Optimizing memory use in Java applications, garbage collectors

Ștefan PREDA
Oracle, Bucharest, Romania
spreda2003@yahoo.com

Java applications are diverse, depending by use case, exist application that use small amount of memory till application that use huge amount, tens or hundreds of gigabits. Java Virtual Machine is designed to automatically manage memory for applications. Even in this case due diversity of hardware, software that coexist on the same system and applications itself, these automatic decision need to be accompanied by developer or system administrator to triage optimal memory use. After developer big role to write optimum code from memory allocation perspective , optimizing memory use at Java Virtual Machine and application level become in last year's one of the most important task. This is explained in special due increased demand in applications scalability.

Keywords*: Java Virtual Machine, garbage collector, Concurrent Mark Sweep (CMS), G1 GC, Shenandoah an Ultra-Low-Pause-Time Garbage Collector.*

1 Introduction

Java applications run in a Java Virtual Machine (JVM or Java HotSpot VM). Memory management is assured automatically based on a set of rules, grouped in a set named garbage collector (GC). In Java Virtual Machine can run a variety of applications from small applets that run in browser through web-services that responds to huge number of requests per time unit, and which runs on big server. From this reason Java Virtual Machine has by default many garbage collectors, each one appropriate to a specific application type.

Java Virtual Machine selects automatically a garbage collector based on application and on a hardware on which is running. Often, chosen GC may not be suitable for application performance, and in such cases developer need to select a specific GC and to make supplementary tuning.

One target in memory optimization is to provide good JVM performance with minimum parameter tuning, this feature is named ergonomics. Using ergonomics JVM try in specified criteria from above to select for application, one of the best: *garbage collector*, *heap* size and *run-time compiler*.

Java Heap is a memory area created by Java Virtual Machine at start-up. From heap is allocated memory to all instances of objects and to arrays allocated. Heap is shared between threads that run inside Java Virtual Machine.

Heap can be fixed size or variable size, means it can be expanded based on application demand. Initial heap size is controlled by option -Xms. Maximum java heap size is controlled by option -Xmx.

2 Garbage collector

Objects are not explicitly released and reallocated from Heap, this task is performed automatically by GC. A simple description for GC [3] is that it searches for objects that are not in use and free memory occupied by those. Free space is used further to allocate other objects. After releasing unused objects, memory space can become fragmented, from this reason allocation for big objects can be a problem. To overcome this, GC do compaction after memory free. All GC do: search for unused objects, free space and compact it. Differentiation between GC's consists in approach on how those operations are performed. Like a basic principle garbage collection is based on fact that when a previously allocated memory is no referenced by any pointers it can be reclaimed for new use. GC find and recycle those memory locations.

GC work is multiple threads and usual application is multi threads, when moving objects for GC a challenge is to move objects while no application access to those is. From this reason sometime there are *pauses* when all application threads are stopped (and GC do his job of moving objects) and from here one of the most important aspect of GC tuning: minimize pauses. Pauses are named usual as "stop-the-world pauses".

Generations in garbage collectors. Some Java objects are used for short time, other for some more time and there may exist objects that are used for long time, thus heap is organized in areas named generations. We have:

- old (*tenured*) generation
- *young* generation
- *eden*
- *survivor* spaces

A Java application usual have many temporary objects, thus those are stored in young generation. According with [1] Objects in general are allocated initial in young generation, more exactly in eden space. Survivor space is empty always, it serve only like a destination from where object will be moved to a next place (or to the tenured space). When young generation is filled, garbage collector stop (pause) application threads, objects that are not in use from young generation are discarded and those that are still in use are moved to other place. Operation is named "minor GC", term minor is because operation is fast due fact that young generation represent only a part of entire heap area, pause will be shorter than situation when garbage collector work with entire heap.

