JVM INTERNALS

Understanding the JDK's New Superfast Garbage Collectors

ZGC, Shenandoah, and improvements to G1 get developers closer than ever to pauseless Java.

by Raoul-Gabriel Urma and Richard Warburton

November 21, 2019

Download a PDF of this article

Some of the most exciting developments that have occurred in the last six months have been under the hood in the JDK's garbage collectors (GCs). This article covers a range of different improvements, many of which first appeared in JDK 12 and continued in JDK 13. First, we'll describe Shenandoah, a low-latency GC that operates mostly concurrently with the application. We will also cover recent improvements to ZGC (a low-latency concurrent GC introduced in Java 11) that were released as part of JDK 12. And we'll explain in detail two improvements to the Garbage First (G1) GC, which has been the default GC from Java 9 onwards.

Overview of GCs

One of Java's greatest productivity benefits for developers compared to older languages such as C and C++ is the use of garbage collection. As a Java developer, you mostly don't need to worry about leaking memory if you don't explicitly free memory locations, and you don't need to worry about crashing your application if you free memory before you're done using it. Garbage collection is a big productivity win, but time and time again, developers have been concerned with its performance implications. Will it slow your application down? Will it cause individual pauses to the application that will cause a poor experience for your users?

Many garbage collection algorithms have been tried and tested over the years, iteratively improving their performance. There are two common areas of performance for such algorithms. The first is garbage collection throughput: How much of your application's CPU time is spent performing garbage collection work rather than running application code? The second is the delay created—that is, the latency of individual pauses.

For many pausing GCs (for example, Parallel GC, which was the default GC before Java 9), increasing the heap size of the application improves throughput but makes worst-case pauses longer. For GCs with this profile, larger heaps mean that your garbage collection cycles run less frequently and, thus, amortize their collection work more effectively, but the individual pause times take longer because there's more work to do in an individual cycle. Using the Parallel GC on a large heap can result in significant pauses, because the time it takes to collect the old generation of allocated objects scales with the size of the generation and, thus, the heap. But if you're running something such as a non-interactive batch job, the Parallel GC can be an efficient collector.

Since Java 9, the G1 collector has been the default GC in OpenJDK and Oracle JDK. G1's overall approach to garbage collection is to slice up GC pauses according to a user-supplied time target. This means that if you want shorter pauses, set a lower target, and if you want less of the CPU used by your GC and more used by your application, set a larger target. Whereas Parallel GC was a throughput-oriented collector, G1 tries to be a jack of all trades: It offers lesser throughput but better pause times.

However, G1 isn't a master of pause times. As the amount of work to be done during a garbage collection cycle increases, either due to a very large heap or to rapidly allocating lots of objects, the time-slicing approach starts to hit a wall. By analogy, chopping a big piece of food into small pieces makes those pieces easier to digest, but if you've just got too much food on your plate, it's going to take you ages to eat dinner. Garbage collection works the same way.

This is the problem space that JDK 12's Shenandoah GC attacks: It's a latency specialist. It can consistently achieve low pause times, even on large heaps. It might spend a bit more CPU time performing garbage collection work than the Parallel GC, but the pause times are greatly reduced. That's great for low-latency systems in the finance, gambling, or advertising industries or even for interactive websites where users can be frustrated by long pauses.

In this article, we explain the latest versions of these GCs as well as the recent updates to G1 and, we hope, help guide you to the balance of features that works best for your applications.

Shenandoah

Shenandoah is a new GC that was released as part of JDK 12. In fact, the Shenandoah development effort backports improvements to JDK 8u and 11u releases as well, which is great if you haven't had the opportunity to upgrade to JDK 12.

Let's look at who should think about switching over to it and why. We won't be going into too much detail about how Shenandoah works under the hood, but if you're interested in the technology, you should look at the accompanying article and also at the Shenandoah page on the OpenJDK wiki.

Shenandoah's key advance over G1 is to do more of its garbage collection cycle work concurrently with the application threads. G1 can evacuate its heap regions, that is, move objects, only when the application is paused, while Shenandoah can relocate objects concurrently with the application. To achieve the concurrent relocation, it uses what's known as a *Brooks pointer*. This pointer is an additional field that each object in the Shenandoah heap has and which points back to the object itself.

Shenandoah does this because when it moves an object, it also needs to fix up all the objects in the heap that have references to that object. When Shenandoah moves an object to a new location, it leaves the old Brooks pointer in place, forwarding references to the new location of the object. When an object is referenced, the application follows the forwarding pointer to the new location. Eventually the old object with the forwarding pointer needs to be cleaned up, but by decoupling the cleanup operation from the step of moving the object itself, Shenandoah can more easily accomplish the concurrent relocation of objects.

