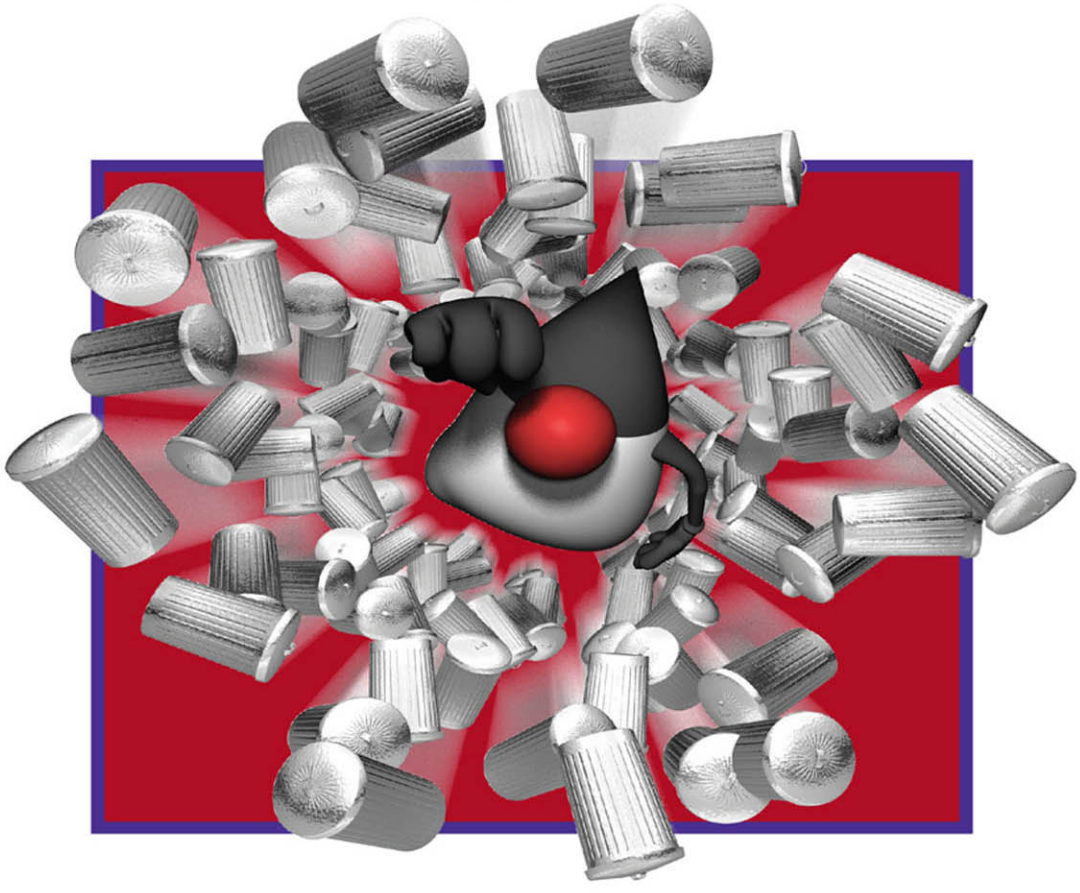
Charlie Hunt · Monica Beckwith
Poonam Parhar · Bengt Rutisson

Java® Performance Companion



FREE SAMPLE CHAPTER

Java® Performance Companion

This page intentionally left blank

Java® Performance Companion

Charlie Hunt
Monica Beckwith
Poonam Parhar
Bengt Rutisson

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-13-379682-7
ISBN-10: 0-13-379682-5

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, April 2016

Contents

Preface

Welcome to the *Java® Performance Companion*. This book offers companion material to *Java™ Performance* [1], which was first published in September 2011. Although the additional topics covered in this book are not as broad as the material in *Java™ Performance*, they go into enormous detail. The topics covered in this book are the G1 garbage collector, also known as the Garbage First garbage collector, and the Java HotSpot VM Serviceability Agent. There is also an appendix that covers additional HotSpot VM command-line options of interest that were not included in the *Java™ Performance* appendix on HotSpot VM command-line options.

If you are currently using Java 8, have interest in migrating to Java 8, or have plans for using Java 9, you will likely be either evaluating G1 GC or already using it. Hence, the information in this book will be useful to you. If you have interest in diagnosing unexpected HotSpot VM failures, or in learning more about the details of a modern Java Virtual Machine, this book's content on the HotSpot VM Serviceability Agent should be of value to you, too. The HotSpot VM Serviceability Agent is the tool of choice for not only HotSpot VM developers but also the Oracle support engineers whose daily job involves diagnosing and troubleshooting unexpected HotSpot VM behavior.

This book begins with an overview of the G1 garbage collector by offering some context around why G1 was implemented and included in HotSpot VM as a GC. It then goes on to offer an overview of how the G1 garbage collector works. This chapter is followed by two additional chapters on G1. The first is an in-depth description of the internals of G1. If you already have a good understanding of how the G1 garbage

collector works, and either have a need to further fine-tune G1 or want to know more about its inner workings, this chapter would be a great place to start. The third chapter on G1 is all about fine-tuning G1 for your application. One of the main design points for G1 was to simplify the tuning required to realize good performance. For instance, the major inputs into G1 are the initial and maximum Java heap size it can use, and a maximum GC pause time you are willing to tolerate. From there G1 will attempt to adaptively adjust to meet those inputs while it executes your application. In circumstances where you would like to achieve better performance, or you would like to do some additional tuning on G1, this chapter has the information you are looking for.

