GC Performance Tuning

Part I - Introduction

Thomas Pliakas

Agenda

- What is Garbage Collection ?
- Garbage Collection in Java
- GC Algorithms in JVM
- GC Tunings - Basics
- References

What is Garbage Collection ?

- Manual Memory Management
  - Use “malloc” & “free”
  - **Always to remember to free the memory**
- Automated way to track down all the objects that are still used and mark the rest as “garbage”.
  - At runtime - always to identify which memory is no longer use and free it
  - First introduced in “Lisp @1995”
  - Mark & Sweep
    - Garbage Collection Roots (Local variables, Active threads, Static fields, JNI references).
    - Marking - walking through all reachable objects starting from GC roots and keeping the ledger in native memory about all such objects.
    - Sweeping is making sure the memory addresses occupied by non-reachable objects can be reused by the next allocations.

GC in Java - Memory Pools

*In Java 8, Permgen memory area is replaced by Metaspace

Why Generational Memory Spaces

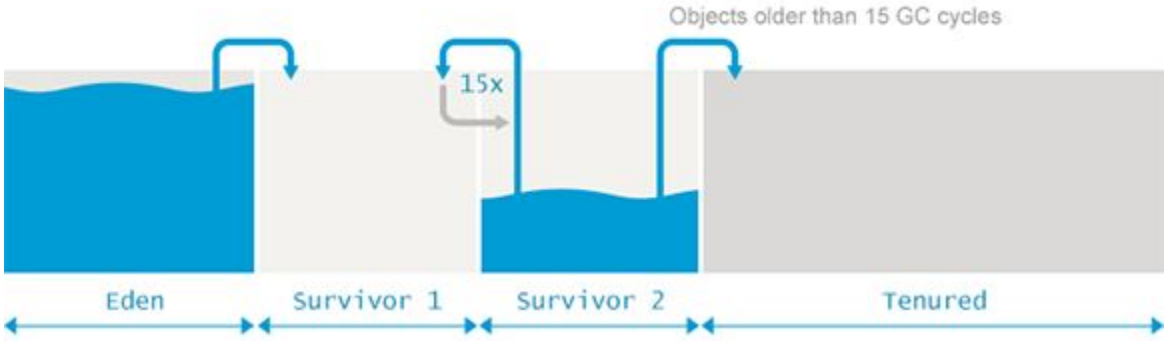
The memory inside the VM can be divided into **Young Generation** & **Old Generation.**

Young Generation - Eden Space

- Eden Space
  - Memory region where the objects are typically allocated during their creation
  - Divided into one or more Thread Local Allocation Buffers (TLAB)
    - Most objects of one thread are allocated in the corresponding TLAB in order to avoid expensive synchronization with other threads.
  - When allocation is not possible to TLAB, then allocation is happening to a shared Eden space.

Young Generation - Survivor Space

Two survivor spaces called “from” and “to”



Always one of the two Survivor spaces is always empty

To determine whether the object is old to be considered ready to be promoted to “Tenured” space, GC tracks the number of collection cycles that this object “survived”. This threshold is called “Tenuring Threshold”.

- **-XX:+MaxTenuringThreshold** - sets the upper limit
  - Default 15
  - 0 results to immediate promotion without copying.

Old Generation

- Base characteristics
  - Much more complex algorithms
  - Significant larger space
  - GC collections happen less frequently
  - No mark and copy actions are happening .. (long lived objects)
- GC Algorithms base characteristics
  - Mark reachable objects by setted the marked bit next to all objects accessible through GC roots.
  - Delete all unreachable objects
  - Compact the content of old space

Minor GC / Major GC / Full GC

- Minor GC
  - Happens to Young Generation
  - Always triggered when the JVM is unable to allocate space for a new object.
    - Higher the allocation rate -> more frequently minor GCs
  - Minor GC triggers Stop-the-World (STP) pauses:
    - Tenured Generation is ignored.
- Major GC & Full GC
  - Major GC cleans up Old Generation
  - Full GC cleans up the complete Heap

JVM GC Algorithms

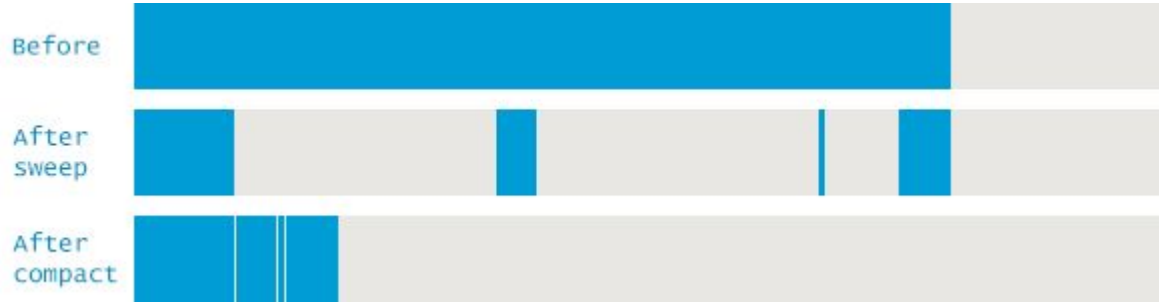
Young	Tenured	JVM Options
Serial	Serial	-XX:+UseSerialGC
Parallel Scavenge	Parallel Old	-XX:+UseParallelGC -XX:+UseParallelOldGC
Parallel New	CMS	-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
G1	-	--XX:+UseG1GC

