Choosing the Right GC

by Ruslan Synytsky · Dec. 08, 17 · Java Zone · Analysis

Size matters when it comes to software. It has become clear that using small pieces within a microservices architecture delivers more advantages compared to the big monolith approach. The recent Java release of Jigsaw helps decompose legacy applications or build new cloud-native apps from scratch.

This approach reduces disk space, build time, and startup time. However, it doesn't help enough with RAM usage management. It is well-known that Java consumes a large amount of memory in many cases. At the same time, many have not noticed that Java has become much more flexible in terms of memory usage and provided features to meet the requirements of microservices.

In this article, we will share our experiences in tuning RAM usage in a Java process to make it more elastic and gain the benefits of faster scaling and lower total cost of ownership (TCO).

There are three options for scaling: vertical, horizontal, and a combination of both. In order to maximize the outcome, first, you need to set up vertical scaling in an optimal way. Then, when your project grows horizontally, the preconfigured resource consumption within a single container will be replicated to each instance, and efficiency will grow proportionally.

If configured properly, vertical scaling works perfectly for both microservices and monoliths, optimizing memory and CPU usage according to the current load inside containers. The selected garbage collector is one of the main foundational bricks, and its settings can influence the whole project.

There are five widely used garbage collector solutions for OpenJDK:

- G1
- Parallel
- ConcMarkSweep (CMS)
- Serial
- Shenandoah

Let's see how each of these performs in terms of scaling and what settings can be applied to improve results.

For testing, we'll use a sample Java application that helps to track JVM vertical scaling results.

The following JVM start options will be initiated for each GC test:

```
1  java -XX:+Use[gc_name]GC -Xmx2g -Xms32m -jar app.jar [sleep]
```

Where:

- [gc_name] will be substituted with the specific garbage collector type
- Xms is the scaling step (32 MB in our case).
- Xmx is the maximum scaling limit (2 GB in our case).
- [sleep] is the interval between memory load cycles in milliseconds. The default is 10.

At the moment, invoking Full GC is required for a proper release of unused resources. It can be easily initiated with various options:

At the moment, invoking Full GC is required for a proper release of unused resources. It can be easily initiated with various options:

- jcmd <pid> GC.run — executing external calls.
- System.gc() — inside the source code.
- jvisualvm — manually via the great VisualVM troubleshooting tool.
- -javaagent:agent.jar — pluggable commonly used approach. This open source automation add-on is available at the GitHub repo Java Memory Agent.

Memory usage can be tracked in output logs or using VisualVM for a deeper review.

G1 Garbage Collector

The good news for the Java ecosystem is that, starting with JDK 9, the modern shrinking G1 garbage collector is enabled by default. If you use a JDK of lower release, G1 can be enabled with the `-XX:+UseG1GC` parameter.

One of G1's main advantages is the ability to compact free memory space without lengthy pause times. It can also uncommit unused heaps. We found this GC to be the best option for vertical scaling of Java applications running on OpenJDK or HotSpot JDK.

To gain a better understanding of how the JVM behaves at different memory pressure levels, we'll run three cases:

