Summer 08

Java Performance Tuning

Michael Finocchiaro

This white paper presents the basics of Java Performance Tuning for large Application Servers.

Table of Contents

Java Performance Tuning

Garbage Collection

The Java programming language is object-oriented and includes automatic garbage collection. Garbage collection is the process of reclaiming memory taken up by unreferenced objects.

A comprehensive discussion on Garbage Collection can be found at:

http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

The following sections will try to resume some of the concepts from this document and their impact on Application Server performance.

Memory Management, Java and Impact on Application Servers

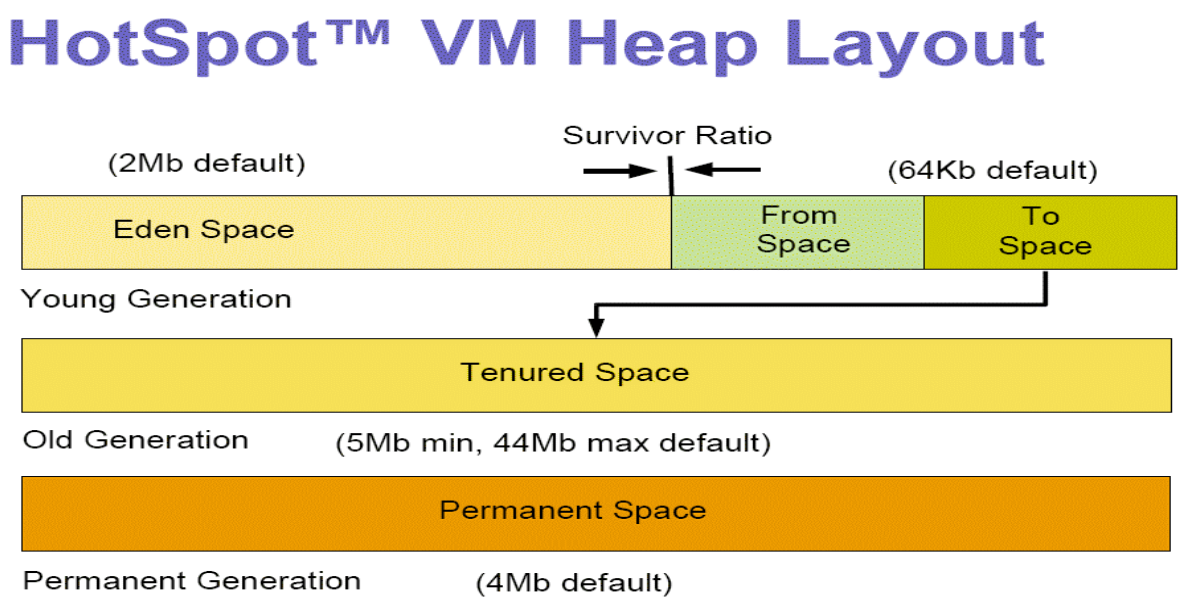
The task of memory management that was always challenging with compiled object-oriented languages such as C++. On the other hand, Java is an interpretive language that takes this memory management out of the hands of developers and gives it directly to the virtual machine where the code will be run. This means that for best performance and stability, it is critical that the Java parameters for the virtual machine be understood and managed by the Application Server deployment team. This section will describe the various parts of the Java heap and then list some useful parameters and tuning tips for ensuring correct runtime stability and performance of Application Servers.

Java Virtual Machines

The Java specification as to what is “standard” for a given release is written and maintained by the JavaSoft division of Sun Microsystems. This specification is then delivered to other JVM providers (IBM, HP, etc). JavaSoft provides the standard implementation on Windows, Solaris, and LINUX. Other platforms are required to deliver the code functionality but their JVM options can be different. Java options that are preceded with “-X” or “-XX” are typically platform-specific. That being said, many options are used on all platforms. One must read in detail the README notes from the various releases on various platforms to be kept up-to-date with the variations. This guide will mention the most critical ones and distinguish between those which are implemented on most platforms and those which are platform-specific.

The Java Heap

The Java heap is divided into three main sections: Young Generation, Old Generation and the Permanent Generation as shown in the figure below.