3 Garbage collector types

Java Virtual Machine depending by necessary application scalability and hardware can use one of the following collectors:

- *Serial collector*
- *Parallel collector*
- *Mostly concurrent collector*
- *Concurrent Mark Sweep (CMS) Collector*
- *Garbage-First Garbage Collector (G1 GC)*
- *Shenandoah: An Ultra-Low-Pause-Time Garbage Collector*

Serial collector. It use a single thread to perform memory management task, this collector is good for applications with small data set (less than 100Mb), having only one thread, managing inter – threads communications does not apply, from this reason it is relative efficient for small applications. It is good for machines also with a single processor. Like drawback it is not suitable for multiprocessor hardware due single thread work. Collector is selected automatically on specific hardware like single CPU systems, or it can be selected automatically using
-XX:+UseSerialGC option.

Parallel collector or throughput collector performs many minor collections in parallel, reducing Garbage Collection overhead. Parallel collector is suitable and is automatically selected if hardware is multiprocessor, multithreaded and or application is medium or large. We can choose parallel collector explicit starting Java Virtual Machine with option:
-XX:+UseParallelGC.
Once used parallel collector it enable implicit *parallel compacting*. Running parallel compacting, major collections are also multi thread. Parallel compactation can be disabled using option:
-XX:-UseParallelOldGC.
Very important, this is not recommended because it make major collection single thread which is not suitable for medium-large applications. If we need good performance and garbage collector pauses around 1 seconds are acceptable, then parallel collector is suitable.

Mostly concurrent collector, it make GC tasks simultaneous (or concurrent) while application is running, resulting in very short GC pauses. This collector is for big applications or for medium applications that need fast response. In Java

Development Kit (JDK) 8 we have two kind of mostly concurrent collectors:
- *Concurrent Mark Sweep (CMS) Collector*
- *Garbage-First Garbage Collector (G1 GC)*

Concurrent Mark Sweep (CMS) Collector. This collector has minor and major collections. Major collections are performed using separate threads to track heap objects concurrently with application execution threads resulting thus small pauses time. During a major collection there is a small pause at the beginning (all applications threads are paused, this pause is named *initial mark pause*), and a second pause, a little longer, near the middle of the collection time (this is named *remark pause*). Running concurrently GC threads and applications threads, application throughput may decrease because part of the CPU's threads are used by GC in detriment of application threads, thus CMS is suitable for multi processor and multi threads architectures.
Running garbage collector threads simultaneously with application threads is needed such as collection of tenured objects to finish before it become full. If this is not happening we have *concurrent mode failure*, in this case application threads are paused till collection is completed. Concurrent mode failure event is a sign that GC CMS parameters need to be changed or tuned.
Time spent with garbage collection, ideal should be as small as possible, while percent of heap recovered need to be as much as possible. When more than 98% time is spent with garbage collection, but less than 2% heap size is recovered, then *OutOfMemoryError* is thrown.
This can be disabled using option:
-XX:-UseGCOverheadLimit.
This is not recommended because mentioned percents 98%, 2% usual show that application is running without progressing and this need to be fixed by tuning. Usually long time spent with garbage collector is associated with concurrent mode failure events. Concurrent collection in CMS start when tenured generation increase over 92%. This threshold depend by JDK release, value can be adjusted using option
XX:CMSInitiatingOccupancyFraction=<percent>
CMS collector use one or more CPU during object tracing process, also one CPU is used during concurrent sweep phase, CPU is not released voluntary to the application, this can influence application throughput and response time. To solve this problem CMS have incremental mode or *i-cms mode*, this mode break up concurrent phases in short bursts scheduled at midway of minor pauses.
CMS voluntary release CPU to application after a percentage of time between young generation collections, this percent is named *duty cycle*. Midway is default, this can be changed using option:
XX:CMSIncrementalOffset=<N>
To enable incremental mode we use option: -XX:+CMSIncrementalMode
Sample combination of options for CMS:

```
-XX:+UseConcMarkSweepGC \
 -XX:+CMSIncrementalMode \
-XX:+PrintGCDetails \
-XX:+PrintGCTimeStamps
```

Here option UseConcMarkSweepGC enable CMS, CMSIncrementalMode enable i-cms, PrintGCDetails and PrintGCTimeStamps print GC activity details for troubleshooting purposes.
Garbage-First Garbage Collector (G1 GC). This collector is suitable for programs that have very large heap which run on multiprocessor servers. The challenge for memory management in case of applications that require very large amount of memory is that garbage collector heap operations can take big time while heap increase, is such situation interruptions can become proportional with heap or data size.
To overcome this G1 GC use similar technique like CMS, i.e. garbage collector heap operations threads are performed concurrently with application threads, and

supplementary G1 GC use *heap partitioning*.

