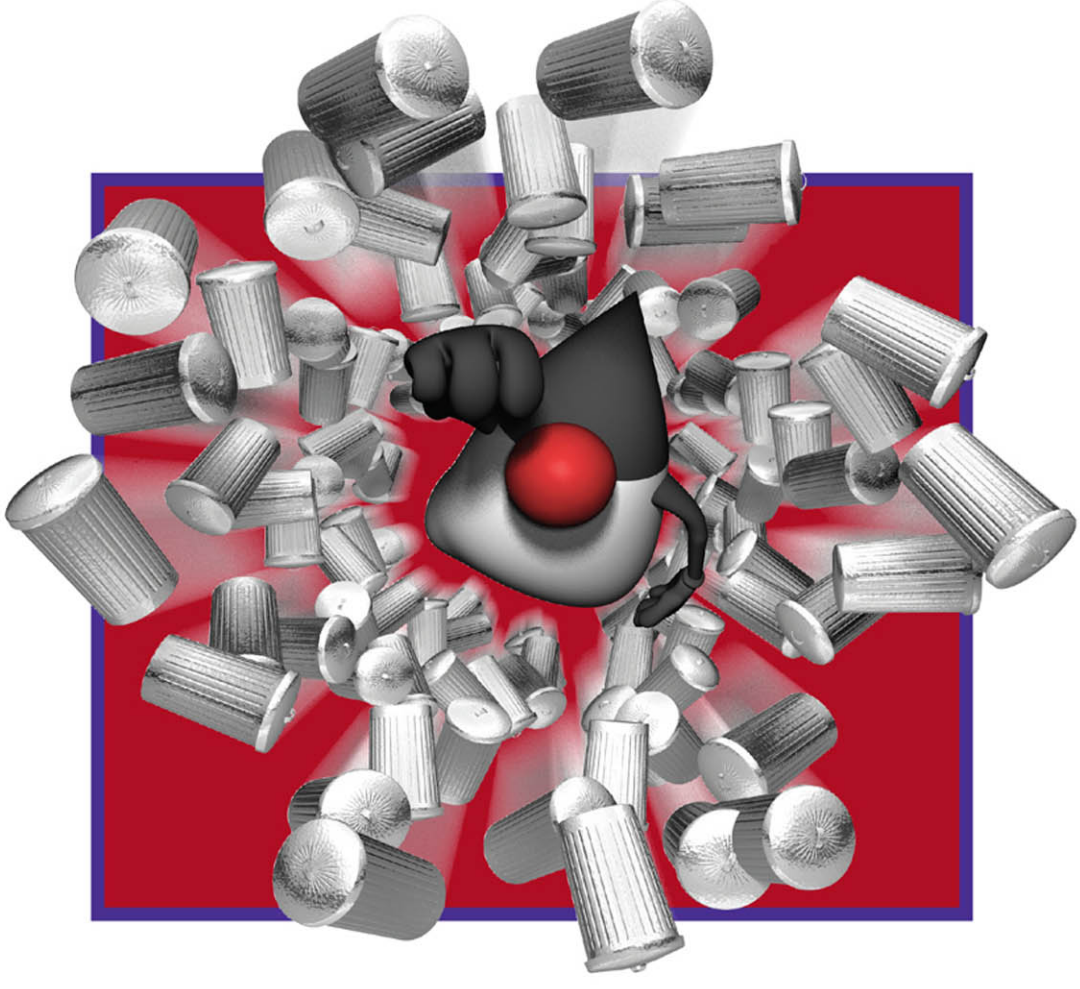
Charlie Hunt · Monica Beckwith
Poonam Parhar · Bengt Rutisson

Java® Performance Companion



Java® Performance Companion

During a full collection, a single thread operates over the entire heap and does mark, sweep, and compaction of all the regions (expensive or otherwise) constituting the generations. After completion of the collection, the resultant heap now consists of purely live objects, and all the generations have been fully compacted.

Tip

Prior to JDK 8u40, unloading of classes was possible only at a full collection.

The single-threaded nature of the serial full collection and the fact that the collection spans the entire heap can make this a very expensive collection, especially if the heap size is fairly large. Hence, it is highly recommended that a nontrivial tuning exercise be done in such cases where full collections are a frequent occurrence.

Tip

For more information on how to get rid of evacuation failures, please refer to Chapter 3.

References

[1] Charlie Hunt and Binu John. *Java™ Performance*. Addison-Wesley, Upper Saddle River, NJ, 2012. ISBN 978-0-13-714252-1.

[2] Tony Printezis and David Detlefs. "A Generational Mostly-Concurrent Garbage Collector." *Proceedings of the 2nd International Symposium on Memory Management.* ACM, New York, 2000, pp. 143–54. ISBN 1-58113-263-8.

[3] Urs Hölzle. "A Fast Write Barrier for Generational Garbage Collectors." Presented at the OOPSLA'93 Garbage Collection Workshop, Washington, DC, October 1993.