Also, not shown here, is another area called the Code Cache which is typically about 50Mb in Size.

HotSpot VM

The HotSpot VM is the garbage collector that comes with the JavaSoft virtual machine. Its specification is delivered to all JVM providers (HP, IBM, BEA, etc) and the standard implementation is deployed by JavaSoft on Solaris, Windows and LINUX.

Java Heap Description

Young Generation

The Eden Space of the Young Generation holds all the newly created objects. When this generation fills, the Scavenge Garbage Collector clears out of memory all objects that are unreferenced. Objects that survive this scavenge moved to the “From” Survivor Space. The Survivor Space is a section of the Young Generation for these intermediate-life objects. It has two equally-sized subspaces “To” and “From” which are used by its algorithm for fast switching and cleanup. Once the Scavange GC is complete, the pointers on the two spaces are reversed: “To” becomes “From” and “From” becomes “To”.

The Young Generation is sized with the -Xmn option on the Java command line. It should never exceed half of the entire heap and is typically set to 1/2 of the heap for JVMs less than 1.5Gb and to 1/3 of the heap for JVMs larger than 1.5Gb. Note that the argument to the -Xm{n,s,x} options can be suffixed with “m” or “M” for megabytes and “g” or “G” for gigabytes. For example, a 256Mb Young Generation is specified by -Xmn256m. As a further example, a 1Gb Young Generation is specified with -Xmn1g or -Xmn1000m.

The Survivor Space in the Young Generation is sized as a ratio of one of the sub-spaces to the Eden Space – this is called the SurvivorRatio. For example, if -Xmn is set to 400m and -XX:SurvivorRatio is set

to 4, then the total Survivor Space will be 133.2Mb with "To" and "From" each being 66.6Mb and the Eden Space being 266.8Mb. The survivor ratio = 266.8 (Eden) / 66.6 (To) = 4.

Old Generation

Once an object survives a given number of Scavenge GCs, it is promoted (or tenured) from the "To" Space to the Old Generation. Objects in this space are never garbage collected except in the two cases: Full Garbage Collection or Concurrent Mark-and-Sweep Garbage Collection. If the Old Generation is full and there is no way for the heap to expand, an Out-of-Memory error (OOME) is thrown and the JVM will crash.

The Old Generation is sized with the -Xms and -Xmx parameters, where -Xms is the initial heap size allocated at startup and -Xmx is the maximum heap size reserved by the JVM at startup. If the heap size exceeds free memory on the system, swapping will occur and performance will be seriously degraded.

The maximum value for -Xmx is architecture-dependent.

Architecture	**-Xmx limit (for total JVM size add ~200Mb)**
32-bit Windows (XP, Server 2003)	~1300m (no /3Gb flag)[1] ~1500m (/3Gb flag in boot.ini)
AIX 5.3L on POWER	~3200m
HP-UX 11iV1 on PA-RISC	~2200m (with chatr +q3p enabled)[2]
HP-UX 11iV2 on IA-64	~3200m (with chatr +as mpas)[3]
Solaris on UltraSPARC or T1 or T2	~3200m

Notes:

1. The /3Gb flag can be added to the boot.ini flag on a Windows XP SP2 or Windows 2003 Server machine in order to expand the lower memory range to allow 32-bit executables to address slightly more memory than without this flag. Note that the two values given are approximate and depend on the device drivers loaded by the operating system. For more information from Microsoft on the /3Gb switch, please see Microsoft Knowledge Base articles 291988 and 833721. For more information on memory addressability on the Windows platform, please see the white paper on this site:
http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/dw3gbswitch3.pdf

Note also that with Java6, the /3Gb flag on Windows will NOT be supported by the JVM.

2. The PA-RISC architecture is being phased out and is not currently recommended for new deployments. On PA-RISC, one must, as root, execute the following command on the java executeable in the /opt/java1.5/bin/PA-RISC2.0/ directory with both Application Server and Tomcat shutdown:
chatr +q3p enable java
This will enable approximately 2.0 to 2.2 Gb of space for the -Xms/-Xmx parameters depending on loaded device drivers, etc.

