



Java Performance Tuning

This white paper presents the basics of Java Performance Tuning and its preferred values for large deployments of Application Servers

Java Performance Tuning

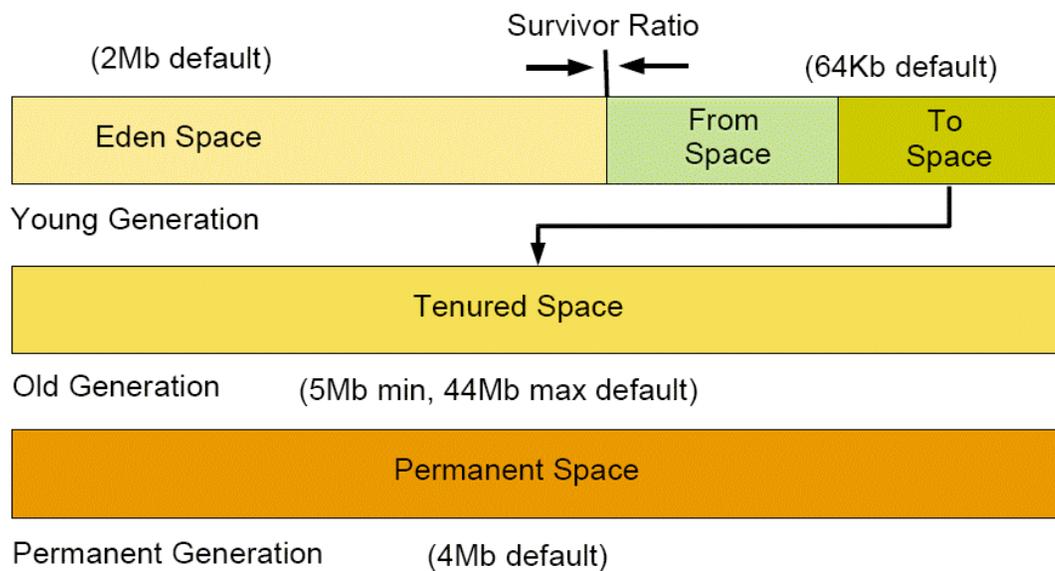
Garbage Collection

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. For best performance and stability, it is critical that the Java parameters for the virtual machine be understood and managed by the Adeptia 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 Adeptia application server.

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.

HotSpot™ VM Heap Layout



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 Scavenge 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 1.5 GB and to 1/3 of the heap for JVMs larger than 1.5 GB.

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 **Survivor Ratio**. 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 = $266.8 \text{ (Eden)} / 66.6 \text{ (To)} = 4$.

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 (OOM) 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 start up and **-Xmx** is the maximum heap size reserved by the JVM at start up. If the heap size exceeds free memory on the system, swapping will occur and performance will be seriously degraded.

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 (OOM) 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 start up 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.



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 = $-Xms + -XX:PermSize + \sim 50\text{Mb}$ and that the maximum total heap size = $-Xmx + -XX:+MaxPermSize + \sim 50\text{Mb}$. 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 but if too more Parallel GC Threads 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.

Full Garbage Collection

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

The Java application explicitly calls `System.gc()`. This can be avoided by implementing the **-XX:+DisableExplicitGC** parameter in the start up command for all Application Server JVMs.



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 start up 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 Full GC 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 Full GC 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.



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 Full GCs while ensuring that an OOME does not occur.

Serial Collector

With the serial collector, both young and old collections are done serially (using a single CPU), in a stop-the-world fashion. That is, application execution is halted while collection is taking place.

Young Generation Collection: The live objects in Eden are copied to the initially empty survivor To-space except for ones that are too large to fit comfortably in the To-space. Such objects are directly copied to the old generation. The live objects in the occupied survivor space (From-space) that are still relatively young are also copied to the other survivor space, while objects that are relatively old are copied to the old generation. Note: If the To-space becomes full, the live objects from Eden or From-space that have not been copied to it are tenured, regardless of how many young generation collections they have survived. Any objects remaining in Eden or the From-space after live objects have been copied are, by definition, not live, and they do not need to be examined.

After a young generation collection is complete, both Eden and the formerly occupied survivor space are empty and only the formerly empty survivor space contains live objects. At this point, the survivor spaces swap roles.