The remaining chapter is dedicated entirely to the HotSpot VM Serviceability Agent. This chapter provides an in-depth description of and instructions for how to use the Serviceability Agent. If you have interest in learning more about the internals of the HotSpot VM, or how to troubleshoot and diagnose unexpected HotSpot VM issues, this is a good chapter for you. In this chapter you will learn how to use the HotSpot VM Serviceability Agent to observe and analyze HotSpot VM behavior in a variety of ways through examples and illustrations.

Last, there is an appendix that includes HotSpot VM command-line options that were not included in *Java™ Performance*'s appendix on HotSpot VM command-line options. Many of the HotSpot VM command-line options found in the appendix are related to G1. And, rather than merely listing these options with only a description, an attempt is made to also mention when it is appropriate to use them.

References

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

Acknowledgments

Charlie Hunt

For those who have ever considered writing a book, or are curious about the effort involved in doing so, the book-writing experience is a major undertaking! For me it just would not have happened without the help of so many people. I cannot begin to mention everyone who made this possible.

In an attempt to at least name those who have had a profound impact on this book getting drafted and eventually into print, I would first like to thank my coauthors, Monica Beckwith, Bengt Rutisson, and Poonam Parhar. When the idea of doing a companion book to *Java™ Performance* first surfaced, I thought it would be great to offer the opportunity to these talented HotSpot VM engineers to showcase their expertise. I am sure I have learned much more from each of them than they have learned from me. I could not be prouder of their contributions to this book.

I also extend sincere thanks to Monica Beckwith for her persistence and passion in sharing her in-depth knowledge of G1 GC. In the early days of G1, I had the pleasure of working with Monica on a daily basis on G1 performance, eventually handing off full reins to her. She has done an exceptional job with driving G1's performance and sharing her G1 knowledge.

I also have to explicitly call out Poonam Parhar and thank her for her patience. Poonam so patiently waited for the other contributors to complete their initial drafts—patiently as in years of patience! Had all of us finished our drafts in a timely way, this book probably would have been on the shelf at least two years earlier.

I also extend my thanks to the entire HotSpot VM team, the HotSpot GC engineering team, and in particular the G1 GC engineers, both past and present.

And to the reviewers of the material in this book: Paul Hohensee for his relentless attention to detail and incredible suggestions for improved readability, and Tony Printezis for his thorough review of the gory details of G1 GC and recommended tuning of G1.

Thanks also to John Cuthbertson for sharing his knowledge of G1. John also happens to be one of the most talented concurrency troubleshooting engineers I have ever worked with. I don't think I ever saw a situation where I was able to stump him with some bizarre observation about G1 that was clearly some kind of concurrency bug. He was always able to track it down.

And to Bernard Traversat and Georges Saab for their support and encouragement in pulling together material for a follow-on to *Java™ Performance*.

And obviously thanks to Greg Doench, our editor, for his patience with our many delays in delivering drafts, completing reviews, and getting the manuscript in shape to put in his hands.

Last, thanks to my wife, Barb, and son, Boyd, for putting up with yet another round of the book-writing experience!

Monica Beckwith

I felt honored when I was approached by my mentor Charlie Hunt to write a few chapters for this book. I didn't have the slightest idea that it would take me so long. So, my first set of thanks goes to my fellow writers for their patience and to Charlie for his persistence and encouragement throughout. While we are talking about encouragement, I want to thank my hubby, Ben Beckwith—when he saw my frustration he had nothing but words of encouragement for me. He was also the initial reviewer of my drafts. Thank you, Ben. And then, of course, my two kiddos, Annika and Bodin, and my mom, Usha, who have been nothing but supportive of me and of this book.

My technical strength on G1 taps off John Cuthbertson, and I am thankful to him for supporting my crazy queries and patiently listening and working with me to "make G1 adaptive" and to "tame mixed collections." When we used to discuss the adaptive marking threshold, I got tired of typing and talking about `InitiatingHeapOccupancyPercent`, so I shortened it to IHOP and John just loved it. It's really hard to find such supportive colleagues as John and Charlie.

And then there are Paul Hohensee and Tony Printezis. They are my mentors in their own right, and I can assure you that their persistence in reviewing my chapters has improved the readability and content by at least 75 percent! :)

Thank you all for trusting me and encouraging me. I am forever in your debt!

Poonam Parhar

I was deeply honored and excited when Charlie suggested that I write a chapter on the Serviceability Agent. I thought it was a great idea, as this wonderful tool is little known to the world, and it would be great to talk about its usefulness and capabilities. But I had never written a book before, and I was nervous. Big thanks to Charlie for his trust in me, for his encouragement, and for guiding me throughout writing the chapter on the SA.

I would like to thank my manager, Mattis Castegren, for always being supportive and encouraging of my work on this book, and for being the first reviewer of the chapter on the SA. Huge thanks to Kevin Walls for reviewing my chapter and helping me improve the quality of the content.

Special thanks to my husband, Onkar, who is my best friend, too, for being supportive and always being there whenever I need help. And of course I am grateful to my two little angels, Amanvir and Karanvir, who are my continuous source of motivation and happiness.

And my most sincere thanks to my father, Subhash C. Bajaj, for his infectious cheerfulness and for being a source of light, and for always inspiring me to never give up.

Bengt Rutisson

When Charlie asked me to write a chapter for this book, I was very honored and flattered. I had never written a book before and clearly had no idea how much work it is—even to write just one chapter! I am very grateful for all the support from Charlie and the reviewers. Without their help, I would not have been able to complete this chapter.