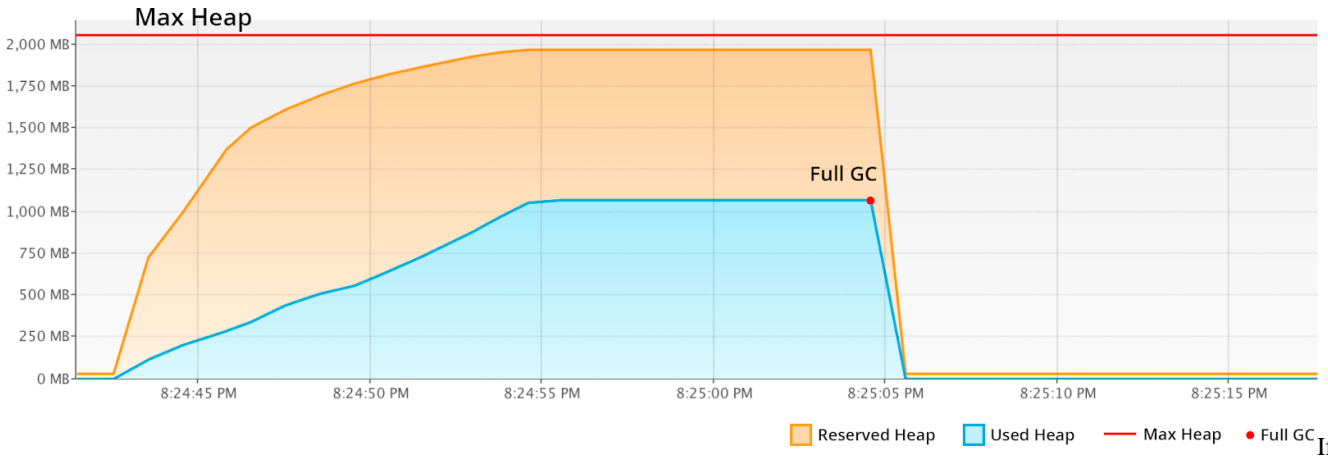
1. Fast memory usage growth.
2. Medium memory usage growth.
3. Slow memory usage growth.

In this way, we can check how smart G1's ergonomics are and how GC handles different memory usage dynamics.

Fast Memory Usage Growth

```
java -XX:+UseG1GC -Xmx2g -Xms32m -jar app.jar 0
```

Memory grew from 32 MiB to 1 GiB in 25 seconds.



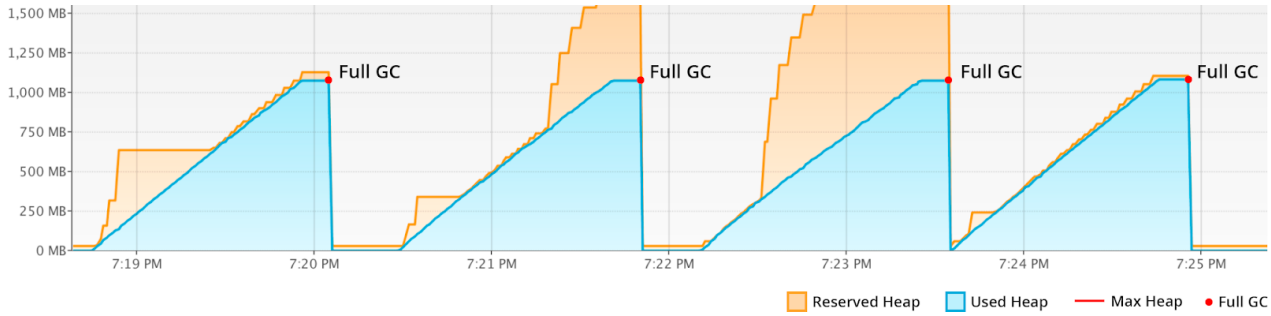
If memory usage growth is very fast, the JVM's ergonomics ignore Xms scaling steps and reserves RAM faster according to its internal adaptive optimization algorithm. As a result, we see much faster RAM allocation for JVM (orange) relative to the fast real usage (blue) growth. So with G1, we are safe, even in the event of load spikes.

Medium Memory Usage Growth

```
java -XX:+UseG1GC -Xmx2g -Xms32m -jar app.jar 10
```

Memory grew from 32 MiB to 1 GiB in 90 seconds during four cycles.

Sometimes it requires several cycles for JVM ergonomic to find an optimal RAM allocation algorithm.

