# JVM Performance Tuning – Part III

Having covered the principal concepts in [Part I](http://donnachaforde.blogspot.com/2015/09/jvm-performance-tuning-part-i.html) where we covered Garbage Collection, and the Java Memory Model in [Part II](http://donnachaforde.blogspot.ca/2015/10/jvm-performance-tuning-part-ii-java.html) where we covered generational memory organization, we are now well placed to discuss Garbage Collectors.

## Garbage Collectors

The HotSpot JVM actually supports a number of different Garbage Collectors, representing different approaches or strategies to the task at hand. It’s an area under constant research and development so in some ways, the various GCs are representative of what is an evolutionary process. Some GCs have given way to others and anyone doing research in this area may find some of the names (especially for the older GCs) a tad confusing.

In this blog, I’ll make a stab at describing the main GCs. Again, as outlined in my first blog in the series, we’re talking about the Java HotSpot™ Virtual Machine, now provided by Oracle and formerly provided by Sun Microsystems.

### Serial Collector

The **Serial Collector** is the default GC so if you don’t ever explicitly configure your JVM, this is the GC that will run. It operates on both the New and Tenured Generations, performing both Minor and Major collections serially.

It’s a *‘Stop the World’* collector meaning that the application is paused while the GC is in effect.

Bear in mind though that the duration of these ‘pauses’ is relative short – they are typically very brief and indiscernible by the human-eye for the most part. Things start to become more evident as the GC comes under pressure to clear space when memory resources are tight. Under those circumstances, the time spent running the application versus the time spent in garbage collection starts to go against us and the pauses certainly become noticeable and performance is affected.

This concept of pausing an application to perform GC gives rise to the concept of ‘throughput’. This basically refers to the amount of time the VM gives to running the application versus running the GC.

As the name suggests, the approach of this GC is to garbage collect dead objects serially. It follows a Mark-Sweep-Compact algorithm, as follows:

* **Mark** – The collector identifies the live objects (i.e. those that have a valid root-reference still in use).
* **Sweep** – The collector identifies dead objects (a.k.a. garbage objects).
* **Compact** – The collector moves (also referred to as ‘slides’) live objects towards the start of the generation, to ensure large, contiguous blocks of memory are available for the next set of allocations.

Section 4 of the old Sun Microsystems Whitepaper on *Memory Management in the Java HotSpot™ Virtual Machine* does an excellent job of visualizing what’s going on here. It’s still available as a PDF document on the Oracle website [here](http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf).

### Incremental Collector

The **Incremental Collector**, also known as the **Train Collector** is an older GC that’s remained unchanged since Java 1.4 (i.e. J2SE 1.4.2) and is not supported in releases beyond Java 5. As such, it’s now redundant and I only mention it for the sake of completeness as you may see references to it in some of the older documentation.

In particular, it should not be confused with the CMS Incremental Mode GC, described below.

### Parallel Collector

The **Parallel Collector** is also known as the **Throughput Collector**. It takes advantage of the availability of multiple CPUs in hosts to perform multithreaded GC, which again likely reflects the evolution of hardware at the time.

(It might be worth reminding younger readers at this juncture that there was a time when PCs and Servers shipped with a single processor and ‘cores’ hadn’t yet become available.)

The Parallel GC is still a ‘Stop the World’ type GC but because it’s multithreaded, it doesn’t do so for as long as the Serial GC.

### Parallel Compacting Collector

The **Parallel Compacting GC** was introduced in Java 5 Update 6 with the view that it would eventually replace the **Parallel GC**. It introduced a new algorithm for Tenured Generation garbage collection though applied the same algorithm for the New Generation.

When you consider that, in a way, this is an enhancement of the Parallel GC then this illustrates how Garbage Collectors tend to evolve over time.

### CMS Collector

The **Concurrent Mark Sweep** (CMS) GC has been known by a few different names. It was initially known as the **Low Latency Collector** but in Java 5 became known as the **Concurrent Low Pause Collector**. In Java 6 the name changed again and it became known as the **Concurrent Collector**.