Old Generation Collection: The old and permanent generations are collected via a mark-sweep-compact collection algorithm. In the mark phase, the collector identifies which objects are still live. The sweep phase “sweeps” over the generations, identifying garbage. The collector then performs sliding compaction, sliding the live objects towards the beginning of the old generation space (and similarly for the permanent generation), leaving any free space in a single contiguous chunk at the opposite end.

Usage: Best suited for applications that run on client-class machines and that don’t have requirement for low pause times.

Selection: The serial collector is automatically chosen as the default garbage collector on machines that are not server-class machines. On other machines, the serial collector can be explicitly requested by using the **-XX:+UseSerialGC** command line option.

Parallel Collector

These days, many Java applications run on machines with a lot of physical memory and multiple CPUs. The parallel collector, also known as the throughput collector, was developed in order to take advantage of available CPUs rather than leaving most of them idle while only one does garbage collection work.

Young Generation Collection: The parallel collector uses a parallel version of the young generation collection algorithm utilized by the serial collector. It is still a stop-the-world and copying collector, but performing the young generation collection in parallel, using many CPUs, decreases garbage collection overhead and hence increases application throughput.

Old Generation Collection: Old generation garbage collection for the parallel collector is done using the same serial mark-sweep compact collection algorithm as the serial collector.

Usage: Applications that can benefit from the parallel collector are those that run on machines with more than one CPU and do not have pause time constraints, since infrequent, but potentially long, old generation collections will still occur.



You may want to consider choosing the parallel compacting collector (described next) over the parallel collector, since the former performs parallel collections of all generations, not just the young generation.

Selection: The parallel collector is automatically chosen as the default garbage collector on server-class machines (described later on in *Ergonomics* section). On other machines, the parallel collector can be explicitly requested by using the `-XX:+UseParallelGC` command line option.

Parallel Compacting Collector

The parallel compacting collector was introduced in J2SE 5.0 update 6. The difference between it and the parallel collector is that it uses a new algorithm for old generation garbage collection.

Young Generation Collection: Young generation garbage collection for the parallel compacting collector is done using the same algorithm as that for young generation collection using the parallel collector.

Old Generation Collection: With the parallel compacting collector, the old and permanent generations are collected in a stop-the-world, mostly parallel fashion with sliding compaction. The collector utilizes three phases. First, each generation is logically divided into fixed-sized regions. In the marking phase, the initial set of live objects directly reachable from the application code is divided among garbage collection threads, and then all live objects are marked in parallel. As an object is identified as live, the data for the region it is in is updated with information about the size and location of the object.

The summary phase operates on regions, not objects. Due to compactions from previous collections, it is typical that some portion of the left side of each generation will be dense, containing mostly live objects. The amount of space that could be recovered from such dense regions is not worth the cost of compacting them. So the first thing the summary phase does is examine the density of the regions, starting with the leftmost one, until it reaches a point where the space that could be recovered from a region and those to the right of it is worth the cost of compacting those regions. The regions to the left of that point are referred to as the dense prefix, and no objects are moved in those regions. The regions to the right of that point will be compacted, eliminating all dead space. The summary phase calculates and stores the new location of the first byte of live data for each compacted region. Note: The summary phase is currently implemented as a serial phase; parallelization is possible but not as important to performance as parallelization of the marking and compaction phases.

In the compaction phase, the garbage collection threads use the summary data to identify regions that need to be filled, and the threads can independently copy data into the regions. This produces a heap that is densely packed on one end, with a single large empty block at the other end.

Usage: Beneficial for applications that are run on machines with more than one CPU. In addition, the parallel operation of old generation collections reduces pause times and makes the parallel compacting collector more suitable than the parallel collector for applications that have pause time constraints.

Selection: By specifying the command line option `-XX:+UseParallelOldGC`.

Concurrent Mark-Sweep (CMS) Collector

For many applications, end-to-end throughput is not as important as fast response time. Young generation collections do not typically cause long pauses. However, old generation collections, though infrequent, can impose long pauses, especially when large heaps are involved. To address this issue, the HotSpot JVM includes a collector called the concurrent mark-sweep (CMS) collector, also known as the low-latency collector.

Young Generation Collection: The CMS collector collects the young generation in the same manner as the parallel collector.

Old Generation Collection: Most of the collection of the old generation using the CMS collector is done concurrently with the execution of the application.