Heap is partitioned in equal contiguous memory regions. G1 GC concurrently checks each regions marking in this way if objects are still live. After this phase G1 collect regions that are empty or almost empty resulting a big free space. As collector concentrate first on doing collection and compaction on regions that are almost full with garbage objects it is named "Garbage-First" or G1.

To fulfil pause time demanded by application, G1 GC use a prediction model to select that regions and number of those such as resulting pause do not increase over imposed application limit.

After marking regions, collector copies objects from selected regions to a single heap region and in the same time compact it, reduce fragmentation and free up space, doing this simultaneous, result is a decreased pause time and better throughput. As G1 GC use a prediction model, *it is not a real-time collector*, means pause time target is not strictly fulfilled, it is just realized with a high probability.

Probability is accurate. G1 collector can provide a good memory management for applications which need large heap and small garbage collector latency, for example a 6 GB application can run with a estimated pause time under 0.5 seconds.

Based on [1] CMS and G1 GC collectors are comparable and specific applications can benefit of both. Both collectors are good for applications that have traits like:

- more than 50% heap is occupied by data
- object allocation or promotion rate vary in time
- applications is experiencing long GC or compactation pauses (between 0.5 – 1 second)

Because G1 is a compacting collector and G1 predicted pauses are very good predictable, according with [1] in the future intention is that G1 GC will replace CMS.

G1 collections are running usual simultaneous with application, due this exist probability that application allocate object faster that GC relocate and free space, this event is named "*Allocation (Evacuation) Failure*" it is very similar with CMS "Concurrent Mode Failure". When "Allocation (Evacuation) Failure" is happening there can be no space for application to allocate live objects, in this case GC G1 will start a full GC collection.

An object can die during G1 collection, and thus not be collected, that can result in improper space released. To prevent this G1 GC consider that any object that is live at start of concurrent marking is considered live for collection (this technique is named *snapshot-at-the-beginning (SATB)*). SATB allow floating garbage similar with CMS incremental.

G1 GC keeps information about old generations pointers to young generation objects in a data structure named a *remembered set.* A particular kind of remembered set is named *card table*, which is an array of bytes. Each byte is referred as a *card* that corresponds to a range of heap addresses. When such byte is changed such as to contain a new pointer from the old generation to the young generation, operation is named "*dirtying a card*", and value of changed byte is named "dirty value".

Having "old generation to young generation pointer" information GC G1 can do something with this information, for example transferring it to other data structure, operation is named "Processing a card".

For GC G1 a concurrent marking phase (marking all live objects from heap) when heap is occupied over a specific percent.

This is controlled by a parameter named InitiatingHeapOccupancyPercent means by option:

XX:InitiatingHeapOccupancyPercent=<N N>

By default *InitiatingHeapOccupancyPercent is 45.*

The other important parameters of GC G1 are:
MaxGCPauseMillis which represent a maximum pause time accepted and *GCPauseIntervalMillis* which is time interval during which pause can occur.