Serial GC

Single Threaded (STW)

Young Generation - Mark and Copy approach

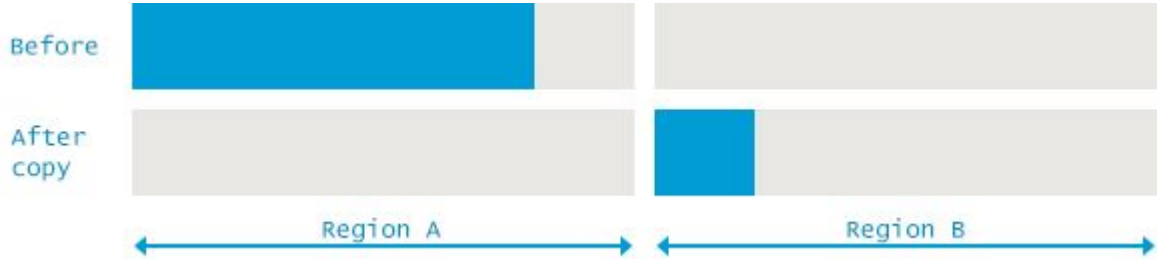
Old Generation - Mark - Sweep - Compact approach



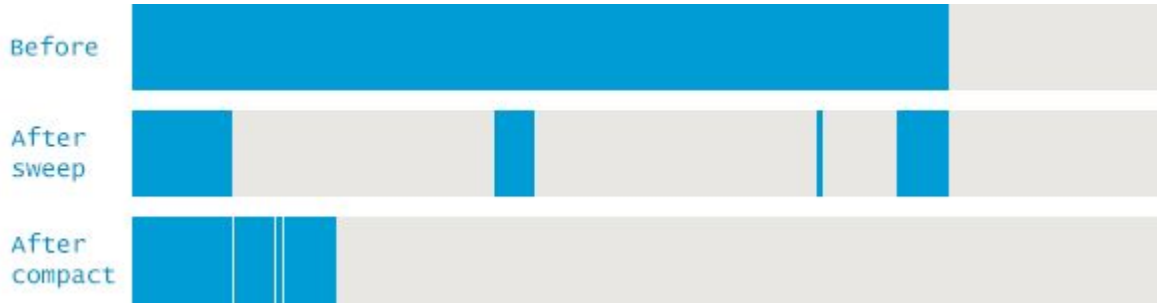
Parallel GC

Multi Threaded (STW)
-XX:+ParallelGCThreads=NN

Young Generation - Mark and Copy approach



Old Generation - Mark - Sweep - Compact approach



Concurrent Mark and Sweep (No compaction)

Young Generation - Mark and Copy approach

Old Generation - Mark - Sweep

<u>Avoid Long Pauses - No compaction</u>

- Uses free-lists to manage reclaimed space
- Mark-and-Sweep phases happened concurrently with the application

CMS - Multi Phases

- Phase 1 : Initial Mark
  - STW event to collect all Garbage Collection Roots
- Phase 2: Concurrent Mark
  - Traverses Old Generations and marks live objects starting from roots.
  - *Runs concurrently with your application threads*
- Phase 3: Concurrent Preclean
  - Runs in parallel with application threads
  - Objects that have their fields changed are marked by JVM as **dirty**. These objects are accounted for and marked as live.
  - Preparation phase for Final Remark
- Phase 4: Final Remark
  - STW event to finalize all live objects in the Old Generation (similar to Initial Mark).
- Phase 5: Concurrent Sweep
  - Concurrently with the application
  - Removes unused objects and reclaim space.

CMS does a great job at reducing the pause duration by offloading a lot of work to concurrent threads. Drawbacks is fragmentation and no predictable in pause durations. Designed for small heaps, low latency applications.

G1 - Garbage First

Duration & Distribution of the STW pauses **predictable** & **configurable**

The heap is split into a number (2048) smaller *heap regions* that can house objects. Each region may be an Eden region, a Survivor region or an Old region. The logical union of all Eden and Survivor regions is the Young Generation, and all the Old regions put together is the Old Generation

GC avoids collecting the entire heap at once, and instead approach the problem *incrementally*: only a subset of the regions, called the *collection set* will be considered at a time. ***All the Young regions are collected during each pause, but some Old regions may be included***

During the concurrent phase it estimates the amount of live data that each region contains. This is used in building the collection set: the regions that contain the most garbage are collected first.