For the most part, it seems to be referred to as the **CMS Garbage Collector**. Regardless of the name changes, it was designed for *Fast Response Times* and *Shorter GC Pauses*. The general idea is that it performs GC in parallel with the application so it claims to not be a ‘Stop the World’ garbage collector or at least, minimizes these ‘pauses’.

The first thing to note about the CMS GC is that its garbage collection of the New Generation is the same as the Parallel GC. It’s the Tenured Generation garbage collection that is performed concurrently with the application. It does this by iterating over the objects in the Tenured Generation for an ‘Initial Mark’ and then iterating again for a ‘Re-mark’.

It’s considered suitable for applications with a larger Tenured Generation and for Interactive Applications (especially in Incremental Mode, described below).

As you might imagine, if an algorithm is going to perform GC in parallel with the application then, as its goals are somewhat orthogonal, its efficiency is somewhat hindered by the application. For example, as the GC iterates over objects in the Tenured Generation marking objects for deletion, the application could well be freeing more objects that should be also deleted in this iteration (i.e. floating garbage). It’s simply a trade-off. Technically, the GC cycle is arguably not as efficient but the upshot is that the application ‘throughput’ is better. And, you have to consider the bigger goal here – the GC really only has to be good enough to keep the application afloat and must do so without hindering the performance of the application.

There are other implications to note:

* The CMS GC is non-compacting meaning that it saves time by skipping this step. The downside of this is that the heap is not contiguous, which means that memory allocations can be more expensive and in turn, this can impact the performance of GC collection in the New Generation.
* The CMS GC requires a larger heap because objects continue to be allocated during the GC cycle and, as described above, the approach will tend to leave ‘floating garbage’.
* The CMS GC has its own space requirements so it has to pre-empt the GC trigger in order to ensure there is sufficient room for it to grab this space so it can function.

#### CMS Incremental Model

The CMS Collector can be run in **Incremental Mode**, which means that the Concurrent Phases are done incrementally. This is intended to lessen the impact of long concurrent phases, which it periodically stops in order to yield to the application.

### Garbage First Collector - G1

This is a more recent garbage collector, introduced in Java 7. Technically, it was available in ‘experimental mode’ in later releases of Java 6 and is intended as the long-term replacement of the CMS collector.

The G1 GC is a ‘server’ styled garbage collector that takes advantage of the fact that server hardware tends to be more capable than desktop, laptop or tablet. It performs heap compaction and promises more predictable ‘pause time’ than CMS.

The G1 documentation signals that the JVM is likely to have a larger JVM process size due to the fact that this GC takes a completely different approach to managing its datasets. It seems this GC is intended for use with larger applications as it’s recommended for use with Heap Sizes from **6 GB** up.

Oracle have a dedicated tutorial on their website documenting the G1 GC [here](http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html).

## JVM Ergonomics

From Java 5, the JVM introduced a different approach to tuning by allowing the user to specify target goals. As mentioned above, this is a feature of the Parallel Collector GC. The basic objective here was to provide good performance with little-to-no manual tuning.

The user specifies either the:

* Max Pause Time – the maximum amount of time the application can be paused while GC is performed
  + configured using -XX:MaxGCPauseMillis=nnn
* Desired ‘Throughput’ - Measure of time spent in Application time and not performing GC.
  + configured using -XX:GCTimeRatio=nnn

The JVM has an implicit secondary goal which is to manage the memory footprint – i.e. GC tries to reduce the size of the heap.

The JVM prioritizes these goals as follows:

1. Achieve the ‘max pause time’.
2. Achieve the desired ‘throughput’.
3. Minimize memory footprint.

Some of the default behaviour with this GC is associated with ‘server-class’, which is a term Java used to make some determinations about the host it was running on. This gets a little involved because the determination depended on what type of OS that Java was running on. You can read more about Ergonomics as introduced in Java 5 [here](http://www.oracle.com/technetwork/java/ergo5-140223.html) or in a more recent version of the tuning guide [here](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html).

