The Z Garbage Collector algorithm (JDK 15 version)

Jesús Navarrete Follow
Dec 9, 2020 · 6 min read

Introduction

It was September 25th when JDK 11 was launched. Among other features, the Z Garbage Collector algorithm also known as ZGC was introduced as an experimental feature. On September 15th, a few weeks ago, JDK 15 was released and the latest version of ZGC is now ready for production.

As a brief summary, ZGC is a scalable low-latency garbage collector with a maximum GC pause time of 10 milliseconds, able to handle from a few megabytes to multi-terabyte heaps and a maximum throughput reduction of 15%.

JVM Garbage collectors

The JVM has introduced an interesting list of garbage collectors algorithms to date, let's remember them with the following list of the most important algorithms introduced and a brief description.

- **Serial (low memory footprint):** it uses a single thread to do the work. It is good for single-processor machines and it is optimized for low memory footprint, typically embedded systems.
- **Parallel (throughput collector):** It makes minor collections in parallel to reduce garbage collection overhead. It was built for medium-sized to large-sized data sets applications, that run on multiprocessor hardware.
- **CMS (Concurrent Mark-Sweep Collector):** It was designed to have shorter garbage collector pauses. Specially designed for applications with a large number of long-lived objects or a large tenured generation. The CMS collector is generational.
- **G1 (throughput/latency balance):** The Garbage-First is a server-style garbage collector, designed for multiprocessor machines with large memories. It is a compacting collector (it compacts enough to avoid the use of fine-grain free lists for allocation).
- **ZGC (low latency)**

> *Serial and Parallel are called "stop of the world" algorithms. CMS was deprecated in JDK 9, having G1 as the replacement.*

It's interesting to compare them. And to highlight that ZGC performs all the heavy operations concurrently, something that none of the other algorithms does (more details down below).

. . .

Diving deep into the Z Garbage Collector

ZGC is a concurrent low latency algorithm, it does everything concurrent (marking, compaction, reference processing, relocation set selection, StringTable cleaning, JNI WeakRef cleaning, JNI GlobalRefs scanning and class unloading), except thread stack scanning. This makes the algorithm really good for low latency.

One important thing to mention is that currently, pause times *do not* increase with the heap or live size, however, pause times *do* increase with the root-set size (number of java threads that your application is using).

From the algorithm point of view, it is a concurrent collector, it does all the heavy lifting work while the java threads continue executing. It is a region-based collector, which means that the heap is divided into smaller regions and the compaction efforts will be focused on a subset of those regions, the ones with the most garbage. It is NUMA-aware, what it decreases latencies because of the local memory to the CPU. It uses colored pointers and load barriers described deeper in the following sections. And it is a single generation collector, it does not have young or old generations, *yet*.

The ZGC Phases

The GC cycle of ZGC is divided into three pauses.

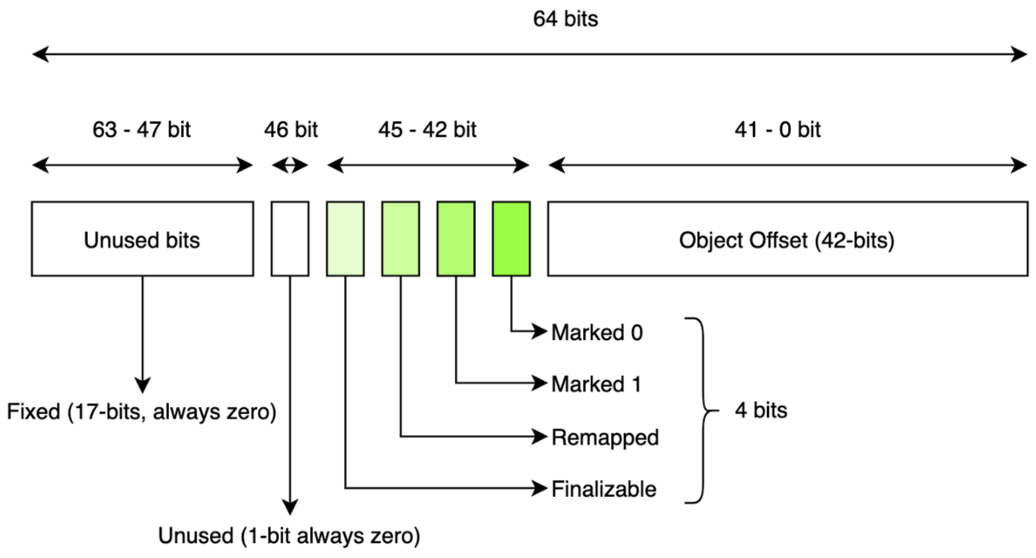
In the first phase, Pause Mark Start, ZGC walks through the object graph to mark the objects live or garbage. This phase also includes the remapping of live data.

The second phase, Pause Mark End, is where reference preprocessing is done. The class unloading and relocation set selection are also done in this phase.

Pause Relocate Start, the last phase, is where the heavy job of compacting the heap happens.

Colored pointers

They are a core design concept in ZGC. The algorithm uses some unused bits in the 64-bit object pointer to store some metadata, that allows finding, mark, locate, and remap the objects. The following diagram shows the 64-bit object pointer and the meaning of each bit.



Colored pointers diagram.