3. The IA-64 architecture is recommended for new deployments. On IA-64, one must, as root, execute the following command on the java executable in /opt/java1.5/bin/IA64N/ directory with both Application Server and Tomcat shutdown:
chatr +as mpas java
This will enable approximately 3.2 to 3.4 Gb of space for the -Xms/-Xmx parameters depending on loaded device drivers, etc.

The JVM is also distributed in a 64-bit version. It is enabled on most platforms via the -d64 flag. This allows for larger heaps but in most cases due to the larger memory spaces, longer sweep times, and larger pointers, it can cause a 5 to 20% performance penalty. It also naturally requires a 64-bit operating system. It is not recommended to use a 64-bit JVM unless all other methods of reducing OOME frequency have been attempted and proven ineffective.

Permanent Generation

The Permanent Generation is where class files are kept. These are the result of compiled classes and jsp pages. If this space is full, it triggers a Full Garbage Collection. If the Full Garbage Collection cannot clean out old unreferenced classes and there is no room left to expand the Permanent Space, an Out-of-Memory error (OOME) is thrown and the JVM will crash.

The Permanent Generation is sized with the -XX:PermSize and -XX:MaxPermSize parameters. For example, to specify a startup Permanent Generation of 48Mb and a maximum Permanent Generation of 128Mb, use the parameters: -XX:PermSize=48m -XX:MaxPermSize=128. It is exceedingly rare that more than 128Mb of memory is required for the Permanent Generation.

Note also that the Permanent Generation is tacked onto the end of the Old Generation. There is also a small code cache of 50Mb for internal JVM memory management. This means that the total initial heap size = -Xms + -XX:PermSize + ~50Mb and that the maximum total heap size = -Xmx + -XX:+MaxPermSize + ~50Mb. For example, if –Xms/–Xmx are set to 512m and –XX:PermSize/MaxPermSize are set to 128m, the total VM will actually about 700 Mb in size.

Default Garbage Collection Algorithms

Scavenge Garbage Collection

Scavenge Garbage Collection (also known as a Minor Collection) occurs when the Eden Space is full. By default, it is single-threaded but does not interrupt the other threads working on objects. It can be parallelized starting at Java 1.4 but if too more ParallelGCThreads are specified than CPU cores where the JVM is running, this can cause bottlenecks. For this reason, it is suggested to be careful in when and where to use the parallel options. These will be discussed further on.

Full Garbage Collection

A Full Garbage Collection (FullGC) occurs under these conditions:

The Java application explicitly calls System.gc(). This can be avoided by implementing the -XX:+DisableExplicitGC parameter in the startup command for all Application Server JVMs (Tomcat, Application Server JVM, etc.)

The RMI protocol explicitly calls System.gc() on a regular basis under normal operation. This can be avoided by implementing the -XX:+DisableExplicitGC parameter in the startup command for all Application Server JVMs.

A memory space, either Old or Permanent, is full and to accommodate new objects or classes, it needs to be expanded towards its max size, if the relevant parameters have different values. In other words, if -Xms and -Xmx have different values and if the size of Old needs be increased from -Xms towards -Xmx to accommodate more objects, a FullGC is called. Similarly, if -XX:PermSize and -XX:MaxPermSize have different values and the Permanent Space needs to be increased towards -XX:MaxPermSize to accommodate new java classes, a FullGC is called. This can be avoided by always setting -Xms and -Xmx as well as -XX:PermSize and -XX:MaxPermSize to the same value.

The Tenured Space is full and the Old Generation is already at the capacity defined by -Xmx. This can be avoided by tuning the Young Generation so that more objects are filtered out before being promoted to the Old Generation, by increasing the -Xmx value and/or by implementing the Concurrent Mark-and-Sweep (CMS) collector. CMS will be discussed below.

The Permanent Space is full and the Permanent Generation is already at the capacity defined by -XX:MaxPermSize. This can be avoided ensuring that the Permanent Space is large enough and that the -Xnoclassgc option is enabled.

The HotSpot VM thinks there is not free space in the heap to collect all the objects in Eden in case of an emergency collection. This can be avoided by using the -XX:MaxLiveObjectEvacuationRatio=30 option on JVMs prior to 1.4.2_13, and by the -XX:+HandlePromotionFailure option on JVMs from 1.4.2_14 and higher as well as Java5 JVMs.