A big thanks to my wife, Sara Fritzell, who encouraged me throughout the work and helped me set up deadlines to get the chapter completed. And, of course, many thanks to our children, Max, Elsa, Teo, Emil, and Lina, for putting up with me during the writing period.

I would also like to thank all of the members of the HotSpot GC engineering team, both past and present. They are by far the most talented bunch of engineers I have ever worked with. I have learned so much from all of them, and they have all inspired me in so many ways.

This page intentionally left blank

About the Authors

Charlie Hunt (Chicago, IL) is currently a JVM Engineer at Oracle leading a variety of Java SE and HotSpot VM projects whose primary focus is reducing memory footprint while maintaining throughput and latency. He is also the lead author of *Java™ Performance* (Addison-Wesley, 2012). He is a regular presenter at the JavaOne Conference where he has been recognized as a Java Rock Star. He has also been a speaker at other well-known conferences, including QCon, Velocity, GoTo, and Dreamforce. Prior to leading a variety of Java SE and HotSpot VM projects for Oracle, Charlie worked in several different performance positions, including Performance Engineering Architect at Salesforce.com and HotSpot VM Performance Architect at Oracle and Sun Microsystems. He wrote his first Java application in 1998, joined Sun Microsystems in 1999 as Senior Java Architect, and has had a passion for Java and JVM performance ever since.

Monica Beckwith is an Independent Performance Consultant optimizing customer applications for server-class systems running the Java Virtual Machine. Her past experiences include working with Oracle, Sun Microsystems, and AMD. Monica has worked with Java HotSpot VM optimizing the JIT compiler, the generated code, the JVM heuristics, and garbage collection and collectors. She is a regular speaker at various conferences and has several published articles on topics including garbage collection, the Java memory model, and others. Monica led Oracle's Garbage First Garbage Collector performance team, and was named a JavaOne Rock Star.

Poonam Parhar (Santa Clara, CA) is currently a JVM Sustaining Engineer at Oracle where her primary responsibility is to resolve customer-escalated problems against JRockit and HotSpot VMs. She loves debugging and troubleshooting problems and is always focused on improving the serviceability and supportability of the HotSpot VM. She has nailed down many complex garbage collection issues in the HotSpot VM and is passionate about improving the debugging tools and the serviceability of the product so as to make it easier to troubleshoot and fix garbage-collector-related issues in the HotSpotVM. She has made several contributions to the Serviceability Agent debugger and also developed a VisualVM plugin for it. She presented "VisualVM Plugin for the SA" at the JavaOne 2011 conference. In an attempt to help customers and the Java community, she shares her work experiences and knowledge through the blog she maintains at https://blogs.oracle.com/poonam/.

Bengt Rutisson (Stockholm, Sweden) is a JVM Engineer at Oracle, working on the HotSpot engineering team. He has worked on garbage collectors in JVMs for the past 10 years, first with the JRockit VM and the last six years with the HotSpot VM. Bengt is an active participant in the OpenJDK project, with many contributions of features, stability fixes, and performance enhancements.

1

Garbage First Overview

This chapter is an introduction to the Garbage First (or G1) garbage collector (GC) along with a historical perspective on the garbage collectors in the Java HotSpot Virtual Machine (VM), hereafter called just HotSpot, and the reasoning behind G1's inclusion in HotSpot. The reader is assumed to be familiar with basic garbage collection concepts such as young generation, old generation, and compaction. Chapter 3, "JVM Overview," of the book *Java™ Performance* [1] is a good source for learning more about these concepts.

Serial GC was the first garbage collector introduced in HotSpot in 1999 as part of Java Development Kit (JDK) 1.3.1. The Parallel and Concurrent Mark Sweep collectors were introduced in 2002 as part of JDK 1.4.2. These three collectors roughly correspond to the three most important GC use cases: "minimize memory footprint and concurrent overhead," "maximize application throughput," and "minimize GC-related pause times." One might ask, "Why do we need a new collector such as G1?" Before answering, let's clarify some terminology that is often used when comparing and contrasting garbage collectors. We'll then move on to a brief overview of the four HotSpot garbage collectors, including G1, and identify how G1 differs from the others.

Terminology

In this section, we define the terms *parallel*, *stop-the-world*, and *concurrent*. The term *parallel* means a multithreaded garbage collection operation. When a GC event activity is described as parallel, multiple threads are used to perform it. When a garbage

collector is described as parallel, it uses multiple threads to perform garbage collection. In the case of the HotSpot garbage collectors, almost all multithreaded GC operations are handled by internal Java VM (JVM) threads. One major exception to this is the G1 garbage collector, in which some background GC work can be taken on by the application threads. For more detail see Chapter 2, "Garbage First Garbage Collector in Depth," and Chapter 3, "Garbage First Garbage Collector Performance Tuning."

The term *stop-the-world* means that all Java application threads are stopped during a GC event. A stop-the-world garbage collector is one that stops all Java application threads when it performs a garbage collection. A GC phase or event may be described as stop-the-world, which means that during that particular GC phase or event all Java application threads are stopped.

The term *concurrent* means that garbage collection activity is occurring at the same time as the Java application is executing. A concurrent GC phase or event means that the GC phase or event executes at the same time as the application.

A garbage collector may be described by any one or a combination of these three terms. For example, a parallel concurrent collector is multithreaded (the parallel part) and also executes at the same time as the application (the concurrent part).