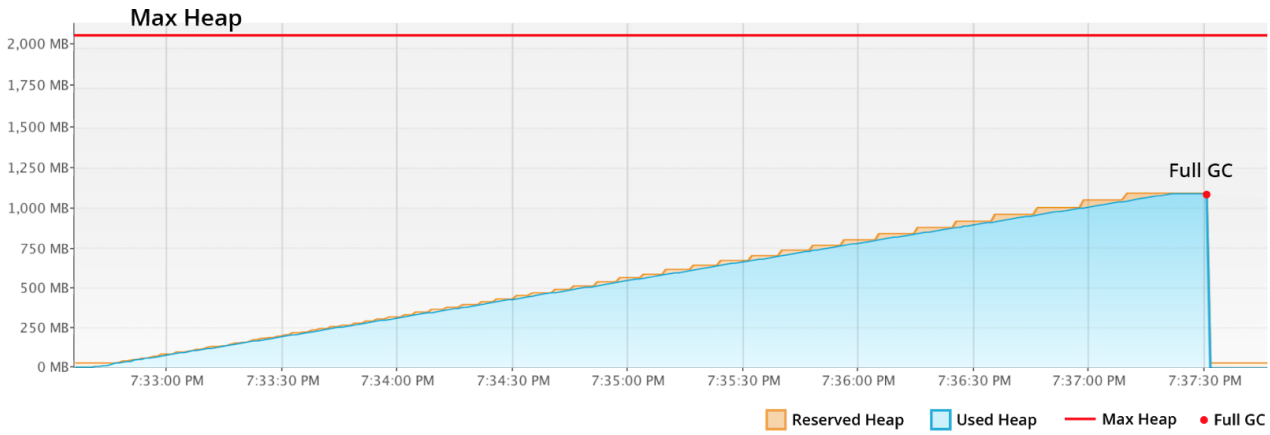


As we can see, the JVM tuned the RAM allocation ergonomics at the fourth cycle to make vertical scaling very efficient for repeatable cycles.

Slow Memory Usage Growth

```
java -XX:+UseG1GC -Xmx2g -Xms32m -jar app.jar 100
```

Memory grew from 32 MiB to 1 GiB with a delta time growth of about 300 seconds. That's very elastic and efficient resource scaling that meets our expectations — impressive.



As you can see, the orange area (reserved RAM) increases slowly, corresponding to the blue area (real usage) growth. That means no overcommitting or unnecessarily reserved memory.

Important: Aggressive Heap or Vertical Scaling

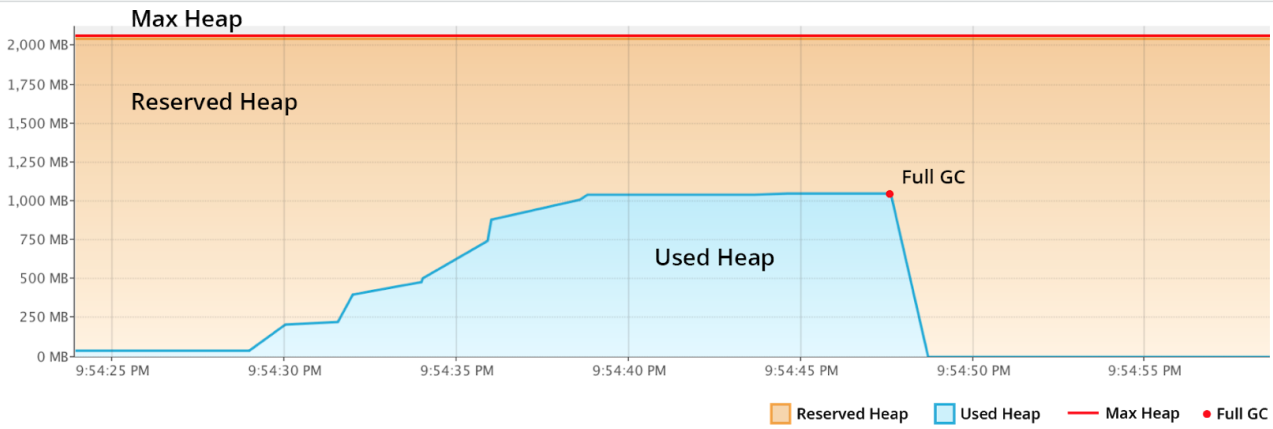
Popular JVM configurations for improving Java application performance can often impede the ability to efficiently scale vertically. So, you need to choose between the priorities of what attributes are most essential for your application.