A Concurrent Mark and Sweep (CMS) GC operation does not have enough space in Old to complete. CMS will be discussed below.

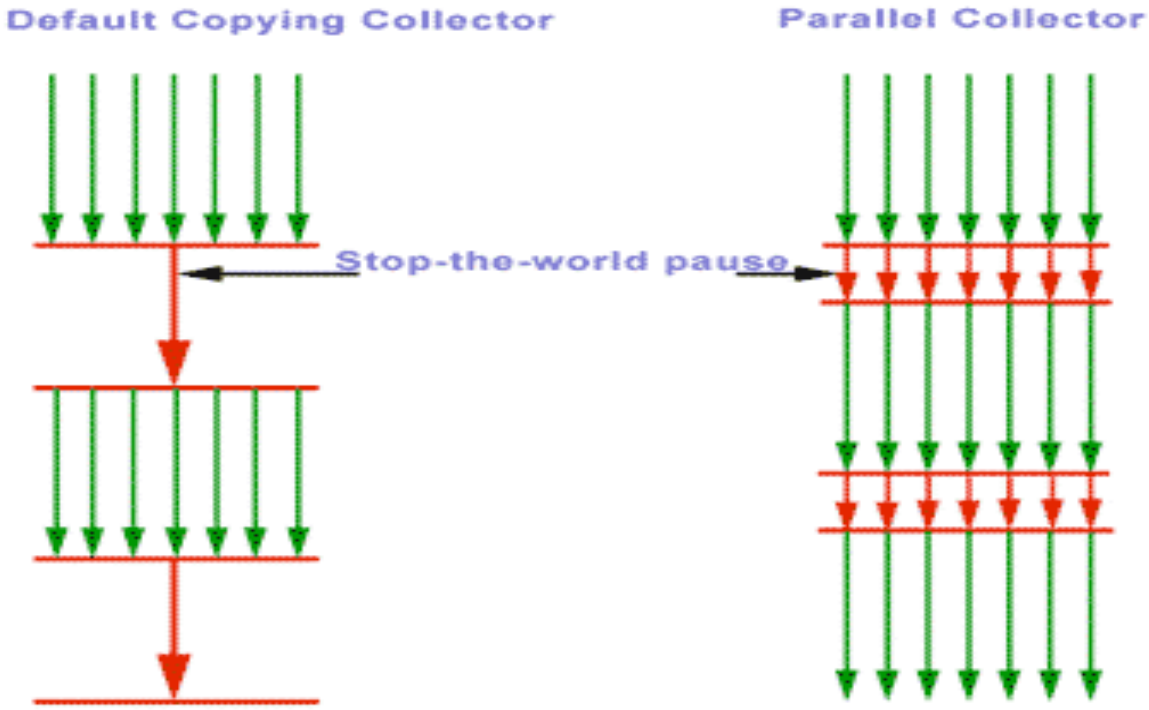
A Full Garbage collection is disruptive in the sense that all working threads are stopped and one JVM thread then will scan the entire heap twice trying to clean out unreferenced objects. At the same time, objects with finalizer clauses are processed. Once the second scan is complete, if some objects on the finalizer stack have not yet been processed, they are left on the queue for the next Full GC. This is an expensive process which causes delays in response time. The goal of tuning the JVM is to minimize the FullGCs while ensuring that an OOME does not occur.

Alternative Garbage Collectors

Parallel Copy GC

The -XX:+UseParNewGC turns on a multi-threaded collector for the Eden space. Idle threads are used for routine garbage collection tasks. The number of threads available is defined by -XX:ParallelGCThreads=#. This can drastically reduce the amount of time spent in Scavenge GC but needs to be used with precaution. The sum of ParallelGCThreads in the JVMs being run on the machine should

not exceed the available number of CPU cores on the machine. This can result is thread contention inside the JVM and reduced performance.