Parallel GC

Parallel GC is a parallel stop-the-world collector, which means that when a GC occurs, it stops all application threads and performs the GC work using multiple threads. The GC work can thus be done very efficiently without any interruptions. This is normally the best way to minimize the total time spent doing GC work relative to application work. However, individual pauses of the Java application induced by GC can be fairly long.

Both the young and old generation collections in Parallel GC are parallel and stop-the-world. Old generation collections also perform compaction. Compaction moves objects closer together to eliminate wasted space between them, leading to an optimal heap layout. However, compaction may take a considerable amount of time, which is generally a function of the size of the Java heap and the number and size of live objects in the old generation.

At the time when Parallel GC was introduced in HotSpot, only the young generation used a parallel stop-the-world collector. Old generation collections used a single-threaded stop-the-world collector. Back when Parallel GC was first introduced, the HotSpot command-line option that enabled Parallel GC in this configuration was `-XX:+UseParallelGC`.

At the time when Parallel GC was introduced, the most common use case for servers required throughput optimization, and hence Parallel GC became the default collector for the HotSpot Server VM. Additionally, the sizes of most Java heaps tended to be between 512MB and 2GB, which keeps Parallel GC pause times

relatively low, even for single-threaded stop-the-world collections. Also at the time, latency requirements tended to be more relaxed than they are today. It was common for Web applications to tolerate GC-induced latencies in excess of one second, and as much as three to five seconds.

As Java heap sizes and the number and size of live objects in old generation grew, the time to collect the old generation became longer and longer. At the same time, hardware advances made more hardware threads available. As a result, Parallel GC was enhanced by adding a multithreaded old generation collector to be used with a multithreaded young generation collector. This enhanced Parallel GC reduced the time required to collect and compact the heap.

The enhanced Parallel GC was delivered in a Java 6 update release. It was enabled by a new command-line option called `-XX:+UseParallelOldGC`. When `-XX:+UseParallelOldGC` is enabled, parallel young generation collection is also enabled. This is what we think of today as Parallel GC in HotSpot, a multithreaded stop-the-world young generation collector combined with a multithreaded stop-the-world old generation collector.

Tip

In Java 7 update release 4 (also referred to as Java 7u4, or JDK 7u4), `-XX:+UseParallelOldGC` was made the default GC and the normal mode of operation for Parallel GC. As of Java 7u4, specifying `-XX:+UseParallelGC` also enables `-XX:+UseParallelOldGC`, and likewise specifying `-XX:+UseParallelOldGC` also enables `-XX:+UseParallelGC`.

Parallel GC is a good choice in the following use cases:

1. Application throughput requirements are much more important than latency requirements.

   A batch processing application is a good example since it is noninteractive. When you start a batch execution, you expect it to run to completion as fast as possible.

2. If worst-case application latency requirements can be met, Parallel GC will offer the best throughput. Worst-case latency requirements include both worst-case pause times, and also how frequently the pauses occur. For example, an application may have a latency requirement of "pauses that exceed 500ms shall not occur more than once every two hours, and all pauses shall not exceed three seconds."

   An interactive application with a sufficiently small live data size such that a Parallel GC's full GC event is able to meet or beat worst-case GC-induced latency requirements for the application is a good example that fits this use case. However, since the amount of live data tends to be highly correlated with the size of the Java heap, the types of applications falling into this category are limited.

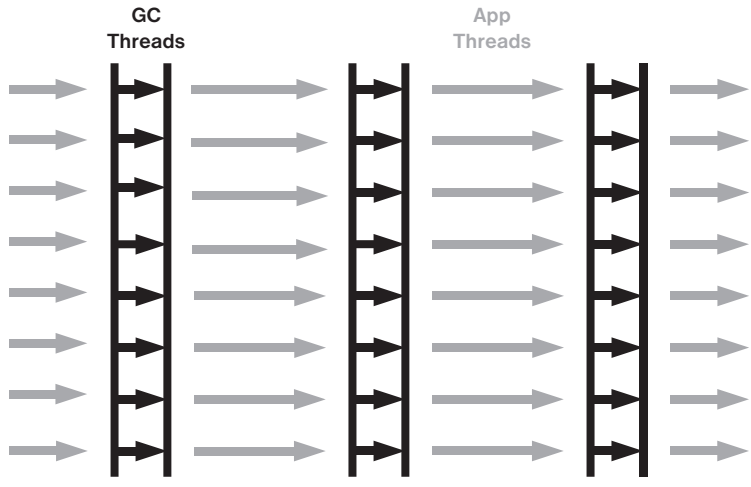


Figure 1.1 How Java application threads are interrupted by GC threads when Parallel GC is used

Parallel GC works well for applications that meet these requirements. For applications that do not meet these requirements, pause times can become excessively long, since a full GC must mark through the entire Java heap and also compact the old generation space. As a result, pause times tend to increase with increased Java heap sizes.

Figure 1.1 illustrates how the Java application threads (gray arrows) are stopped and the GC threads (black arrows) take over to do the garbage collection work. In this diagram there are eight parallel GC threads and eight Java application threads, although in most applications the number of application threads usually exceeds the number of GC threads, especially in cases where some application threads may be idle. When a GC occurs, all application threads are stopped, and multiple GC threads execute during GC.

Serial GC

Serial GC is very similar to Parallel GC except that it does all its work in a single thread. The single-threaded approach allows for a less complex GC implementation and requires very few external runtime data structures. The memory footprint is the lowest of all HotSpot collectors. The challenges with Serial GC are similar to those for Parallel GC. Pause times can be long, and they grow more or less linearly with the heap size and amount of live data. In addition, with Serial GC the long pauses are more pronounced, since the GC work is done in a single thread.

