Performance Tuning JVM - Concepts, Case Studies, Practical Implementations for Oracle hotspot JVM

by

Kranthi Paidi, Performance Architect, Cigniti Technologies

Copyright: STeP-IN Forum
Published with permission for restricted use in ‘STeP-IN SUMMIT 2014’ in agreement with full copyrights from owner(s) / author(s) of material. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior consent of the owner(s) / author(s). This edition is manufactured in India and is authorized for distribution only during ‘STeP-IN SUMMIT 2014’ as per the applicable conditions.
Performance Tuning JVM – Concepts, Case Studies, Practical Implementations for Oracle hotspot JVM

Kranthi Paidi, Performance Architect, Cigniti Technologies

Agenda

» JVM and Why you should learn about it?
» Heap and how it is divided in JVM
» Architecture of HotSpot JVM
» Garbage Collection
» Different generations and their sizing
» Live examples on how different flags can change the way GC works
» Different Collectors - Pros and Cons
» An overview on JVisualVM
» An overview on JConsole
Abstract & Speaker Profile

Speaker Profile

Kranthi Paidi has over 9 years of experience across enterprise application development in J2EE, Quality Assurance and Performance Testing fields. He has a wide experience in providing performance recommendations across .net, Java and mobile applications.

At Cigniti, Kranthi works as a Performance Architect providing solutions to various enterprise clients on performance improvements and capacity planning recommendations. He also leads a lot of activities in setting up a center of excellence for Performance Testing operations at Cigniti.

Kranthi Paidi believes in Steve Jobs' words - "The only way to do great work is to love what you do. If you haven’t found it yet, keep looking. Don’t settle".

During free time, he likes to travel to the Himalayan region.

Performance Architect
Cigniti Technologies

https://twitter.com/KKPaidi

JVM and Different Vendors

» A virtual machine that enables portability of code across platforms
» Manages memory allocation
» Tracks object lifecycle
» There are open source JVMs as well as proprietary ones. HotSpot from Oracle, IBM JVM and JRockit are some of the more popular versions
Why should you learn about JVM?

Top 10 Java performance problems

» OutOfMemoryError
» High CPU utilization
» Excessive Garbage collection
» Poor Integrations with other systems
» Poor DB tuning
» Application specific problems
» Thread contentions
» I/O, JDBC Problems
» Lack of / Excessive caching
» Poorly tuned middleware

Why should you learn about JVM?

Top 10 Java performance problems

» OutOfMemoryError
» High CPU utilization
» Excessive Garbage collection
» Poor Integrations with other systems
» Poor DB tuning
» Application specific problems
» Thread contentions
» I/O, JDBC Problems
» Lack of / Excessive caching
» Poorly tuned middleware

Heap, Native Heap, OS Reserved

Your RAM

Kernel Mode

Java Heap

Native Heap

java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: PermGen space

java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?
Understanding the java heap

» Java runs as a process on the OS. As a result, OS imposes certain constraints. A 32 bit OS can only give at most 4 GB less the space used by OS.
» An Out Of Memory Error can be thrown either by java heap or native heap
» A java heap is the memory allocated by –xmx option
» A native heap is the area used by jvm and the space left out after allocation to native OS (kernel mode) and java heap
» When you create an object on the java heap, depending on what the object does, you will have some of the native heap being used as a result of the action.

Understanding Java heap

» A java object has 3 slots of data. In case of arrays, it is 4
  » Class → Pointer to class information
  » Flags → shape, hash code, etc
  » Lock → flatlock or pointer to inflated monitor
  » Size → the length of an array (in case of arrays only)
» The memory taken by each slot is dependent on the processor architecture. For a 32 bit, each slot takes 32 bits of space. The same slot takes 64 bits on 64 bit machine.
  » As a result, if you take the same application running on 32 bit JDK and move it to 64 bit, it is going to use more memory.
  » Remember that this memory expansion is applicable only to Objects and not native data types
» Migrating a java app from 32 bit to 64 bit increases the java heap usage by ~70% and native heap usage by ~90%
HotSpot JVM Architecture

Understanding the Generations

» Memory is managed in *generations*, or memory pools holding objects of different ages.
» The *permanent generation* is a non-heap memory area that stores per-class structures as well as interned strings.
» Garbage collection occurs in each generation when the generation fills up.
» *Heap memory* is the runtime data area from which memory for all class instances and arrays is allocated.
» Only one survivor space can be used at a time.
How Generations are sized?