[4] Darko Stefanovic, Matthew Hertz, Stephen M. Blackburn, Kathryn S McKinley, and J. Eliot B. Moss. "Older-First Garbage Collection in Practice: Evaluation in a Java Virtual Machine." *Proceedings of the 2002 Workshop on Memory System Performance.* ACM, New York, 2002, pp. 25–36.

[5] Taiichi Yuasa. "Real-Time Garbage Collection on General Purpose Machines." *Journal of Systems and Software,* Volume 11, Issue 3, March 1990, pp. 181–98. Elsevier Science, Inc., New York.

3

Garbage First Garbage Collector Performance Tuning

Performance engineering deals with the nonfunctional performance requirements of a system or its software and ensures that the design requirements are met in the product implementation. Thus it goes hand-in-hand with systems or software engineering.

In this chapter, we will discuss performance tuning in detail, concentrating on the newest garbage collector in Java HotSpot VM: Garbage First, or G1. We will skip young generation tuning advice already provided in *Java™ Performance* [1], particularly Chapter 7, "Tuning the JVM, Step by Step." Readers are encouraged to read that chapter and also the previous chapters in this supplemental book.

The Stages of a Young Collection

A G1 young collection has serial and parallel phases. The pause is serial in the sense that several tasks can be carried out only after certain other tasks are completed during a given stop-the-world pause. The parallel phases employ multiple GC worker threads that have their own work queues and can perform work stealing from other threads' work queues when their own queue tasks are completed.

Tip

The serial stages of the young collection pause can be multithreaded and use the value of `-XX:ParallelGCThreads` to determine the GC worker thread count.

Let's look at an excerpt from a G1 GC output log generated while running DaCapo with the HotSpot VM command-line option `-XX:+PrintGCDetails`. Here is the command-line and "Java version" output:

```
JAVA_OPTS="-XX:+UseG1GC -XX:+PrintGCDetails -Xloggc:jdk8u45_h2.log"
       MB:DaCapo mb$ java -version
       java version "1.8.0_45"
       Java(TM) SE Runtime Environment (build 1.8.0_45-b14)
       Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)
```

This is the GC log snippet (jdk8u45_h2.log):

```
108.815: [GC pause (G1 Evacuation Pause) (young), 0.0543862 secs]
   [Parallel Time: 52.1 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 108815.5, Avg: 108815.5, Max: 108815.6,
Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.2, Diff: 0.1,
Sum: 1.2]
      [Update RS (ms): Min: 12.8, Avg: 13.0, Max: 13.2, Diff: 0.4, Sum:
103.6]
         [Processed Buffers: Min: 15, Avg: 16.0, Max: 17, Diff: 2, Sum: 128]
      [Scan RS (ms): Min: 13.4, Avg: 13.6, Max: 13.7, Diff: 0.3, Sum: 109.0]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0,
Sum: 0.1]
      [Object Copy (ms): Min: 25.1, Avg: 25.2, Max: 25.2, Diff: 0.1, Sum:
201.5]
      [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum:
0.4]
      [GC Worker Total (ms): Min: 51.9, Avg: 52.0, Max: 52.1, Diff: 0.1,
Sum: 416.0]
      [GC Worker End (ms): Min: 108867.5, Avg: 108867.5, Max: 108867.6,
Diff: 0.1]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.2 ms]
   [Other: 2.0 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.1 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.2 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 1.2 ms]
   [Eden: 537.0M(537.0M)->0.0B(538.0M) Survivors: 23.0M->31.0M Heap:
849.2M(1024.0M)->321.4M(1024.0M)]
```

The snippet shows one G1 GC young collection pause, identified in the first line by `(G1 Evacuation Pause)` and `(young)`. The line's timestamp is 108.815, and total pause time is 0.0543862 seconds:

```
108.815: [GC pause (G1 Evacuation Pause) (young), 0.0543862 secs]
```

Start of All Parallel Activities

The second line of the log snippet shows the total time spent in the parallel phase and the GC worker thread count:

```
[Parallel Time: 52.1 ms, GC Workers: 8]
```

The following lines show the major parallel work carried out by the eight worker threads:

```
[GC Worker Start (ms): Min: 108815.5, Avg: 108815.5, Max: 108815.6, Diff:
0.1]
[Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.2, Diff: 0.1, Sum: 1.2]
[Update RS (ms): Min: 12.8, Avg: 13.0, Max: 13.2, Diff: 0.4, Sum: 103.6]
    [Processed Buffers: Min: 15, Avg: 16.0, Max: 17, Diff: 2, Sum: 128]
[Scan RS (ms): Min: 13.4, Avg: 13.6, Max: 13.7, Diff: 0.3, Sum: 109.0]
[Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
[Object Copy (ms): Min: 25.1, Avg: 25.2, Max: 25.2, Diff: 0.1, Sum: 201.5]
[Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
[GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.4]
[GC Worker Total (ms): Min: 51.9, Avg: 52.0, Max: 52.1, Diff: 0.1,
Sum: 416.0]
[GC Worker End (ms): Min: 108867.5, Avg: 108867.5, Max: 108867.6, Diff: 0.1]
```