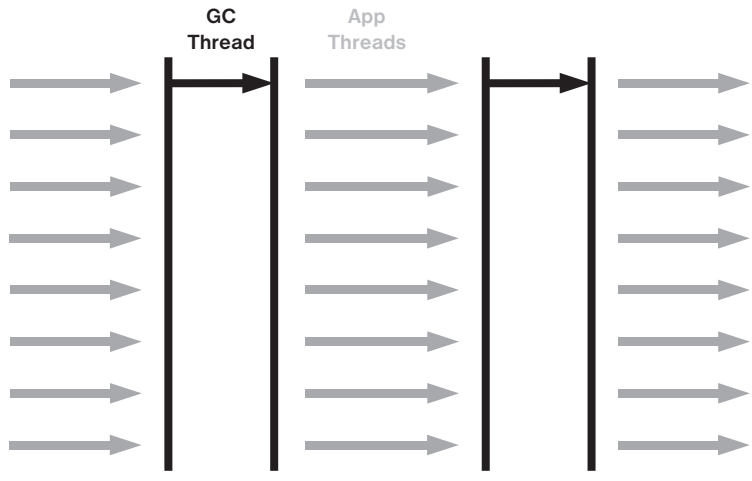


Figure 1.2 How Java application threads are interrupted by a single GC thread when Serial GC is used

Because of the low memory footprint, Serial GC is the default on the Java HotSpot Client VM. It also addresses the requirements for many embedded use cases. Serial GC can be explicitly specified as the GC to use with the `-XX:+UseSerialGC` HotSpot command-line option.

Figure 1.2 illustrates how Java application threads (gray arrows) are stopped and a single GC thread (black arrow) takes over to do the garbage collection work on a machine running eight Java application threads. Because it is single-threaded, Serial GC in most cases will take longer to execute a GC event than Parallel GC since Parallel GC can spread out the GC work to multiple threads.

Concurrent Mark Sweep (CMS) GC

CMS GC was developed in response to an increasing number of applications that demand a GC with lower worst-case pause times than Serial or Parallel GC and where it is acceptable to sacrifice some application throughput to eliminate or greatly reduce the number of lengthy GC pauses.

In CMS GC, young garbage collections are similar to those of Parallel GC. They are parallel stop-the-world, meaning all Java application threads are paused during young garbage collections and the garbage collection work is performed by multiple threads. Note that you can configure CMS GC with a single-threaded young generation collector, but this option has been deprecated in Java 8 and is removed in Java 9.

The major difference between Parallel GC and CMS GC is the old generation collection. For CMS GC, the old generation collections attempt to avoid long pauses in

application threads. To achieve this, the CMS old generation collector does most of its work concurrently with application thread execution, except for a few relatively short GC synchronization pauses. CMS is often referred to as mostly concurrent, since there are some phases of old generation collection that pause application threads. Examples are the initial-mark and remark phases. In CMS's initial implementation, both the initial-mark and remark phases were single-threaded, but they have since been enhanced to be multithreaded. The HotSpot command-line options to support multithreaded initial-mark and remark phases are `-XX:+CMSParallelInitialMarkEnabled` and `-XX:CMSParallelRemarkEnabled`. These are automatically enabled by default when CMS GC is enabled by the `-XX:+UseConcurrentMarkSweepGC` command-line option.

It is possible, and quite likely, for a young generation collection to occur while an old generation concurrent collection is taking place. When this happens, the old generation concurrent collection is interrupted by the young generation collection and immediately resumes upon the latter's completion. The default young generation collector for CMS GC is commonly referred to as ParNew.

Figure 1.3 shows how Java application threads (gray arrows) are stopped for the young GCs (black arrows) and for the CMS initial-mark and remark phases, and old generation GC stop-the-world phases (also black arrows). An old generation collection in CMS GC begins with a stop-the-world initial-mark phase. Once initial mark completes, the concurrent marking phase begins where the Java application threads are allowed to execute concurrently with the CMS marking threads. In Figure 1.3, the concurrent marking threads are the first two longer black arrows, one on top of the other below the "Marking/Pre-cleaning" label. Once concurrent marking completes, concurrent pre-cleaning is executed by the CMS threads, as shown by the two shorter black arrows under the "Marking/Pre-cleaning" label. Note that if there are enough available hardware threads, CMS thread execution overhead will not have much effect on the performance of Java application threads. If, however, the hardware threads are saturated or highly utilized, CMS threads will compete for CPU cycles with Java application threads. Once concurrent pre-cleaning completes, the stop-the-world remark phase begins. The remark phase marks objects that may have been missed after the initial mark and while concurrent marking and concurrent pre-cleaning execute. After the remark phase completes, concurrent sweeping begins, which frees all dead object space.

One of the challenges with CMS GC is tuning it such that the concurrent work can complete before the application runs out of available Java heap space. Hence, one tricky part about CMS is to find the right time to start the concurrent work. A common consequence of the concurrent approach is that CMS normally requires on the order of 10 to 20 percent more Java heap space than Parallel GC to handle the same application. That is part of the price paid for shorter GC pause times.