» Reserved Vs Committed

» -Xms and -Xmx values will influence how the generations are sized
  » -Xms < -Xmx ➔ -Xms is “Committed”
  » The uncommitted is “virtual”

» Each generation will grow to the limit of their respective virtual space

» When sizing the generations, remember one rule – “Most Objects die young”

» There is no one right way to size generations. The best choice is determined by the way the application uses memory as well as user requirements.

An object’s journey through generations
Most objects die young

Garbage Collection

- Automatic memory recycling
- HotSpot JVM’s key feature is generational GC
- The entire heap need not be collected every time, but, a proportion of it
- HotSpot provides multiple garbage collectors
- Starting from J2SE5.0, garbage collector is selected based on the class of the machine on which the application is run.
- JVMs from sun prior to J2SE1.4 do not support parallel gc.
- A serial collector is usually adequate for small applications whose heap requirement is up to 100MB
- For an application that runs on a machine with large memory, 2 or more processors, and is heavily threaded, you need to choose collectors other than serial collector. Starting J2SE5.0, when applications are run on these server class machine, parallel collector is chosen by default.
Garbage Collection

» Use -verbose:gc to print details of garbage collection and heap at each collection
» Option -XX:PrintGCDetails will give you additional information about the collection
» Option -XX:PrintGCTimeStamps adds a timestamp at the start of each collection. Shows how frequently garbage collections occur

Understanding GC Pauses

» When a STW Collection happens, all non-GC related threads are suspended which directly affect the latency
» The total time in a young generation’s minor GC is the time it takes to scan the roots in stacks (T1), scan roots in old space (T2) and copy the live objects (T3)
» Major factors affecting the time of STW are T2 and T3
» More number of GCs, more pauses
» Frequency of GCs (T4) is determined by application allocation rate and size of the Eden space. More the allocation rate and less the size of the eden, more pause times.
» Larger the Old gen, larger the T2.
» Smaller the Eden, larger the T4
Ergonomics

» Process of automatically tuning flags is called ergonomics

» Ergonomics was introduced in J2SE5.0 with a goal of providing good performance with little or no tuning of command line options by selecting the gc, heap size and runtime compiler at JVM startup instead of using fixed defaults.

» Any 32-bit JVM running on a Microsoft Windows irrespective of the number of CPUs is a client class

» Any 32 bit JVM running on a machine with one CPU regardless of the operating system is a client class

» Remaining every thing is a Server class

Ergonomics

» On server class machines, by default the following are selected:
  » Throughput garbage collector
  » Heap sizes
    » Initial heap size of 1/64 of physical memory up to 1 GB
    » Maximum heap size of ¼ of physical memory up to 1 GB
  » Server runtime compiler

» Prior to J2SE5, tuning involved adjusting heap size, size of the generations in heap, size of the survivor spaces in young generation and good judgment

» In version 5.0, two parameters were introduced as a part of ergonomics to meet the garbage collection tuning
  » Maximum Pause time goal
  » Application throughput goal
Minor Collections vs Major Collections

» Once Eden becomes full, a minor GC fires up
» A minor collection cleans only young generation
» When the tenured space fills up, a major collection is triggered
» All major collections are Stop The World Collections a.k.a STW
» Different collectors are available to meet individual application performance targets
» A major collection can happen without STW too depending on the choice of the collector for GC
» Minor collections and Major collections – both affect throughput and response times in several ways

See how generations are filling up

» Visual GC from Visual VM
» Jconsole Memory Tab
» Command line option: -verbose:gc
Growing and shrinking of heap

» The VM grows or shrinks the heap at the end of each collection to keep the proportion of free space to live objects in a specific range
» You can set these values using –XX:MinHeapFreeRatio and –XX:MaxHeapFreeRatio
» For Min and Max values of 30 and 80, the generation will expand to maintain 30% free space and shrinks to keep it within 80%
» Total size is still bound by –Xms and –Xmx
» To remove the entire overhead of growing and shrinking, you can set –Xms=-Xmx. But, be careful. VM will not come to rescue if your choice is poor.