A collection cycle for the CMS collector starts with a short pause, called the initial mark, that identifies the initial set of live objects directly reachable from the application code. Then, during the concurrent marking phase, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase. To handle this, the application stops again for a second pause, called remark, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency.

At the end of the remark phase, all live objects in the heap are guaranteed to have been marked, so the subsequent concurrent sweep phase reclaims all the garbage that has been identified

| | |
|---|--|
|  | The CMS collector is the only collector that is non-compacting. That is, after it frees the space that was occupied by dead objects, it does not move the live objects to one end of the old generation. This saves time, but since the free space is not contiguous, it leads to <i>fragmentation</i> . |
|---|--|

| | |
|---|---|
|  | Although the collector guarantees to identify all live objects during a marking phase, some objects may become garbage during that phase and they will not be reclaimed until the next old generation collection. Such objects are referred to as <i>floating garbage</i> . |
|---|---|

Usage: Use the CMS collector if your application needs shorter garbage collection pauses and can afford to share processor resources with the garbage collector when the application is running. Typically, applications that have a relatively large set of long-lived data (a large old generation), and that run on machines with two or more processors, tend to benefit from the use of this collector. The CMS collector should be considered for any application with a low pause time requirement.

Selection: By specifying the command line option `-XX:+UseConcMarkSweepGC`

Ergonomics - Automatic Selections and Behaviour Tuning

In the J2SE 5.0 and later releases, default values for the garbage collector, heap size, and Hotspot virtual machine (client or server) are automatically chosen based on the platform and operating system on which the application is running. These automatic selections better match the needs of different types of applications, while requiring fewer command line options than in previous releases.

In addition, a new way of dynamically tuning collection has been added for the parallel garbage collectors. With this approach, the user specifies the desired behavior, and the garbage collector dynamically tunes the sizes of the heap regions in an attempt to achieve the requested behavior. The combination of platform-dependent default selections and garbage collection tuning that uses desired behavior is referred to as **ergonomics**. The goal of ergonomics is to provide good performance from the JVM with a minimum of command line tuning.

Automatic Selection of Collector, Heap Sizes, and Virtual Machine

A server-class machine is defined to be one with

- 2 or more physical processors and
- 2 or more gigabytes of physical memory

This definition of a server-class machine applies to all platforms, with the exception of 32-bit platforms running a version of the Windows operating system.

On machines that are not server-class machines, the default values for JVM, garbage collector, and heap sizes are:

- Client JVM
- Serial garbage collector
- Initial heap size of 4MB
- Maximum heap size of 64MB

On a server-class machine, the JVM is always the server JVM unless you explicitly specify the `-client` command line option to request the client JVM. On a server-class machine running the server JVM, the default garbage collector is the parallel collector. Otherwise, the default is the serial collector.

On a server-class machine running either JVM (client or server) with the parallel garbage collector, the default initial and maximum heap sizes are

- Initial heap size of 1/64th of the physical memory, up to 1GB. (Note that the minimum initial heap size is 32MB, since a server-class machine is defined to have at least 2GB of memory and 1/64th of 2GB is 32MB.)
- Maximum heap size of 1/4th of the physical memory, up to 1GB.

Otherwise, the same default sizes as for non-server-class machines are used (4MB initial heap size and 64MB maximum heap size). Default values can always be overridden by command line options.

Behaviour-based Parallel Collector Tuning

In the J2SE 5.0 and later releases, a new method of tuning has been added for the parallel garbage collectors, based on desired behavior of the application with respect to garbage collection. Command line options are used to specify the desired behavior in terms of goals for maximum pause time and application throughput.

Maximum Pause Time Goal

The maximum pause time goal is specified with the command line option:

```
-XX:MaxGCPauseMillis=n
```

This is interpreted as a hint to the parallel collector that pause times of n milliseconds or less are desired. The parallel collector will adjust the heap size and other garbage collection-related parameters in an attempt to keep garbage collection pauses shorter than n milliseconds. These adjustments may cause the garbage collector to reduce overall throughput of the application, and in some cases the desired pause time goal cannot be met.

The maximum pause time goal is applied to each generation separately. Typically, if the goal is not met, the generation is made smaller in an attempt to meet the goal. No maximum pause time goal is set by default.