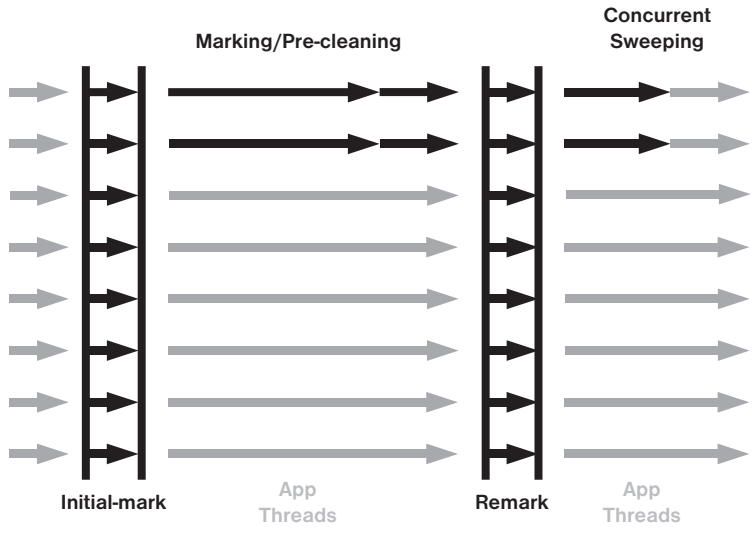


Figure 1.3 How Java application threads are impacted by the GC threads when CMS is used

Another challenge with CMS GC is how it deals with fragmentation in the old generation. Fragmentation occurs when the free space between objects in the old generation becomes so small or nonexistent that an object being promoted from the young generation cannot fit into an available hole. The CMS concurrent collection cycle does not perform compaction, not even incremental or partial compaction. A failure to find an available hole causes CMS to fall back to a full collection using Serial GC, typically resulting in a lengthy pause. Another unfortunate challenge associated with fragmentation in CMS is that it is unpredictable. Some application runs may never experience a full GC resulting from old generation fragmentation while others may experience it regularly.

Tuning CMS GC can help postpone fragmentation, as can application modifications such as avoiding large object allocations. Tuning can be a nontrivial task and requires much expertise. Making changes to the application to avoid fragmentation may also be challenging.

Summary of the Collectors

All of the collectors described thus far have some common issues. One is that the old generation collectors must scan the entire old generation for most of their operations such as marking, sweeping, and compacting. This means that the time to perform the work scales more or less linearly with the Java heap size. Another is that it must

be decided up front where the young and old generations should be placed in the virtual address space, since the young and old generations are separate consecutive chunks of memory.

Garbage First (G1) GC

The G1 garbage collector addresses many of the shortcomings of Parallel, Serial, and CMS GC by taking a somewhat different approach. G1 divides the heap into a set of regions. Most GC operations can then be performed a region at a time rather than on the entire Java heap or an entire generation.

In G1, the young generation is just a set of regions, which means that it is not required to be a consecutive chunk of memory. Similarly, the old generation is also just a set of regions. There is no need to decide at JVM launch time which regions should be part of the old or young generation. In fact, the normal operational state for G1 is that over time the virtual memory mapped to G1 regions moves back and forth between the generations. A G1 region may be designated as young and later, after a young generation collection, become available for use elsewhere, since young generation regions are completely evacuated to unused regions.

In the remainder of this chapter, the term *available region* is used to identify regions that are unused and available for use by G1. An available region can be used or designated as a young or old generation region. It is possible that after a young generation collection, a young generation region can at some future time be used as an old generation region. Likewise, after collection of an old generation region, it becomes an available region that can at some future time be used as a young generation region.

G1 young collections are parallel stop-the-world collections. As mentioned earlier, parallel stop-the-world collections pause all Java application threads while the garbage collector threads execute, and the GC work is spread across multiple threads. As with the other HotSpot garbage collectors, when a young generation collection occurs, the entire young generation is collected.

Old generation G1 collections are quite different from those of the other HotSpot collectors. G1 old generation collections do not require the entire old generation to be collected in order to free space in the old generation. Instead, only a subset of the old generation regions may be collected at any one time. In addition, this subset of old generation regions is collected in conjunction with a young collection.

Tip

The term to describe the collection of a subset of old generation regions in conjunction with a young collection is *mixed GC*. Hence, a mixed GC is a GC event in which all young generation regions are collected in addition to a subset of old generation regions. In other words, a mixed GC is a mix of young and old generation regions that are being collected.

Similar to CMS GC, there is a fail-safe to collect and compact the entire old generation in dire situations such as when old generation space is exhausted.

A G1 old generation collection, ignoring the fail-safe type of collection, is a set of phases, some of which are parallel stop-the-world and some of which are parallel concurrent. That is, some phases are multithreaded and stop all application threads, and others are multithreaded and execute at the same time as the application threads. Chapters 2 and 3 provide more detail on each of these phases.

G1 initiates an old generation collection when a Java heap occupancy threshold is exceeded. It is important to note that the heap occupancy threshold in G1 measures the old generation occupancy compared to the entire Java heap. Readers who are familiar with CMS GC remember that CMS initiates an old generation collection using an occupancy threshold applied against the old generation space only. In G1, once the heap occupancy threshold is reached or exceeded, a parallel stop-the-world initial-mark phase is scheduled to execute.

The initial-mark phase executes at the same time as the next young GC. Once the initial-mark phase completes, a concurrent multithreaded marking phase is initiated to mark all live objects in the old generation. When the concurrent marking phase is completed, a parallel stop-the-world remark phase is scheduled to mark any objects that may have been missed due to application threads executing concurrently with the marking phase. At the end of the remark phase, G1 has full marking information on the old generation regions. If there happen to be old generation regions that do not have any live objects in them, they can be reclaimed without any additional GC work during the next phase of the concurrent cycle, the cleanup phase.

Also at the end of the remark phase, G1 can identify an optimal set of old generations to collect.