One of many widely-used settings is the activation of Aggressive Heap in an attempt to make maximum use of physical memory for the heap. Let's analyze what happens while using this configuration.

```
java -XX:+UseG1GC -Xmx2g -Xms2g
```

or

```
java -XX:+UseG1GC -Xmx2g -XX:+AggressiveHeap
```



As we can see, Reserved Heap (orange) is constant and doesn't change throughout time, so there is no vertical scaling of the JVM in the container. Even if your application uses only a little part of available RAM (blue), the rest cannot be shared with other processes or other

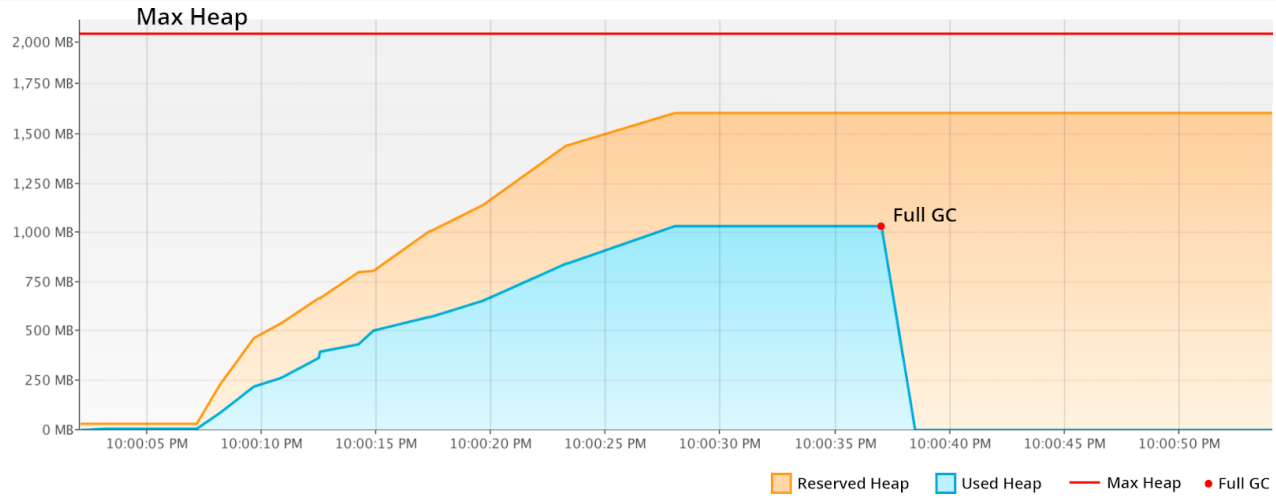
containers, as it's fully allocated for the JVM.

So if you want to scale your application vertically, make sure that Aggressive Heap is not enabled (the parameter should be `-XX:-AggressiveHeap`) and do not define -Xms as high as -Xmx (for example, do not state -Xmx2g -Xms2g).

Parallel Garbage Collector

Parallel is a high-throughput GC and is used by default in JDK8. At the same time, it does not suit memory shrinking, which makes it inappropriate for flexible vertical scaling. To confirm this, let's run a test with our sample application:

```
java -XX:+UseParallelGC -Xmx2g -Xms32m -jar app.jar 10
```



As we can see, the unused RAM is not released back to the OS. The JVM with Parallel GC keeps it forever, even disregarding the explicit Full GC calls.