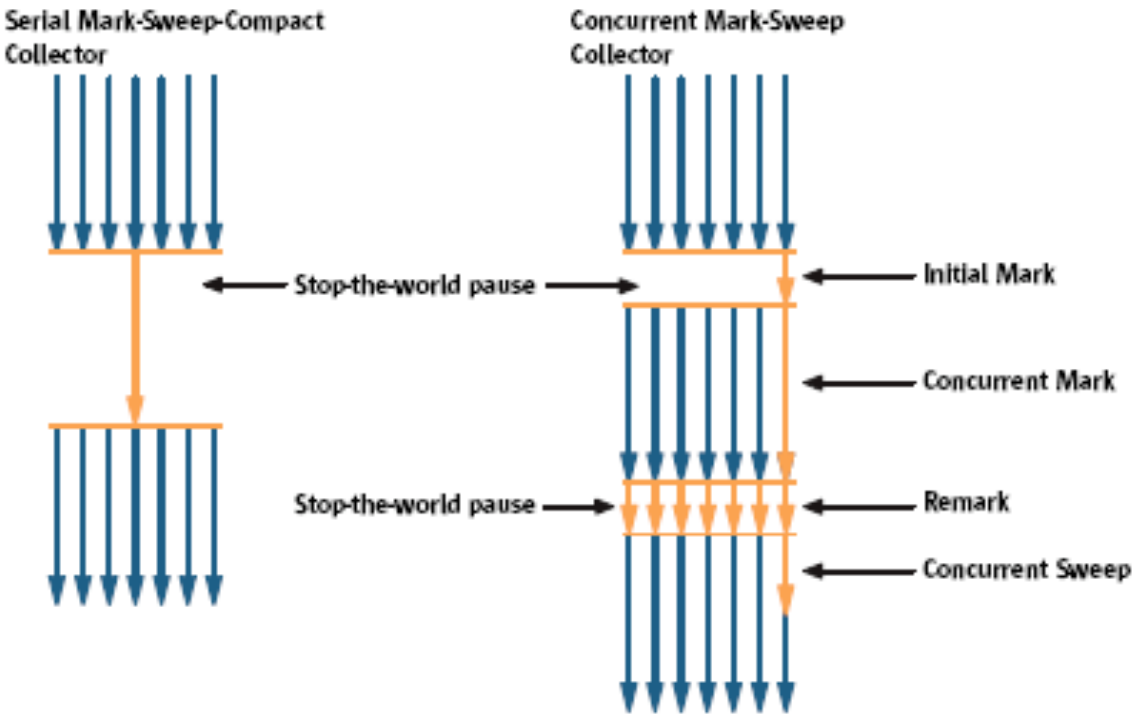


The figure above demonstrates the difference between the default Scavenge collector (also called here the default Copying Collector) and the Parallel Collector where the two pauses are now shorter due to the multiple threads deployed during the pauses.

Note that the default value of ParallelGCThreads is the number of CPU cores. Since Application Server uses at least three and usually four or more JVMs, particular attention should be taken to specify ParallelGCThreads explicitly and ensure that the sum across all the JVMs on a particular machine is equal to or less than the number of CPU cores.

Concurrent Mark-and-Sweep

The Concurrent Mark-and-Sweep Collector is a parallel collector for the Old Generation. It breaks the garbage collection into an initial marking pause, a concurrent marking phase, a second remarking clause, and a concurrent sweeping phase . This has the positive impact of avoiding FullGC and OOME. It has the negative side-effect of being very expensive as one or more CPU cores will be dedicated to CMS throughout the entire concurrent marking phase and one CPU will be dedicated to CMS throughout the entire concurrent sweeping phase.



The figure above demonstrates the difference between the single-threaded (here called Serial) Mark-Sweep Compact Collector and the Concurrent Mark-Sweep collector where the pauses shorter and many of the phases are done with multiple threads.

The CMS collector is enabled using the -XX:+UseConcMarkSweepGC flag. It is also suggested to use the -XX:+CMSParallelRemarkEnabled flag so that the Remarking phase happens with multiple threads as illustrated above.

CMS should be used only on relatively large systems (more than 4 CPU cores) where OOME is experienced regularly due to the CPU cost discussed earlier. Heaps need to at their maximum size with the -Xmn at 1/4 of the total heap for the algorithm to function efficiently.

