# Gargbage Collection (GC) Tuning Guidelines & Examples

# Topics

- General GC Tuning Guidelines
- GC Tuning Examples

- This presentation is based on the following document: http://java.sun.com/performance/reference/whitepapers/tuning.html

# General Tuning Guidelines

# GC Tuning Guidelines

- Be aware of Ergonomic settings
- Heap sizing
- GC policy
- Other tuning parameters

# Be Aware of Ergonomic Settings

- Before you start to tune the command line arguments for Java be aware that Sun's HotSpot™ Java Virtual Machine has incorporated technology to begin to tune itself.
- This smart tuning is referred to as Ergonomics.
- Most computers that have at least 2 CPU's and at least 2 GB of physical memory are considered server-class machines which means that by default the settings are:
  - The -server compiler
  - -XX:+UseParallelGC parallel (throughput) garbage collector
  - -Xms initial heap size is 1/64th of the machine's physical memory
  - -Xmx maximum heap size is 1/4th of the machine's physical memory (up to 1 GB max).

# Heap Sizing

- Even though Ergonomics significantly improves the "out of the box" experience for many applications, optimal tuning often requires more attention to the sizing of the Java memory regions.
- The maximum heap size of a Java application is limited by three factors:
  - The process data model (32-bit or 64-bit) and the associated operating system limitations
  - The amount of virtual memory available on the system
  - The amount of physical memory available on the system.
- The size of the Java heap for a particular application can never exceed or even reach the maximum virtual address space of the process data model.

# Heap Sizing for 32bit/64bit Systems

- For a 32-bit process model, the maximum virtual address size of the process is typically 4 GB, though some operating systems limit this to 2 GB or 3 GB.
  - The maximum heap size is typically -Xmx3800m (1600m) for 2 GB limits), though the actual limitation is application dependent.
- For 64-bit process models, the maximum is essentially unlimited.

# Heap Sizing & Physical RAM

- For a single Java application on a dedicated system, the size of the Java heap should never be set to the amount of physical RAM on the system, as additional RAM is needed for the operating system, other system processes, and even for other JVM operations.
- On systems with multiple Java processes, or multiple processes in general, the sum of the Java heaps for those processes should also not exceed the the size of the physical RAM in the system.

# Heap Sizing & Young Generation

- The next most important Java memory tunable is the size of if the young generation (also known as the NewSize).
- Generally speaking the largest recommended value for the young generation is 3/8 of the maximum heap size.
  - Note that with the throughput and low pause time collectors it may be possible to exceed this ratio.

# Garbage Collector Schemes

- The -XX:+UseParallelGC parallel (throughput) garbage collector
- The -XX:+UseConcMarkSweepGC concurrent (low pause time) garbage collector (also known as CMS)
- The -XX:+UseSerialGC serial garbage collector (for smaller applications and systems)

# Other Tuning Parameters

- By appropriately configuring the operating system and then using the command line options -XX :+UseLargePages (on by default for Solaris) and -XX :LargePageSizeInBytes you can get the best efficiency out of the memory management system of your server.
- Note that with larger page sizes we can make better use of virtual memory hardware resources (TLBs), but that may cause larger space sizes for the Permanent Generation and the Code Cache, which in turn can force you to reduce the size of your Java heap.
  - This is a small concern with 2 MB or 4 MB page sizes but a more interesting concern with 256 MB page sizes.

# GC Tuning Examples

# Caveat

- Here are some specific tuning examples for your experimentation.
- Please understand that these are only examples and that the optimal heap sizes and tuning parameters for your application on your hardware may differ.
- This is based on JDK 5.

# Example 1: Tuning for Throughput

# Ex1: Tuning for Throughput

- Tuning for a server application running on system with 4 GB of memory and capable of running 32 threads simultaneously (CPU's and cores or contexts).
- java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX :+UseParallelGC -XX:ParallelGCThreads=20
- -Xmx3800m -Xms3800m
  - Configures a large Java heap to take advantage of the large memory system.
- -Xmn2g
  - Configures a large heap for the young generation (which can be collected in parallel), again taking advantage of the large memory system.
  - It helps prevent short lived objects from being prematurely promoted to the old generation, where garbage collection is more expensive.

# Ex1: Tuning for Throughput (Cont.)

- -Xss128k
  - Reduces the default maximum thread stack size, which allows more of the process' virtual memory address space to be used by the Java heap.
- -XX:+UseParallelGC
  - Selects the parallel garbage collector for the young generation of the Java heap (note: this is generally the default on server-class machines)
- -XX:ParallelGCThreads=20
  - Reduces the number of garbage collection threads. The default would be equal to the processor count, which would probably be unnecessarily high on a 32 thread capable system.

# Example 2: Try the Parallel Old Generation Collector

# Ex2: Try Parallel Old Gen. Collector

- Similar to example 1 we here want to test the impact of the parallel old generation collector.
- java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX :+UseParallelGC -XX:ParallelGCThreads=20 -XX :+UseParallelOldGC
- -Xmx3550m -Xms3550m
  - Sizes have been reduced. The ParallelOldGC collector has additional native, non-Java heap memory requirements and so the Java heap sizes may need to be reduced when running a 32-bit JVM.
- -XX:+UseParallelOldGC
  - Use the parallel old generation collector. Certain phases of an old generation collection can be performed in parallel, speeding up a old generation collection.