Shenandoah: An Ultra-Low-Pause-Time Garbage Collector. Just looking in previous discussion about different garbage collectors, evolution was tight related to hardware complexity.
The last evolution in hardware is big machine, like Exadata or Exalogic for example, with multi-core machines which should run applications with very large heaps (about 100 GB, as mentioned in [4]) This evolution conduct to a new garbage collector: *Shenandoah* Ultra-Low-Pause-Time Garbage Collector. *Shenandoah* garbage collector is still in draft according with [5]. Shenandoah is designed to manage applications that have over 100GB heaps with pause less than 10ms.
Shenandoah garbage collector is similar with G1 GC, a “region-based” collector, it works also in phases. First is marking phase when all live objects from heap are marked, a count also of live objects in every region is maintained. In second phase, is similar like in G1 collector, evacuation phase where live objects from best regions to collect are copied to new regions. Then follow concurrent marking phase and then phase where evacuated regions are reclaimed *concurrently*.
Concurrent evacuation is based on fact that application threads and garbage collector threads know and agree about the location of objects. To achieve this Shenandoah garbage collector use “b*rooks forwarding pointer*”. Application threads reads are accomplished indirectly via forwarding pointer. Writes of objects in targeted regions copy objects and then writes those in new location. Forwarding pointer is main difference compared with G1 GC. Shenandoah using forwarding pointer is focused on working with regions with most garbage regardless by age (it is not focused on young generation like other collectors).
Referring to huge systems, similar collector with Shenandoah is *Zing/Azul collector* [6]. This collector have similar features: is a predictable garbage collector, have comparable response time with Shenandoah, scale to huge heap sizes (hundreds of GB's), good application scalability.

4. About garbage collectors in embedded or limited devices.

According with [7] embedded devices have limited resources thus garbage collectors are rarely used! Nevertheless is very important to mention that such kind of devices that use Java Platform, Micro Edition are embedded software devices and use Garbage Colector, usually garbage collector from Java ME is a serial collector.
Initial using garbage collector in embedded devices was introduced very slow because Garbage Collection come with a performance cost, garbage collector use between 30-150% more address space than a classic memory management algorithm. Also garbage collector can sometime, on embedded or limited devices, to increase high water mark of memory usage and also to lead to high CPU usage. This was one reason why garbage collector was not introduced early on mobile platforms.
Based on [8] new embedded JVM handle garbage collector more efficiently, main improvement area is to perform garbage collection faster. For this is used a “*hybrid garbage collection approach*”, thus memory is divided in multiple regions, there will be frequent generation garbage collections in “nursery” followed by mark-and-sweep stage over all regions. garbage collector needs is anticipated using heuristics algorithms. Garbage collector can be tailored according with application needs.

5 Conclusions.

Memory optimization in java application is a complex problem which depends by many factors: hardware, CPU, memory, application itself, connectivity, etc. Garbage collector try to make this optimization automatically. JVM have many garbage collectors available, based on the application and on hardware system, a specific garbage collector is selected automatically for the application.

This automatic selection simplifies considerable programmer and system administrator work. Nevertheless it is only a first steps, every garbage collector can require parameter tuning, a simple explanation is that even if the systems are identical like hardware, applications aren't, and even if we will presume that we have the same applications on two systems, there may exist different other applications that are running, or there will be differences, which explain why those garbage collector will beehive different.

Apart from tuning we mention that the main task still remain for programmers: means to write code that consume as less memory as possible. Garbage collector evolution was in to main directions first to assure scalability on hardware that become more and more complex like CPU and memory resources. The second, a new direction in last year's is to include also garbage collector in embedded or limited devices.

References

[1] Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/

[2] The Java® Language Specification. Java SE 8 Edition https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[3] Java Performance: The Definitive Guide, By: Scott Oaks, Publisher: O'Reilly Media, Inc.
Pub. Date: April 21, 2014, Print ISBN-13: 978-1-4493-5845-7

[4] GC Algorithms: Implementations https://plumbr.eu/handbook/garbage-collection-algorithms-implementations#shenandoah

[5] OpenJDK JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector http://openjdk.java.net/jeps/189

[6] Pauseless Garbage Collection for Java https://www.azul.com/products/zing/pgc/

[7] Garbage collection (computer science) https://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29

[8] Real-Time Garbage Collection Speeds Embedded Java http://electronicdesign.com/embedded/real-time-garbage-collection-speeds-embedded-java

Stefan PREDA graduated the Faculty of Economic Cybernetics, Statistics and Informatics, with a bachelor degree in Economic Informatics in 2013. In 2015 he got his master degree from the same faculty of the Bucharest University of Economic Studies, specialization in Databases Support for Business. Currently he is working on Oracle Corporation, like Principal Technical Support Engineer, software analyst, in Fusion Middleware, EMEA Identity Management team.