Note that turning on CMS also implicitly turns on -XX:+UseParNewGC. For this reason, whenever CMS is activated, the ParallelGCThreads option must always be explicitly set to limit the number of threads as discussed above in the section on ParallelCopyGC.

Summary of Java1.4 and Java5 Command Line Options

Option	Description	Recommended
-Xmn<x><m|g>	Young Generation Size, suffixed with m (Mb) or g (Gb). It is recommended that	<1.5Gb heap: 1/2 of -Xmx

>1.5Gb heap: 1/3 of -Xmx CMS: 1/4 of -Xmx		
-Xms<x><m|g>	Initial size of Young Generation + Old Generation.	same as -Xmx unless server memory is <4Gb
-Xmx<x><m|g>	Maximum size of Young Generation + Old Generation	As large as possible given installed memory, number of JVMs, and server architecture
-XX:SurvivorRatio=<x>	Ratio of one Survivor Space to the Eden space.	4
-XX:PermSize=<x><m or g>	Initial Permanent Generation size	128m
-XX:MaxPermSize=<x><m or g>	Maximum Permanent Generation size	128m
-XX:+DisableExplicitGC	Ignore all calls to System.gc()	Deployed in all JVMs
-Xloggc:<filename>	Log gc activity to <filename>	Deployed in all JVMs
-XX:+PrintGC	Turn on GC logging	Deployed in all JVMs
-XX:+PrintGCDetails	Turn on detailed GC logging	Deployed in all JVMs
-XX:+PrintGCTimeStamps	Turn on timestamps in GC logging	Deployed in all JVMs
-XX:+JavaMonitorsInStackTrace	Monitor details when a stack trace is called for thread and monitor lock analysis	Deployed in all JVMs
-XX:+UseTLAB	Use a thread-local address block to reduce contention in eden for new allocations	Deployed in all JVMs
-XX:TLABSize	Initial TLAB Size	32k
-XX:+ResizeTLAB	Allow dynamic resizingof TLAB	Deployed in all JVMs
-XX:+UseParNewGC	Use ParallelCopy Collector	Use on systems with >4 CPU cores
-XX:ParallelGCThreads=<x>	Number of threads dedicated to	Start at 2 but ensure that sum

	ParallelCopy GC	of ParallelGCThreads across all JVMs is less than number of available CPU cores. Always explicitly specify when using CMS.
-XX:+UseConcMarkSweepGC	Enable CMS algorithm	Use on systems with >4 CPU cores ONLY if other methods of eliminating OOME have been exhausted
-XX:+CMSParallelRemarkEnabled	Enable Parallel Remarking in CMS	Always specify when using CMS
-XX:MaxLiveObjectEvacuationRatio	Percentage of Old used before the HotSpot collector throws an OldFull exception and causes a Full GC	Only use with JVMs up to 1.4.2_12.
-XX:+HandlePromotionFailure	Fix for issue mentioned above.	Use with all JVMs over 1.4.2_13 and with Java5.
-XX:SoftRefLRUPolicyMSPerMB=<x>	Frequency for removing soft references in milliseconds per megabyte	1 Use if OOME is experienced.

Sizing the JVMs

The default values of the basic Java parameters:

-Xms=3670k
-Xmx=64m
-XX:SurvivorRatio=32 (on Solaris, 8 on HP-UX, 32 on Windows)
-XX:NewSize=2228k
-XX:MaxNewSize=unlimited
-XX:NewRatio=2 (on Solaris, default is 8 on Windows)
-XX:PermSize=16m
-XX:MaxPermSize=64m

The Application Server and servlet engine experience two problems with these default VM default heap parameters:

• One is slow startup, because the default initial heap is small and must be resized over many major collections. The default initial heap size for an Application Server has been increased to 128MB.

• A more pressing problem is that the default maximum heap size is typically unreasonably small.

It is highly recommended that the maximum heap size be further increased based on the available RAM of your application server by adjusting the appropriate parameters in your particular Application Server.