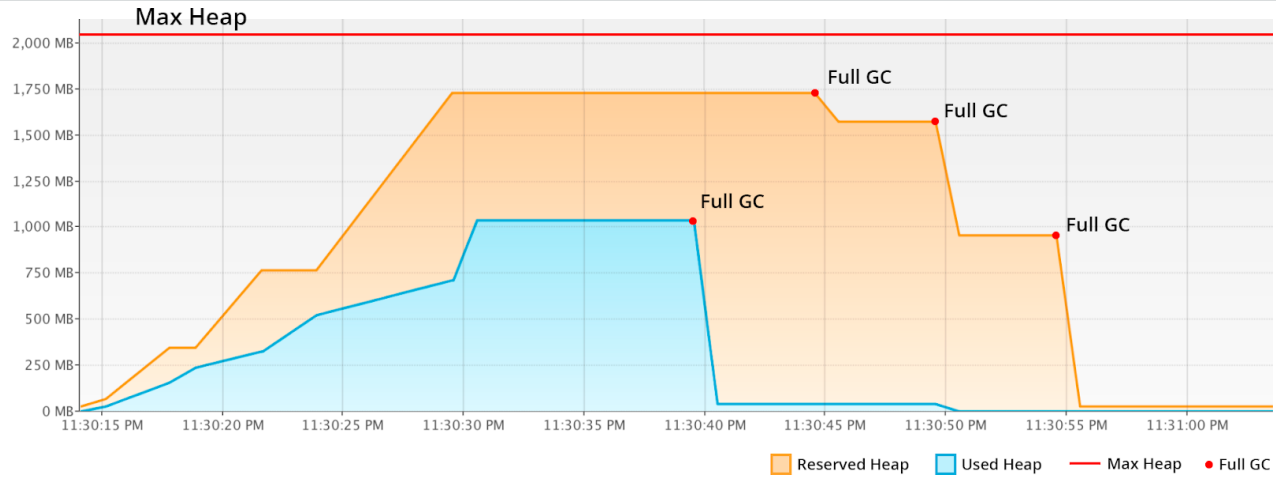
So if you want to benefit from vertical scaling according to the application load, change the Parallel to the shrinking GC available in your JDK. It will package all the live objects together, remove garbage objects, and uncommit and release unused memory back to the operating system.

Serial and ConcMarkSweep Garbage Collector

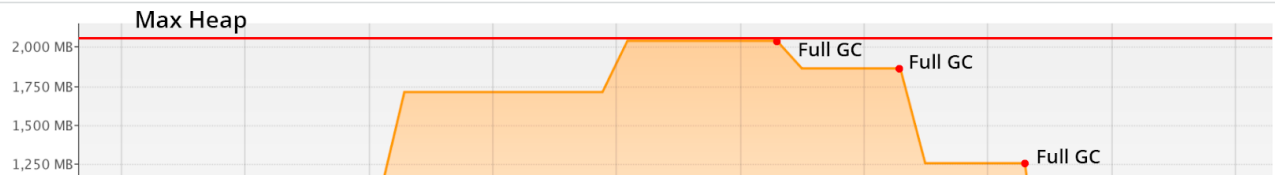
Serial and ConcMarkSweep are also shrinking garbage collectors and can scale memory usage in the JVM vertically. But in comparison to G1, they require four Full GC cycles to release all unused resources.

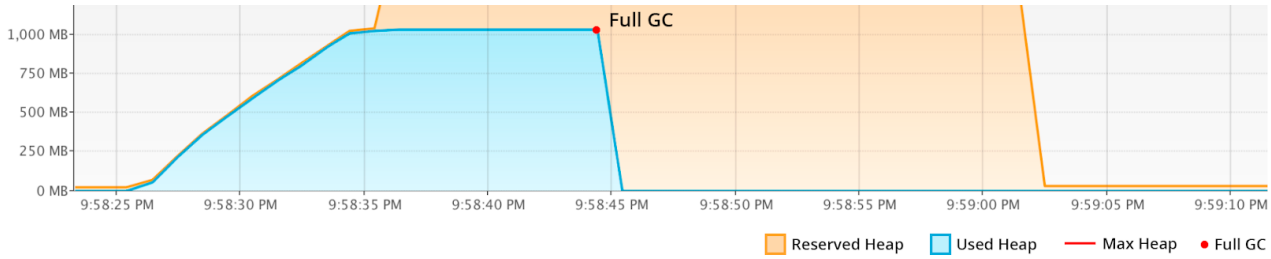
Let's see the results of the test for the both of these garbage collectors:

```
java-XX:+UseSerialGC -Xmx2g -Xms32m -jar app.jar 10
```




```
java -XX:+UseConcMarkSweepGC -Xmx2g -Xms32m -jar app.jar 10
```





Starting from JDK9, the releasing of memory can be sped up with the new JVM option -XX:-ShrinkHeapInSteps, which brings down committed RAM right after the first Full GC cycle.