Tip

The set of regions to collect during a garbage collection is referred to as a collection set (CSet).

The regions selected for inclusion in a CSet are based on how much space can be freed and the G1 pause time target. After the CSet has been identified, G1 schedules a GC to collect regions in the CSet during the next several young generation GCs. That is, over the next several young GCs, a portion of the old generation will be collected in addition to the young generation. This is the mixed GC type of garbage collection event mentioned earlier.

With G1, every region that is garbage collected, regardless of whether it is young or old generation, has its live objects evacuated to an available region. Once the live objects have been evacuated, the young and/or old regions that have been collected become available regions.

An attractive outcome of evacuating live objects from old generation regions into available regions is that the evacuated objects end up next to each other in the virtual

address space. There is no fragmented empty space between objects. Effectively, G1 does partial compaction of the old generation. Remember that CMS, Parallel, and Serial GC all require a full GC to compact the old generation and that compaction scans the entire old generation.

Since G1 performs GC operations on a per-region basis, it is suitable for large Java heaps. The amount of GC work can be limited to a small set of regions even though the Java heap size may be rather large.

The largest contributors to pause times in G1 are young and mixed collections, so one of the design goals of G1 is to allow the user to set a GC pause time goal. G1 attempts to meet the specified pause time goal through adaptive sizing of the Java heap. It will automatically adjust the size of the young generation and the total Java heap size based on the pause time goal. The lower the pause time goal, the smaller the young generation and the larger the total heap size, making the old generation relatively large.

A G1 design goal is to limit required tuning to setting a maximum Java heap size and specifying a GC pause time target. Otherwise, G1 is designed to dynamically tune itself using internal heuristics. At the time of writing, the heuristics within G1 are where most active HotSpot GC development is taking place. Also as of this writing, G1 may require additional tuning in some cases, but the prerequisites to building good heuristics are present and look promising. For advice on how to tune G1, see Chapter 3.

To summarize, G1 scales better than the other garbage collectors for large Java heaps by splitting the Java heap into regions. G1 deals with Java heap fragmentation with the help of partial compactions, and it does almost all its work in a multithreaded fashion.

As of this writing, G1 primarily targets the use case of large Java heaps with reasonably low pauses, and also those applications that are using CMS GC. There are plans to use G1 to also target the throughput use case, but for applications looking for high throughput that can tolerate longer GC pauses, Parallel GC is currently the better choice.

G1 Design

As mentioned earlier, G1 divides the Java heap into regions. The region size can vary depending on the size of the heap but must be a power of 2 and at least 1MB and at most 32MB. Possible region sizes are therefore 1, 2, 4, 8, 16, and 32MB. All regions are the same size, and their size does not change during execution of the JVM. The region size calculation is based on the average of the initial and maximum Java heap sizes such that there are about 2000 regions for that average heap size. As an example, for a 16GB Java heap with `-Xmx16g -Xms16g` command-line options, G1 will choose a region size of 16GB/2000 = 8MB.

If the initial and maximum Java heap sizes are far apart or if the heap size is very large, it is possible to have many more than 2000 regions. Similarly, a small heap size may end up with many fewer than 2000 regions.

Each region has an associated remembered set (a collection of the locations that contain pointers into the region, shortened to RSet). The total RSet size is limited but noticeable, so the number of regions has a direct effect on HotSpot's memory footprint. The total size of the RSets heavily depends on application behavior. At the low end, RSet overhead is around 1 percent and at the high end 20 percent of the heap size.

A particular region is used for only one purpose at a time, but when the region is included in a collection, it will be completely evacuated and released as an available region.

There are several types of regions in G1. Available regions are currently unused. Eden regions constitute the young generation eden space, and survivor regions constitute the young generation survivor space. The set of all eden and survivor regions together is the young generation. The number of eden or survivor regions can change from one GC to the next, between young, mixed, or full GCs. Old generation regions comprise most of the old generation. Finally, humongous regions are considered to be part of the old generation and contain objects whose size is 50 percent or more of a region. Until a JDK 8u40 change, humongous regions were collected as part of the old generation, but in JDK 8u40 certain humongous regions are collected as part of a young collection. There is more detail on humongous regions later in this chapter.

The fact that a region can be used for any purpose means that there is no need to partition the heap into contiguous young and old generation segments. Instead, G1 heuristics estimate how many regions the young generation can consist of and still be collected within a given GC pause time target. As the application starts allocating objects, G1 chooses an available region, designates it as an eden region, and starts handing out memory chunks from it to Java threads. Once the region is full, another unused region is designated an eden region. The process continues until the maximum number of eden regions is reached, at which point a young GC is initiated.

During a young GC, all young regions, eden and survivor, are collected. All live objects in those regions are evacuated to either a new survivor region or to an old generation region. Available regions are tagged as survivor or old generation regions as needed when the current evacuation target region becomes full.

When the occupancy of the old generation space, after a GC, reaches or exceeds the initiating heap occupancy threshold, G1 initiates an old generation collection. The occupancy threshold is controlled by the command-line option `-XX:InitiatingHeapOccupancyPercent`, which defaults to 45 percent of the Java heap.

G1 can reclaim old generation regions early when the marking phase shows that they contain no live objects. Such regions are added to the available region set. Old regions containing live objects are scheduled to be included in a future mixed collection.

G1 uses multiple concurrent marking threads. In an attempt to avoid stealing too much CPU from application threads, marking threads do their work in bursts. They do as much work as they can fit into a given time slot and then pause for a while, allowing the Java threads to execute instead.