To use Shenandoah in your application from Java 12 onwards, enable it with the following options:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoah
```

If you can't yet make the jump to Java 12, but you are interested in trying out Shenandoah, backports to Java 8 and Java 11 are available. It's worth noting that Shenandoah isn't enabled in the JDK builds that Oracle ships, but other OpenJDK distributors enable Shenandoah by default. More details on Shenandoah can be found in JEP 189.

Shenandoah isn't the only option when it comes to concurrent GCs. ZGC is another GC that is shipped with OpenJDK (including with Oracle's builds), and it has been improved in JDK 12. So if you have an app that suffers from garbage collection pause problems and you are thinking about trying Shenandoah, you should also look at ZGC, which we describe next.

ZGC with Concurrent Class Unloading

The primary goals of ZGC are low latency, scalability, and ease of use. To achieve this, ZGC allows a Java application to continue running while it performs all garbage collection operations except thread stack scanning. It scales from a few hundred MB to TB-size Java heaps, while consistently maintaining very low pause times—typically within 2 ms.

The implications of predictably low pause times could be profound for both application developers and system architects. Developers will no longer need to worry about designing elaborate ways to avoid garbage collection pauses. And system architects will not require specialized GC performance tuning expertise to achieve the dependably low pause times that are very important for so many use cases. This makes ZGC a good fit for applications that require large amounts of memory, such as with big data. However, ZGC is also a good candidate for smaller heaps that require predictable and extremely low pause times.

ZGC was added to JDK 11 as an experimental feature. In JDK 12, ZGC added support for concurrent class unloading, allowing Java applications to continue running during the unloading of unused classes instead of pausing execution.

Performing concurrent class unloading is complicated and, therefore, class unloading has traditionally been done in a stop-the-world pause. Determining the set of classes that are no longer used requires performing reference processing first. Then there's the processing of finalizers–which is how we refer to implementations of the `Object.finalize()` method. As part of reference processing, the set of objects reachable from finalizers must be traversed, because a finalizer could transitively keep a class alive through an unbounded chain of links. Unfortunately, visiting all objects reachable from finalizers could take a very long time. In the worst-case scenario, the whole Java heap could be reachable from a single finalizer. ZGC runs reference processing concurrently with the Java application (since the introduction of ZGC in JDK 11).

After reference processing has finished, ZGC knows which classes are no longer needed. The next step is to clean all data structures containing stale and invalid data as a result of these classes dying. Links from data structures that are alive to data structures that have become invalid or dead are cleared. The data structures that need to be walked for this unlinking operation include several internal JVM data structures, such as the code cache (containing all JIT-compiled code), class loader data graph, string table, symbol table, profile data, and so forth. After unlinking the dead data structures is finished, those dead data structures are walked again to delete them, so that memory is finally reclaimed.

Until now, all JDK GCs have done all of this in a stop-the-world operation, causing latency issues for Java applications. For a low-latency GC, this is problematic. Therefore, ZGC now runs all of this concurrently with the Java application and, hence, pays no latency penalty for supporting class unloading. In fact, the mechanisms introduced to perform concurrent class unloading improved latencies even further. The time spent inside of stop-the-world pauses for garbage collection is now proportional only to the number of threads in the application. The significant effect this approach has on pause times is shown in **Figure 1**.

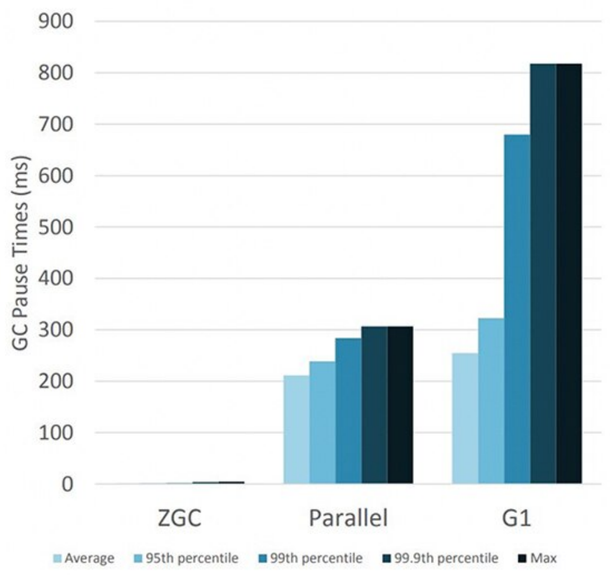

Figure 1. The pause times of ZGC compared with other GCs

ZGC is currently available as an experimental GC for the Linux/x86 64-bit platform and, as of Java 13, on Linux/Aarch. You can enable it with the following command-line options:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC
```

More information on ZGC can be found on the OpenJDK wiki.

G1 Improvements

Some organizations cannot change their runtime systems to use experimental GCs. They will be happy to know that G1 has enjoyed several improvements. The G1 collector time-slices its garbage collection cycles into multiple different pauses.