Growing and shrinking of heap

» By default, a generation grows in increments of 20% and shrinks in increments of 5%
» You can control the growth of generations by using the flags -XX:YoungGenerationSizeIncrement for young generation and -XX:TenuredGenerationSizeIncrement for tenured generation
» You can control the shrinking by using -XX:AdaptiveSizeDecrementScaleFactor
The Entire Heap

- The most popular flags -Xms and -Xmx will define the lower and upper bounds for heap
- -Xms and -Xmx are actually shortcuts. They are internally mapped to -XX:InitialHeapSize and -XX:MaxHeapSize
- Not setting an -Xmx will most of the time result in OutOfMemory Error

Permanent Generation

- A heap area which contains object representations of all classes loaded by the JVM
- If you have a lot of class loading to be done during the lifetime of your application, better have a good amount of memory allocated to this area by using -XX:MaxPermSize
- The Perm Gen is “NOT” a part of the actual heap
Young Generation

- This is where most of the action takes place and ends.
- You can control the size of the young generation by modifying `-XX:NewRatio`
- Setting `-XX:NewRatio=4` will give 20% of the heap size to Eden and survivors and 80% of the heap space to tenured generation
- You can also set the absolute size by using `-XX:NewSize` and `-XX:MaxNewSize`
- If you do not set these values explicitly, minor GCs happen frequently. OTOH, setting these gives you that familiar saw tooth wave and minor collections happen only when the generation is filled up.

Young generation sizing

- Too small young generation causes short lived objects to move into old generation where they are harder to collect
- Too large young generation can cause more longer pauses during minor GCs as you need a lot of copying activity.
- Larger younger generation will lead to smaller older generation and induces instability
- The max value for `-XX:MaxNewSize` has an upper bound of `-Xmx/2`
- Absolute values have precedence over relative measurements
Survivor Space Sizing

- Two survivor spaces – S0 and S1 or “To” and “From”
- Job is to keep young objects in young generation
- Very helpful in observing the lifetime distribution of an application
- Survivor ratio can be tuned by using \(-XX:\text{SurvivorRatio}\)
- Not a big performance gain in using this
- Too small survivor space will cause the collection overflow to tenured generation
- Too large space will be uselessly empty
- \(-XX:\text{MaxTenuringThreshold}\) and \(-XX:\text{NeverTenure}\) will affect how survivor space is utilized

Survivor Space Sizing

- At the beginning of GC, the survivor space “To” is empty
- During GC, all objects in GC that are still being referenced are moved into TO
- Objects in the From region will move into TO if their tenuring threshold is not met. They move into Old otherwise.
- This From and To regions, also called S0 and S1 swap their roles for next GC cycle
- You see, if your survivor spaces are not properly sized, you risk growing the Old gen quicker than if they are properly sized. The final result, more full GCs
- \(-XX:+\text{PrintTenuringDistribution}\) will reveal what sizes are being anticipated in TO region
Old generation

» Objects that survived tenuring threshold move into Old generation
» Objects can also be directly born in old generation if their size is big and cannot fit in the young generation
» You can use -XX:PretenureSizeThreshold to set a threshold size. When you set this, any object of size greater than this threshold is directly created in Old generation.
» When you have to reach a point in minimizing the minor GC pauses vs sizing the young generation, a well thought out threshold size can greatly improve the performance.

Serial Collector

» Enabled by using -XX:+UseSerialGC
» “Mostly Enough” for “Small” applications which doesn’t require more than 100 MB of heap space
» Uses single processor
» Uses single thread
» Cannot take advantage of multiple processors
» Not a good choice for applications that are multi threaded and use large memory
» Good point is, it keeps the amount of work to bare minimum
» It’s a STOP THE WORLD collector. Nothing can work when it’s working
Parallel Collector

- Also known as throughput collector
- Performs minor collections in parallel
- Enabled by using `–XX:+UseParallelGC`
- Reduces Stop the world time
- Number of threads can be configured by using `–XX:ParallelGCThreads`
- Default number of threads equal number of CPUs
- For applications with medium to large sized data sets
- Default on most systems
- By default, only minor collections are done in parallel

Parallel Collector

- Parallel compaction is a feature available from J2SE 5.0
- Parallel compaction allows major collections in parallel
- Without parallel compaction, major collections are performed using a single thread and thus limits scalability
- Enabled by using `-XX:+UseParallelOldGC`
- Parallel collector uses a method of automatic tuning
- Values can be set for maximum pause time goal by using `-XX:MaxGCPauseMillis`
- By default, there is no max pause time goal
- Max pause time goal can reduce the throughput
Parallel Collector