Load barrier

It is code injected by the JIT in some strategic places. The goal is to check whether the loaded object reference has a bad color. The load barrier code will run when a thread loads an object reference from the heap.

```
Object foo = obj.fieldValue;  //Loads an object reference from heap
<load barrier will be needed here>

Object bar = foo;             //No barrier here
foo.doSomething();            //No barrier here
```

Load Barrier example

. . .

Tuning options

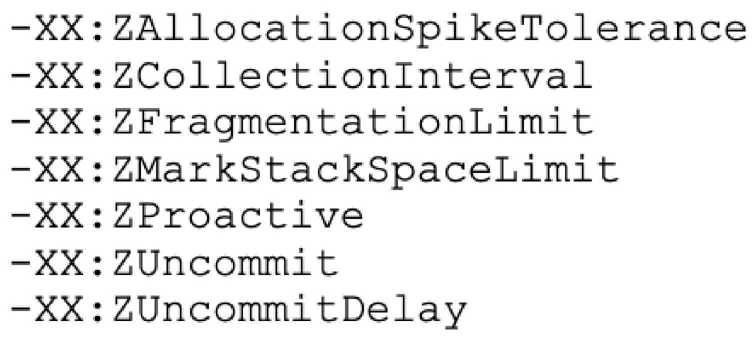
Let me describe how to use ZGC and some interesting parameters to tune. From the JDK 11 until the JDK 15 release, you must unlock the experimental options if you want to use the ZGC algorithm:

-XX:+UnlockExperimentalVMOptions -XX:+UseZGC

From JDK 15 in advance, you can use it just by specifying:

-XX:+UseZGC

The ZGC was designed to be easy to tune. Below there is a list of the specific ZGC options:



```
-XX:ZAllocationSpikeTolerance
-XX:ZCollectionInterval
-XX:ZFragmentationLimit
-XX:ZMarkStackSpaceLimit
-XX:ZProactive
-XX:ZUncommit
-XX:ZUncommitDelay
```

In order to appreciate the times used and to see some numbers about the behavior of the algorithm, it is good to print the garbage collector logs, just add the following command when you choose ZGC to see the simple log:

-XX:+UseZGC -Xmx<size> -Xlog:gc

Or if you want to print the garbage collector logs with more details:

*-XX:+UseZGC -Xmx<size> -Xlog:gc**

Now, let's start taking a look at the most interesting tuning options.

Setting Heap Size

One of the most important options to tune in ZGC is setting the max heap size (-*Xmx<size>*). We must find the correct value for our application because we don't want to lose memory, and we want to allow our application to have enough space for live objects and allocations to happen, while GC is running. The following is an example of use:

-XX:+UseZGC -Xmx<size>

Setting Concurrent GC Threads

Although ZGC has heuristics for setting this number automatically, sometimes, depending on our application it could be interesting to specify the number of concurrent GC threads. This option rules how much CPU will your GC take, so you must be careful about the capacity you want to give.

-XX:+UseZGC -Xmx<size> -XX:ConcGCThreads=<number>

Returning Unused Memory to the Operating System

Unlike other GC algorithms, ZGC uncommits unused memory, giving it back to the operating system. This can be necessary for applications where memory footprint can be a problem. If you want to disable this option, you can use -XX:-ZUncommit.

-XX:+UseZGC -Xmx<size> -XX:-ZUncommit

Enabling Large Pages On Linux

This is an option that comes with an improvement in performance and no disadvantages or side effects. The only issue is that it requires root privileges, that's why is not a default option and might not be possible to enable it for your application. Take a look at the documentation to set up this option properly. It requires to prepare somethings, and the options would look like the following:

-XX:+UseZGC -Xms16G -Xmx16G -XX:+UseLargePages

Enabling Transparent Huge Pages On Linux

Huges pages are not recommended for latency-sensitive applications, although it serves as a good alternative to the previous tuning option.

-XX:+UseZGC -… -XX:+UseLargePages -XX:+UseTransparentHugePages

In this case, I highly recommend you to experiment with it in your application and pay attention to the spikes, if they happen maybe it's not an option for your case.

Enabling NUMA Support

As I said previously, ZGC is NUMA-aware, which means this option is enabled by default. This will direct Java heap allocations to NUMA-local memory. It can automatically be disabled by the JVM, if you need to override explicitly the behavior you can use the options -XX:+UseNUMA or -XX:-UseNUMA.

-XX:+UseZGC -Xmx<size> -XX:+UseNUMA

or

-XX:+UseZGC -Xmx<size> -XX:-UseNUMA

Many more on the wiki page of the algorithm.

. . .

What's coming next?

There are two things that got my attention from the roadmap of the ZGC team.

The first is that the team is working on reducing the maximum pause time to 1 ms. The Thread Stack scanning will be done concurrently, and that would also mean that the pause times will not increase with the root-set size (JEP 376).

And second, they are going to make ZGC generational, because most of the objects are short-lived, this will be another nice improvement.

References

https://openjdk.java.net/jeps/333

https://openjdk.java.net/jeps/377

http://hg.openjdk.java.net/zgc/zgc/file/59c07aef65ac/src/hotspot/os_cpu/linux_x86/zGlobals_linux_x86.hpp#l39

https://wiki.openjdk.java.net/display/zgc/Main

https://www.youtube.com/watch?v=88E86quLmQA

http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf

Garbage Collector Jdk 15 Zgc Java Algorithms