Objects are initially considered to be part of the “young” generation after they are allocated. As they stay alive over multiple garbage collection cycles, they eventually “tenure” and are then considered “old.” Different regions within G1 contain objects from only one generation and can thus be referred to as young regions or old regions.

For G1 to meet the pause-time goals, it needs to be able to identify a chunk of work that can be done within the pause time goal and finish that work by the time the pause goal expires. G1 has a complicated set of heuristics for identifying the right size of work, and these heuristics are good at predicting the required work time, but they are not always accurate. Complicating the picture still further is the fact that G1 can’t collect only parts of young regions; it collects all the young regions in one garbage collection pass.

In Java 12, this situation is improved by adding the ability to abort G1 collection pauses. G1 keeps track of how accurately its heuristics are predicting the number of regions to collect and proceeds only with abortable garbage collections if it needs to. It proceeds by splitting up the collection set (the set of regions that will be garbage collected in a cycle) into two groups: *mandatory regions* and *optional regions*.

Mandatory regions are always collected within a GC cycle. Optional regions are collected as time allows, and the collection pass is aborted if it runs out of time without collecting the optional regions. The mandatory regions are all the young regions and potentially some old regions. Old-generation regions are added to this set to respond to two criteria. Some are added to ensure that the evacuation of objects can proceed and some in order to use up the expected pause time.

The heuristic to calculate how many regions to add proceeds by dividing the number of regions in the collection set candidates by the value of `-XX:G1MixedGCCountTarget`. If G1 predicts there will be time left to collect more old-generation regions, then it also adds more regions to the mandatory region set until it expects to use up 80% of the available pause time.

The result of this work means that G1 is able to abort, or end, its mixed GC cycles. This results in lower GC pause latency and a high probability that G1 is able to achieve its pause-time target more frequently. This improvement is detailed in JEP 344.

Prompt Return of Unused, Committed Memory

One of the most common criticisms leveled at Java is that it's a memory hog—well, not anymore! Sometimes, JVMs are allocated more memory than they need through command-line options; and if no memory-related command-line options are provided, the JVM may allocate more memory than needed. Allocating RAM that goes unused wastes money, especially in cloud environments where all resources are metered and costed appropriately. But what can be done to solve this situation, and can Java be improved in terms of resource consumption?

A common situation is that the workload that a JVM must handle changes over time: Sometimes it needs more memory and sometimes less. In practice, this is often irrelevant because JVMs tend to allocate a large quantity of memory on startup and greedily hold onto it even when they don't need it. In an ideal world, unused memory could be returned from the JVM back to the operating system so other applications or the container would be able to use it. As of Java 12, this return of unused memory is now possible.

G1 already has the capability to free unused memory, but it does so only during full garbage collection passes. Full garbage collection passes are often infrequent and an undesirable occurrence, because they can entail a long stop-the-world application pause. In JDK 12, G1 gained the ability to free unused memory during concurrent garbage collection passes. This feature is especially useful for mostly empty heaps. When heaps are mostly empty, it can take a while for a GC cycle to scoop up the memory and return it to the operating system. To ensure that memory is promptly returned to the operating system, as of Java 12, G1 will try to trigger concurrent garbage collection cycles if a garbage collection cycle hasn't happened for the period specified on the command line by the `G1PeriodicGCInterval` argument. This concurrent garbage collection cycle will then release memory to the operating system at the end of the cycle.

To ensure that these periodic concurrent garbage collection passes don't add unnecessary CPU overhead, they are run only when the system is partially idle. The measurement used to trigger whether the concurrent cycle runs or not is the average one-minute system load value, which has to be below the value specified by `G1PeriodicGCSystemLoadThreshold`.

More details can be found in JEP 346.

Conclusion

This article presented several ways in which you can stop worrying about GC-induced pause times in your applications. While G1 continues to improve, it's good to know that as heap sizes increase and the acceptability of pause times is reduced, new GCs such as Shenandoah and ZGC offer a scalable, low-pause future.

Also in This Issue

Epsilon: The JDK's Do-Nothing Garbage Collector
Understanding Garbage Collectors
Testing HTML and JSF-Based UIs with Arquillian

Take Notes As You Code—Lots of ’em!
For the Fun of It: Writing Your Own Text Editor, Part 2
Quiz Yourself: Identify the Scope of Variables (Intermediate)
Quiz Yourself: Inner, Nested, and Anonymous Classes (Advanced)
Quiz Yourself: String Manipulation (Intermediate)
Quiz Yourself: Variable Declaration (Intermediate)
Book Review: The Pragmatic Programmer, 20th Anniversary Edition

Raoul-Gabriel Urma

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is coauthor of the best-selling programming book *Java 8 in Action* (Manning Publications, 2015). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the author of the best-selling *Java 8 Lambdas* (O’Reilly Media, 2014) and helps developers learn via Iteratr Learning and at Pluralsight. Warburton has delivered hundreds of talks and training courses. He holds a PhD from the University of Warwick.

Share this Page