G1 Key Characteristics

Evacuation Pause: Fully Young

When the Young Generation fills up, the application threads are stopped, and the live data inside the Young regions is copied to Survivor regions, or any free regions that thereby become Survivor.

Concurrent Marking

G1 Concurrent Marking uses the **Snapshot-At-The-Beginning** technique that marks all the objects that were live at the beginning of the marking cycle. The information on which objects are live allows to build up the liveness stats for each region so that the collection set could be efficiently chosen afterwards. (Initial Mark, Root Region Scan, Concurrent Mark, Remark, Cleanup)

Evacuation Paused: Mixed

G1 will schedule a mixed collection that will not only get the garbage away from the young regions, but also throw in a bunch of Old regions to the collection set. The exact number of Old regions to be added to the collection set, and the order in which they are added, is also selected based on a number of rules. These include the soft real-time performance goals specified for the application, the liveness and gc efficiency data collected during concurrent marking, and a number of configurable JVM options. (More details in Part II)

GCG1 Tunings

Here are a couple of command line for tuning G1GC.

- -XX:G1HeapRegionSize=n – a power of 2 value between 1M and 32M. Approximately, there should be around 2048 regions on the minimum Java heap size.
- -XX:MaxGCPauseMillis=200 – how long can a GC pause be. This is a suggestion, though G1GC will try to make the pauses shorter than the value. The default values is 200ms.
- -XX:G1HeapWastePercent=10 – how much space can be wasted in a region. The larger the waste, the faster the GC will allocate objects in the new region, rather than try to save these bits of space wasted.

GC Tuning - Goals

- Set Performance Goals in the the following areas:
  - Latency
  - Throughput
  - Capacity
- Three different goals in mind
  - Making sure the worst-case GC pause does not exceed a predetermined threshold
  - Making sure the total time during which application threads are stopped does not exceed a predetermined threshold
  - Reducing infrastructure costs while making sure we can still achieve reasonable latency and/or throughput targets

GC Tuning - High Allocation Rate

- The amount of memory allocated per time unit. Often it is expressed in MB/sec.
- The allocation rate can be calculated as the difference between the sizes of the young generation after the completion of the last collection and before the start of the next one.

```
0.291: [GC (Allocation Failure) [PSYoungGen: 33280K->5088K(38400K)] 33280K->24360K(125952K), 0.0365286 secs]
[Times: user=0.11 sys=0.02, real=0.04 secs]
0.446: [GC (Allocation Failure) [PSYoungGen: 38368K->5120K(71680K)] 57640K->46240K(159232K), 0.0456796 secs]
[Times: user=0.15 sys=0.02, real=0.04 secs]
0.829: [GC (Allocation Failure) [PSYoungGen: 71680K->5120K(71680K)] 112800K->81912K(159232K), 0.0861795 secs]
[Times: user=0.23 sys=0.03, real=0.09 secs]
```

-XX:NewSize -XX:MaxNewSize & -XX:SurvivorRatio

Premature Promotion

Situation where objects with a short life expectancy are not collected in the young generation and get promoted to the old generation, is called **premature promotion.**

```
0.291: [GC (Allocation Failure) [PSYoungGen: 33280K->5088K(38400K)] 33280K->24360K(125952K), 0.0365286 secs]
[Times: user=0.11 sys=0.02, real=0.04 secs]
0.446: [GC (Allocation Failure) [PSYoungGen: 38368K->5120K(71680K)] 57640K->46240K(159232K), 0.0456796 secs]
[Times: user=0.15 sys=0.02, real=0.04 secs]
0.829: [GC (Allocation Failure) [PSYoungGen: 71680K->5120K(71680K)] 112800K->81912K(159232K), 0.0861795
secs] [Times: user=0.23 sys=0.03, real=0.09 secs]
```

- ***The application goes through frequent full GC runs over a short period of time.***
- ***The old generation consumption after each full GC is low, often under 10-20% of the total size of the old generation.***
- ***Facing the promotion rate approaching the allocation rate.***

[Hint] - How to measure it ?
Extract the size of the Young Generation and the total heap both before and after the collection event. Knowing the consumption of the young generation and the total heap, it is easy to calculate the consumption of the old generation as just the delta between the two.

References

- GC Confession of a performance Engineer
  - https://zeroturnaround.com/rebellabs/gc-tuning-confessions-of-a-performance-engineer-by-monica-beckwith/?utm_source=twitter&utm_medium=social&utm_campaign=rebellabs_vjug
- Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide
  - https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html
- Garbage Collectors Log analyzers
  - http://fasterj.com/tools/gcloganalysers.shtml
- Java Garbage Collection handbook
  - https://plumbr.eu/java-garbage-collection-handbook
- G1GC Concept and Performance Tuning
  - https://blogs.oracle.com/g1gc/