Shenandoah Garbage Collector

Shenandoah is a rising star among garbage collectors that can already be considered as the best upcoming solution for vertical JVM scaling.

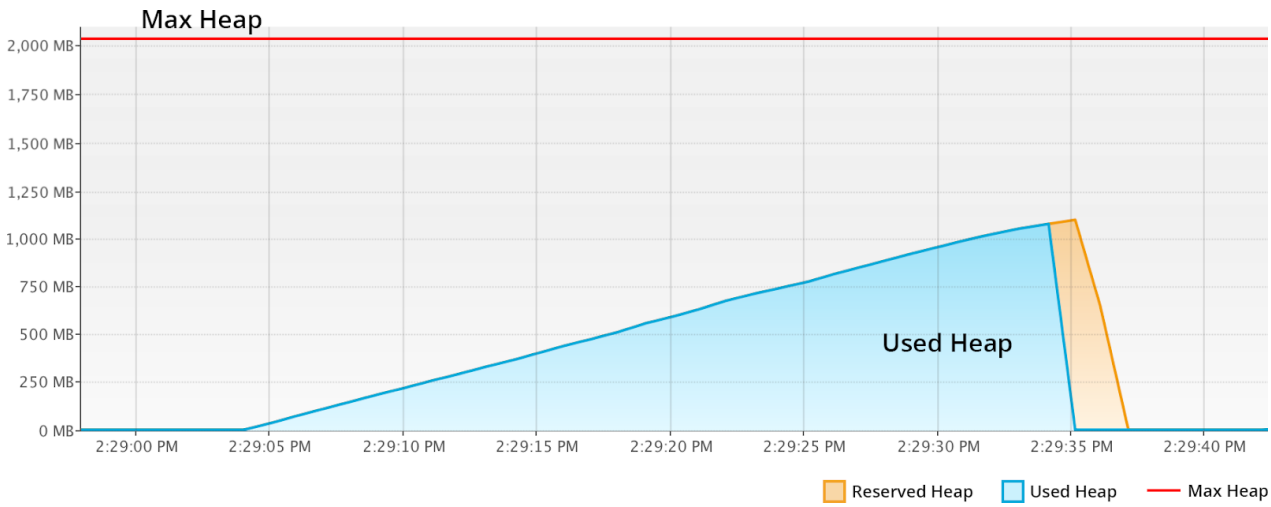
The main difference, compared to the others, is the ability to shrink (uncommit and release unused RAM to OS) asynchronously without needing to call a Full GC. Shenandoah can compact live objects, clean garbage, and release RAM back to the OS almost immediately after detecting free memory. And the possibility of omitting Full GC leads to eliminating related performance degradation.

Let's see how it works in practice:

```
java -XX:+UseShenandoahGC -Xmx2g -Xms32m -XX:+UnlockExperimentalVMOptions \
-XX:ShenandoahUncommitDelay=1000 -XX:ShenandoahGuaranteedGCInterval=10000 -jar app.jar 10
```

Here we added some extra parameters available in Shenandoah:

- -XX:+UnlockExperimentalVMOptions — needed to enable the uncommit option listed below.
- -XX:ShenandoahUncommitDelay=1000 — GC will start uncommitting memory that was not used for more than a designated time (in milliseconds). Note that making the delay too low may introduce allocation stalls when the application would like to get the memory back. In real-world deployments, making the delay larger than 1 second is recommended.
- -XX:ShenandoahGuaranteedGCInterval=10000 — guarantees a GC cycle within the stated interval (in milliseconds).



Shenandoah is very elastic and allocates only necessary resources. It also compacts used RAM (blue) and releases unconsumed reserved RAM (orange) back to the OS on the fly and without costly Full GC calls. Please note this GC is experimental, so your feedback about stability will be helpful for its creators.

Conclusion

Java keeps perfecting and adapting to always-changing demands. Currently, its RAM appetite is no longer a problem for microservices and cloud hosting with traditional applications, as there are already the right tools and ways to scale it properly, clean the garbage, and release resources for required processes. Being configured smartly, Java can be cost-effective for all ranges of projects — from cloud-native startups to legacy enterprise applications.