Monitoring Garbage Collection Duration & Frequency

Each vendor provides its own tools for monitoring garbage collection. This section describes some tools available for Windows and Solaris platforms. For additional information on garbage collection, see the following sites:

Sun:

http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

HP-UX:

http://www.hp.com/products1/unix/java/infolibrary/prog_guide/hotspot.html

AIX

http://www-128.ibm.com/developerworks/java/jdk/diagnosis/

HPjtune

HP-UX provides a garbage collection log analysis tool called HPjtune that plots garbage collection activity graphically. It is primarily used for the HP-UX JVM but, when only –verbose:gc or –Xloggc are present in the Solaris or Windows JVMs, it can be used to read those as well.

jps

From the Java5 command line, run the jps command to obtain a list of running JVM processes that may be monitored. The process identifier (pid) and main class are displayed for each such process.

jstat

Jstat is part of the standard Java5 distribution and is capable or printing out statistics in real time on a running JVM. For example, specifying the -gc option:

```
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT
  136512.0 136512.0  0.0   28820.6 546176.0 508081.8 1277952.0   121102.8  98304.0 45056.9    235
60.181   0      0.000   60.181
```

where: S0 and S1 are the two survivor spaces, E is the Eden Space, O is the Old Generation, P is the Permanent Generation, YGC is a ScavengeGC, FGC is a FullGC, the suffix C means “Capacity”, the suffix U means “Used” and the suffix “T” means Time.

visualgc

The visualgc tool is available at java.sun.com (http://java.sun.com/performance/jvmstat/)

The same information that you can get from jstat is available graphically using the visualgc tool. For example, enter the following for a visual display:

D:\jvmstat\bat>visualgc 9040

The following picture shows a sample application display taken at a different point in time from the previous example: The left window shows the current sizes of the Perm, Old, Eden and survivor spaces (S0 and S1) in the heap. The right window also shows timings for significant JVM events such as garbage collections. The right window shows sizes of these same spaces in the heap as strip charts. Through the strip charts, some history is available for spotting trends. Unfortunately, the visualgc tool does not support logging of garbage collection activity to file.

Java Command Line

The java command accepts the directive -Xloggc that specifies a file name where the garbage collection diagnostic data is written. Details about the garbage collection activity, such as size of the young and old generation before and after garbage collection, size of total heap, elapsed time it takes for a garbage collection to happen in young and old generation, and size of objects promoted at every garbage collection, and other data, can be recorded by using the both the -XX:+PrintGCDetails and -XX:+PrintGCTimeStamps argument. Even more detail (but not recommended for production environments) can be obtained with the -XX:+PrintTenuringDistribution.

Managing System Memory

An integral part of optimizing Application Server performance is proper memory management. The frequency and duration of Java garbage collection (heap management) is greatly influenced by transaction load and available memory.

Both the Application Server and the database server can efficiently cache persistent data in memory to minimize disk reads and thus increase transaction throughput. In addition, the Web server, servlet engine and operating system require significant amounts of memory. The goal is to balance all memory requirements with physical memory and avoid excessive virtual memory paging.

Balancing System Memory

Use this section as a starting point when determining memory allocation on your initial installation of the Application Server. The information provided assumes that the servers involved are dedicated to

running your Application Server solution. The information should not be taken as fixed rules that must be followed; there can be many site specific reasons to use a different allocation scheme.

Note: After you have your production system running, use the performance analysis techniques described in this guide to arrive at the memory allocation scheme that best meets your production system. Use the information in the following Memory Allocation Guidelines table as a set of guidelines for memory allocation. The first column lists the types of processes needed in a Application Server installation and columns two through four indicate the percentage of total memory to be allocated for the type of process.

The second column of the table gives percentages for a single server system (Application Server, Oracle, Web server, and servlet engine all on one system). The third and fourth columns give percentages for a split Application Server server / Oracle server configuration.

Application / Database Process Type	**Single Server**	**Split Servers / Application Server**	**Split Servers / Oracle Server**
Application Server	21.00%	30.00%	
Oracle	21.00%		70.00%
Servlet Engine	21.00%	30.00%	
Web Server	5.00%	5.00%	
Search Engine	5.00%	5.00%	
Total Used Memory	**73.0%**	**70.0%**	**70.0%**

Note: The memory allocated to the various processes should never exceed 75%. This allows memory for other processes that are a part of normal system operation including about 15% on average for the operating system itself and 5-10% for other applications.