## Java JRE Client and Server Versions

One little known aspect about Sun/Oracle Java is that the standard install comes with two versions of the JRE – i.e. client and server versions. As the name suggests, these are intended for different types of applications and the main differences seems to centre on how the Just-in-Time (JIT) compiler behaves.

The details are set out Chapter 2 of the [Java Hotspot Performance Engine Architecture](http://www.oracle.com/technetwork/java/whitepaper-135217.html#2), as discussed in this [StackOverflow](http://stackoverflow.com/questions/198577/real-differences-between-java-server-and-java-client) discussion thread.

## Explicit Garbage Collection

Nowadays, this is probably considered something of a legacy, associated with much earlier versions of Java (i.e. pre-Java 1.4), and is arguably less appropriate with later versions as it can adversely affect performance.

Basically, there’s a programmer directive invoked by calling:

System.gc()

This call ‘advises’ rather than instructs the JVM to start a GC cycle. It was used to allow the programmer give a hint to the JVM after they’d released a large chunk of heap memory.

Nowadays, it’s unnecessary and probably shouldn’t be present in any of your Java code. After all, if you’re reading this, then you’re trying to tune your JVM using options and this may just get in the way. The best thing all-round is to simply disable it, which you can do with the following JVM option.

|  |
| --- |
| Disable Explicit Garbage Collection |
| -XX:-DisableExplicitGC |

## JVM Options

Here, I’ve collated the various JVM switch options for selecting and configuring the Garbage Collector.

The **Serial GC** can be explicitly enabled using the following JVM argument.

|  |
| --- |
| Explicitly enable Serial GC |
| -XX:+UseSerialGC |

The **Parallel GC** options are as follows:

|  |
| --- |
| Options for enabling Parallel GC |
| -XX:+UseParallelGC |
| -XX:UseParallelOldGC |
| -XX:+UseParNewGC |

You can further configure now many threads you’d like the GC to use, as follows:

|  |
| --- |
| JVM Switch |
| -XX:ParallelGCThreads=n |

As mentioned above, the Parallel GC supports **Ergonomics Goals**, which basically means you can give the JVM a target goal, as follows:

|  |
| --- |
| Set Ergonomics Goal |
| -XX:MaxPauseTimeMillis |
| -XX:GCTimeRatio=n |

Here you can see instructions to either perform the GC within a certain amount of time or effectively as a percentage of the ‘Throughput’.

The **CMS GC** can be enabled as follows:

|  |
| --- |
| Enable Concurrent Mark Sweep GC |
| -XX:+UseConcMarkSweepGC |

Incremental Mode CMS can be enabled as follows:

|  |
| --- |
| Enable Incremental CMS |
| -XX:+CMSIncrementalMode |

In this mode, you have much finer grained control over the GC cycle.

|  |
| --- |
| Configuring CMS Incremental Mode |
| -XX:+CMSIncrementalPacing |
| -XX:CMSIncrementalDutyCycleMin=n |
| -XX:CMSIncrementalDutyCycle=n |
| -XX:+UseCMSCompactAtFullCollection |
| -XX:CMSIncrementalSafetyFactor=n |
| -XX:CMSExpAvgFactor=n |

The G1 Collector can be enabled as follows:

|  |
| --- |
| Enable the G1 GC |
| -XX:+UseG1GC |

A more complete set of JVM options is published on the Oracle site [here](http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html).

In my next blog in this series, I will discuss Heap Configuration, which goes *‘hand-in-glove’* with your GC tuning, Stack Size and the JIT compiler options. In practice, if you’re going to tune the GC, you’ll need to have a handle on these concepts as at times, one has implications on the other and you’ll need to understand these. Otherwise, you may well find your JVM rejecting your choice of options.

Donnacha Forde  
<https://ie.linkedin.com/in/donnachaforde>

References  
<http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html>

<http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>

<http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>

<http://www.oracle.com/technetwork/java/ergo5-140223.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html>

<http://www.oracle.com/technetwork/java/whitepaper-135217.html>