# Example 3:
# Try 256MB pages

# Ex3: Try 258M Pages

- This tuning example is specific to those Solaris-based systems that would support the huge page size of 256 MB.
- java -Xmx2506m -Xms2506m -Xmn1536m -Xss128k -XX :+UseParallelGC -XX:ParallelGCThreads=20 -XX :+UseParallelOldGC -XX:LargePageSizeInBytes=256m
- -Xmx2506m -Xms2506m
  - Sizes have been reduced because using the large page setting causes the permanent generation and code caches sizes to be 256 MB and this reduces memory available for the Java heap.

# Ex3: Try 258M Pages (Cont.)

- -Xmn1536m
  - The young generation heap is often sized as a fraction of the overall Java heap size.
  - Typically we suggest you start tuning with a young generation size of 1/4th the overall heap size.
  - The young generation was reduced in this case to maintain a similar ratio between young generation and old generation sizing used in the previous example option used.
- -XX:LargePageSizeInBytes=256m
  - Causes the Java heap, including the permanent generation, and the compiled code cache to use as a minimum size one 256 MB page (for those platforms which support it).

# Example 4: Try -XX :+AggressiveOpts

# Ex4: Try -XX:+AggressiveOpts

- This tuning example is similar to Example 2, but adds the AggressiveOpts option.
- java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX :+UseParallelGC -XX:ParallelGCThreads=20 -XX :+UseParallelOldGC -XX:+AggressiveOpts
- -Xmx3550m -Xms3550m
  - Sizes have been increased back to the level of Example 2 since we no longer using huge pages.
- -Xmn2g
  - Sizes have been increased back to the level of Example 2 since we no longer using huge pages.

# Ex4: Try -XX:+AggressiveOpts (Cont.)

- -XX:+AggressiveOpts
  - Turns on point performance optimizations that are expected to be on by default in upcoming releases.
  - The changes grouped by this flag are minor changes to JVM runtime compiled code and not distinct performance features (such as BiasedLocking and ParallelOldGC).
  - This is a good flag to try the JVM engineering team's latest performance tweaks for upcoming releases. Note: this option is experimental! The specific optimizations enabled by this option can change from release to release and even build to build.
  - You should reevaluate the effects of this option with prior to deploying a new release of Java.

# Example 5:
# Try Biased Locking

# Ex5: Try Biased Locking

- This tuning example is builds on Example 4, and adds the Biased Locking option.
- java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX :+UseParallelGC -XX:ParallelGCThreads=20 -XX :+UseParallelOldGC -XX:+AggressiveOpts -XX :+UseBiasedLocking

# Ex5: Try Biased Locking (Cont.)

- -XX:+UseBiasedLocking
  - Enables a technique for improving the performance of uncontended synchronization.
  - An object is "biased" toward the thread which first acquires its monitor via a monitorenter bytecode or synchronized method invocation; subsequent monitor-related operations performed by that thread are relatively much faster on multiprocessor machines.
  - Some applications with significant amounts of uncontended synchronization may attain significant speedups with this flag enabled; some applications with certain patterns of locking may see slowdowns, though attempts have been made to minimize the negative impact.

# Example 6: Tuning for low pause times and high throughput

# Ex6: Tuning for low pause times and high throughput

- This tuning example similar to Example 2, but uses the concurrent garbage collector (instead of the parallel throughput collector).
- java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX :ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX :+UseParNewGC -XX:SurvivorRatio=8 -XX :TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=31
- -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
  - Selects the Concurrent Mark Sweep collector. This collector may deliver better response time properties for the application (i.e., low application pause time).
  - It is a parallel and mostly-concurrent collector and and can be a good match for the threading ability of an large multi-processor systems.

# Ex6: Tuning for low pause times and high throughput (Cont.)

- -XX:SurvivorRatio=8
  - Sets survivor space ratio to 1:8, resulting in larger survivor spaces (the smaller the ratio, the larger the space).
  - Larger survivor spaces allow short lived objects a longer time period to die in the young generation.
- -XX:TargetSurvivorRatio=90
  - Allows 90% of the survivor spaces to be occupied instead of the default 50%, allowing better utilization of the survivor space memory.

# Ex6: Tuning for low pause times and high throughput (Cont.)

- -XX:MaxTenuringThreshold=31
  - Allows short lived objects a longer time period to die in the young generation (and hence, avoid promotion). A consequence of this setting is that minor GC times can increase due to additional objects to copy.
  - This value and survivor space sizes may need to be adjusted so as to balance overheads of copying between survivor spaces versus tenuring objects that are going to live for a long time. The default settings for CMS are SurvivorRatio=1024 and MaxTenuringThreshold=0 which cause all survivors of a scavenge to be promoted.
  - This can place a lot of pressure on the single concurrent thread collecting the tenured generation. Note: when used with -XX :+UseBiasedLocking, this setting should be 15.

# Example 7: Try AggressiveOpts for low pause times and high throughput

# Ex7: Try AggressiveOpts for low pause times and high throughput

- This tuning example is builds on Example 6, and adds the AggressiveOpts option.
- java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX :ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX :+UseParNewGC -XX:SurvivorRatio=8 -XX :TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=31 -XX :+AggressiveOpts

# Ex7: Try AggressiveOpts for low pause times and high throughput (Cont.)

- -XX:+AggressiveOpts
  - Turns on point performance optimizations that are expected to be on by default in upcoming releases.
  - The changes grouped by this flag are minor changes to JVM runtime compiled code and not distinct performance features (such as BiasedLocking and ParallelOldGC). This is a good flag to try the JVM engineering team's latest performance tweaks for upcoming releases. Note: this option is experimental!
  - The specific optimizations enabled by this option can change from release to release and even build to build. You should reevaluate the effects of this option with prior to deploying a new release of Java.

# Gargbage Collection (GC) Tuning Guidelines & Examples