- Max pause time goal can sometimes be not met
- Throughput goal can be set using `-XX:GCTimeRatio`
- `GCTimeRatio` is the ratio of GC time to application time
- Default value is 99 which means a goal of 1% time in GC
- Order of these goals:
  - Maximum pause time
  - Throughput
- Parallel collector throws an `OutOfMemoryError` if too much time is being spent in GC

Concurrent Collector

- Enabled by using `-XX:+UseConcMarkSweepGC`
- Performs “most” of its work concurrently
- Keeps GC pauses short
- Preferred for applications having large tenured generation
- Reduces gc time by using threads concurrently with the application for reaching out to objects
- Similar to parallel collector, throws an `OutOfMemoryError` if more than 98% of time is spent in GC
- Pauses an application twice during a concurrent collection cycle. A brief pause at the start of each collection and one during the middle of the collection
Concurrent Collector

» First pause marks all the reachable live objects. Also referred to as initial mark pause
» Once this is done, concurrent tracing continues
» Second pause comes at the end of concurrent tracing
» Concurrent collector starts at a time such that collection can finish before the tenured generation fills up. Failing which, you encounter “concurrent mode failure”
» You can control this by using -XX:CMSInitiatingOccupancyFraction
» Default value is 92%

Concurrent Collector - Incremental Mode

» Lessens the impact of long concurrent phases by periodically stopping the concurrent phase to yield back the processor to the application
» Referred to as i-cms
» Useful when applications that need low pause times run on small number of processors
» Enabled by using -XX:+CMSIncrementalMode. Disabled by default
» Uses a duty cycle to control the amount of work the collector can do before relinquishing the CPU
» These duty cycles are automatically adjusted by the JVM by looking into the history
Concurrent Collector - Incremental Mode

» The knob is -XX:+CMSIncrementalPacing

» This automatic pacing uses stats gathered while the program is running so that concurrent mode failures do not fire up

» If too many full collections occur,
  » Increase the safety factor using -XX:CMSIncrementalSafetyFactor
  » Increase the minimum duty cycle by using the flag -XX:CMSIncrementalDutyCycleMin
  » Disable automatic pacing by using -XX:-CMSIncrementalPacing

G1 Collector

» Server style garbage collector

» Also called “Garbage First” collector

» Targeted for multi processor machines with large memories

» Meets pause time goals with high probability while achieving high throughput

» Supported from JDK 7 update 4 and later

» Long term replacement for CMS collector

» G1 collector totally changes the hitherto way of partitioning the heap by serial, parallel and CMS garbage collectors
**G1 Collector**

- Heap is partitioned into a set of equal sized heap regions
- No fixed size for eden, old and survivor
- G1 has both concurrent and parallel phases
- Full GCs are still single threaded
- Heap is split into approx 2K regions with a minimum size of 1 Mb and a maximum size of 32 Mb
- Initial mark (STW) \(\rightarrow\) Root Region scanning \(\rightarrow\) Concurrent Marking \(\rightarrow\) Remark (STW) \(\rightarrow\) Cleanup(STW) \(\rightarrow\) Copying (STW)
- Enable by using `-XX:+UseG1GC`
G1 Collector

» Set the target for max GC pause time by using -XX:MaxGCPauseMillis. Default is 200
» Control when a concurrent GC starts by setting a percentage value to -XX:InitiatingHeapOccupancyPercent
» Do not set young generation size while using G1
» Increase the number of marking threads using the JVM flag -XX:ConcGCThreads
» To avoid evacuation failure, increase -XX:G1ReservePercent. Default is 10

JVM tuning Flags

» JVM accepts two kinds of flags: boolean flags and flags that require a parameter
» Boolean flags: -XX:+FlagName enables where as –XX:-FlagName disables
» Flags that require a parameter: -XX:FlagName=something
Some Myths

» Performance Tuning JVM ≠ Performance tuning Java Code
» If I know all tuning flags and their defaults, I can tune the JVM
» There is no one right way to size generations.

Memory leaks

» Unintentional Object retention
» Can be caused by
  » Poor coding
  » Excessive use of finalizers
Selecting a Collector

» Let VM choose the best one for you
» If you see problems, select a collector based on application’s memory usage
» If response times are more important, use concurrent collector
» If the collector does not achieve the desired performance, attempt to adjust the heap and generation sizes
» If you use JDK 7 update 5 and later, and run applications on multi processor machines, use G1 collector

References

» http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html#PerformanceTuning
» http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html
» http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html#PerformanceTuning
» http://blog.griddynamics.com/