`GC Worker Start` and `GC Worker End` tag the starting and ending timestamps respectively of the parallel phase. The `Min` timestamp for `GC Worker Start` is the time at which the first worker thread started; similarly, the `Max` timestamp for `GC Worker End` is the time at which the last worker thread completed all its tasks. The lines also contain `Avg` and `Diff` values in milliseconds. The things to look out for in those lines are:

- How far away the `Diff` value is from 0, 0 being the ideal.
- Any major variance in `Max`, `Min`, or `Avg`. This indicates that the worker threads could not start or finish their parallel work around the same time. That could mean that some sort of queue-handling issue exists that requires further analysis by looking at the parallel work done during the parallel phase.

External Root Regions

External root region scanning (`Ext Root Scanning`) is one of the first parallel tasks. During this phase the external (off-heap) roots such as the JVM's system dictionary, VM data structures, JNI thread handles, hardware registers, global variables, and thread stack roots are scanned to find out if any point into the current pause's collection set (CSet).

```
[Ext Root Scanning (ms): Min: 0.1, Avg: 0.2, Max: 0.2, Diff: 0.1, Sum: 1.2]
```

Here again, we look for `Diff` >> 0 and major variance in `Max`, `Min`, or `Avg`.

Tip

The variance (`Diff`) is shown for all the timed activities that make up the parallel phase. A high variance usually means that the work is not balanced across the parallel threads for that particular activity. This knowledge is an analysis starting point, and ideally a deeper dive will identify the potential cause, which may require refactoring the Java application.

Another thing to watch out for is whether a worker thread is caught up in dealing with a single root. We have seen issues where the system dictionary, which is treated as a single root, ends up holding up a worker thread when there is a large number of loaded classes. When a worker thread is late for "termination" (explained later in this section), it is also considered held up.

Remembered Sets and Processed Buffers

```
[Update RS (ms): Min: 12.8, Avg: 13.0, Max: 13.2, Diff: 0.4, Sum: 103.6]
    [Processed Buffers: Min: 15, Avg: 16.0, Max: 17, Diff: 2, Sum: 128]
```

As explained in Chapter 2, "Garbage First Garbage Collection in Depth," G1 GC uses remembered sets (RSets) to help maintain and track references into G1 GC regions that "own" those RSets. The concurrent refinement threads, also discussed in Chapter 2, are tasked with scanning the update log buffers and updating RSets for the regions with dirty cards. In order to supplement the work carried out by the concurrent refinement threads, any remainder buffers that were logged but not yet processed by the refinement threads are handled during the parallel phase of the collection pause and are processed by the worker threads. These buffers are what are referred to as `Processed Buffers` in the log snippet.

In order to limit the time spent updating RSets, G1 sets a target time as a percentage of the pause time goal (`-XX:MaxGCPauseMillis`). The target time defaults to 10 percent of the pause time goal. Any evacuation pause should spend most of its time copying live objects, and 10 percent of the pause time goal is considered a reasonable amount of time to spend updating RSets. If after looking at the logs you realize that spending 10 percent of your pause time goal in updating RSets is undesirable, you can change the percentage by updating the `-XX:G1RSetUpdatingPauseTimePercent` command-line option to reflect your desired value. It is important to remember, however, that if the number of updated log buffers does not change, any decrease in RSet update time during the collection pause will result in fewer buffers being processed during that pause. This will push the log buffer update work off onto the concurrent refinement threads and will result in increased concurrent work and sharing of resources with the Java application mutator threads. Also, worst case, if the concurrent refinement threads cannot keep up with the log buffer update rate, the Java application mutators must step in and help with the processing—a scenario best avoided!

Tip

As discussed in Chapter 2, there is a command-line option called `–XX:G1ConcRefinementThreads`. By default it is set to the same value as `–XX:ParallelGCThreads`, which means that any change in `XX:ParallelGCThreads` will change the `–XX:G1ConcRefinementThreads` value as well.

Before collecting regions in the current CSet, the RSets for the regions in the CSet must be scanned for references into the CSet regions. As discussed in Chapter 2, a popular object in a region or a popular region itself can lead to its RSet being coarsened from a sparse PRT (per-region table) to a fine-grained PRT or even a coarsened bitmap, and thus scanning such an RSet will require more time. In such a scenario, you will see an increase in the `Scan RS` time shown here since the scan times depend on the coarseness gradient of the RSet data structures:

```
[Scan RS (ms): Min: 13.4, Avg: 13.6, Max: 13.7, Diff: 0.3, Sum: 109.0]
```

Another parallel task related to RSets is code root scanning, during which the code root set is scanned to find references into the current CSet:

```
[Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0543862,
Sum: 0.1]
```