Humongous Objects

G1 deals specially with large object allocations, or what G1 calls "humongous objects." As mentioned earlier, a humongous object is an object that is 50 percent or more of a region size. That size includes the Java object header. Object header sizes vary between 32- and 64-bit HotSpot VMs. The header size for a given object within a given HotSpot VM can be obtained using the Java Object Layout tool, also known as JOL. As of this writing, the Java Object Layout tool can be found on the Internet [2].

When a humongous object allocation occurs, G1 locates a set of consecutive available regions that together add up to enough memory to contain the humongous object. The first region is tagged as a "humongous start" region and the other regions are marked as "humongous continues" regions. If there are not enough consecutive available regions, G1 will do a full GC to compact the Java heap.

Humongous regions are considered part of the old generation, but they contain only one object. This property allows G1 to eagerly collect a humongous region when the concurrent marking phase detects that it is no longer live. When this happens, all the regions containing the humongous object can be reclaimed at once.

A potential challenge for G1 is that short-lived humongous objects may not be reclaimed until well past the point at which they become unreferenced. JDK 8u40 implemented a method to, in some cases, reclaim a humongous region during a young collection. Avoiding frequent humongous object allocations can be crucial to achieving application performance goals when using G1. The enhancements available in JDK 8u40 help but may not be a solution for all applications having many short-lived humongous objects.

Full Garbage Collections

Full GCs in G1 are implemented using the same algorithm as the Serial GC collector. When a full GC occurs, a full compaction of the entire Java heap is performed. This ensures that the maximum amount of free memory is available to the system. It is important to note that full GCs in G1 are single-threaded and as a result may introduce exceptionally long pause times. Also, G1 is designed such that full GCs are not expected to be necessary. G1 is expected to satisfy application performance

goals without requiring a full GC and can usually be tuned such that a full GC is not needed.

Concurrent Cycle

A G1 concurrent cycle includes the activity of several phases: initial marking, concurrent root region scanning, concurrent marking, remarking, and cleanup. The beginning of a concurrent cycle is the initial mark, and the ending phase is cleanup. All these phases are considered part of "marking the live object graph" with the exception of the cleanup phase.

The purpose of the initial-mark phase is to gather all GC roots. Roots are the starting points of the object graphs. To collect root references from application threads, the application threads must be stopped; thus the initial-mark phase is stop-the-world. In G1, the initial marking is done as part of a young GC pause since a young GC must gather all roots anyway.

The marking operation must also scan and follow all references from objects in the survivor regions. This is what the concurrent root region scanning phase does. During this phase all Java threads are allowed to execute, so no application pauses occur. The only limitation is that the scanning must be completed before the next GC is allowed to start. The reason for that is that a new GC will generate a new set of survivor objects that are different from the initial mark's survivor objects.

Most marking work is done during the concurrent marking phase. Multiple threads cooperate to mark the live object graph. All Java threads are allowed to execute at the same time as the concurrent marking threads, so there is no pause in the application, though an application may experience some throughput reduction.

After concurrent marking is done, another stop-the-world phase is needed to finalize all marking work. This phase is called the "remark phase" and is usually a very short stop-the-world pause.

The final phase of concurrent marking is the cleanup phase. In this phase, regions that were found not to contain any live objects are reclaimed. These regions are not included in a young or mixed GC since they contain no live objects. They are added to the list of available regions.

The marking phases must be completed in order to find out what objects are live so as to make informed decisions about what regions to include in the mixed GCs. Since it is the mixed GCs that are the primary mechanism for freeing up memory in G1, it is important that the marking phase finishes before G1 runs out of available regions. If the marking phase does not finish prior to running out of available regions, G1 will fall back to a full GC to free up memory. This is reliable but slow. Ensuring that the marking phases complete in time to avoid a full GC may require tuning, which is covered in detail in Chapter 3.

Heap Sizing

The Java heap size in G1 is always a multiple of the region size. Except for that limitation, G1 can grow and shrink the heap size dynamically between `-Xms` and `-Xmx` just as the other HotSpot GCs do.

G1 may increase the Java heap size for several reasons:

1. An increase in size can occur based on heap size calculations during a full GC.
2. When a young or mixed GC occurs, G1 calculates the time spent to perform the GC compared to the time spent executing the Java application. If too much time is spent in GC according to the command-line setting `-XX:GCTimeRatio`, the Java heap size is increased. The idea behind growing the Java heap size in this situation is to allow GCs to happen less frequently so that the time spent in GC compared to the time spent executing the application is reduced.

   The default value for `-XX:GCTimeRatio` in G1 is 9. All other HotSpot garbage collectors default to a value of 99. The larger the value for `GCTimeRatio`, the more aggressive the increase in Java heap size. The other HotSpot collectors are thus more aggressive in their decision to increase Java heap size and by default are targeted to spend less time in GC relative to the time spent executing the application.
3. If an object allocation fails, even after having done a GC, rather than immediately falling back to doing a full GC, G1 will attempt to increase the heap size to satisfy the object allocation.
4. If a humongous object allocation fails to find enough consecutive free regions to allocate the object, G1 will try to expand the Java heap to obtain more available regions rather than doing a full GC.
5. When a GC requests a new region into which to evacuate objects, G1 will prefer to increase the size of the Java heap to obtain a new region rather than failing the GC and falling back to a full GC in an attempt to find an available region.

References

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

[2] "Code Tools: jol." OpenJDK, circa 2014. http://openjdk.java.net/projects/code-tools/jol/.

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y