Throughput Goal

The throughput goal is measured in terms of the time spent doing garbage collection and the time spent outside of garbage collection (referred to as application time). The goal is specified by the command line option:

```
-XX:GCTimeRatio=n
```

The ratio of garbage collection time to application time is:

$$1 / (1 + n)$$

For example `-XX:GCTimeRatio=19` sets a goal of 5% of the total time for garbage collection. The default goal is 1% (i.e. $n=99$). The time spent in garbage collection is the total time for all generations. If the throughput goal is not being met, the sizes of the generations are increased in an effort to increase the time the application can run between collections. A larger generation takes more time to fill up.

Key Command Line Option Related to Garbage Collection

Garbage Collector Selection

| Option | Garbage Collector Selected |
|--------------------------------------|-----------------------------|
| <code>-XX:+UseSerialGC</code> | Serial |
| <code>-XX:+UseParallelGC</code> | Parallel |
| <code>-XX:+UseParallelOldGC</code> | Parallel compacting |
| <code>-XX:+UseConcMarkSweepGC</code> | Concurrent mark-sweep (CMS) |

Garbage Collector Statistics

| Option | Description |
|-------------------------------------|--|
| <code>-XX:+PrintGC</code> | Outputs basic information at every garbage collection. |
| <code>-XX:+PrintGCDetails</code> | Outputs more detailed information at every garbage collection. |
| <code>-XX:+PrintGCTimeStamps</code> | Outputs a time stamp at the start of each garbage collection event. Used with <code>-XX:+PrintGC</code> or <code>-XX:+PrintGCDetails</code> to show when each garbage collection begins. |

Options for the Parallel and Parallel Compacting Collectors

| Option | Default | Description |
|--------------------------------------|--------------------|---|
| <code>-XX:ParallelGCThreads=n</code> | The number of CPUs | Number of garbage collector threads. Start at 2 but ensure that sum of <code>ParallelGCThreads</code> across all JVMs is less than number of available CPU cores. |
| <code>-XX:MaxGCPauseMillis=n</code> | No default | Indicates to the collector that pause times of n milliseconds or less are desired. |
| <code>-XX:GCTimeRatio=n</code> | 99 | Number that sets a goal that $1/(1+n)$ of the total time be spent on garbage collection. |

Options for the CMS Collector

| Option | Default | Description |
|--------------------------------------|----------|--|
| <code>-XX:+CMSIncrementalMode</code> | Disabled | Enables a mode in which the concurrent phases are done |

| | | |
|-------------------------------|--------------------|---|
| | | incrementally, periodically stopping the concurrent phase to yield back the processor to the application. |
| -XX:+CMSIncrementalPacing | Disabled | Enables automatic control of the amount of work the CMS collector is allowed to do before giving up the processor, based on application behaviour. |
| -XX:ParallelGCThreads=n | The number of CPUs | Number of garbage collector threads for the parallel young generation collections and for the parallel parts of the old generation collections. Start at 2 but ensure that sum of ParallelGCThreads across all JVMs is less than number of available CPU cores. |
| -XX:+CMSParallelRemarkEnabled | - | Enable Parallel Remarking in CMS. |
| -XX:+UseParNewGC | - | The parallel young generation collector is similar to the parallel garbage collector (-XX:+UseParallelGC) in intent and differs in implementation. Most of the above description for the parallel garbage collector (-XX:+UseParallelGC) therefore applies equally for the parallel young generation collector. Unlike the parallel garbage collector (-XX:+UseParallelGC) this parallel young generation collector can be used with the concurrent low pause collector that collects the tenured generation. |

Heap and Generation Sizes

| Option | Default | Description |
|--------------|--------------------------------|---|
| -Xmn<x><m g> | - | Young Generation Size, suffixed with m (Mb) or g (Gb). Recommendation: For heap size < 1.5G : 1/2 of -Xmx For heap size > 1.5G : 1/3 of -Xmx For CMS: 1/4 of -Xmx |
| -Xms<x> | Platform and machine dependent | Initial size, in bytes, of the heap. |
| -Xmx<x> | Platform and | Maximum size, in bytes, of the |

| | machine dependent | heap. |
|---|----------------------------------|--|
| -XX:MinHeapFreeRatio=minimum and -XX:MaxHeapFreeRatio=maximum | 99 | Target range for the proportion of free space to total heap size. These are applied per generation. For example, if minimum is 30 and the percent of free space in a generation falls below 30%, the size of the generation is expanded so as to have 30% of the space free. Similarly, if maximum is 60 and the percent of free space exceeds 60%, the size of the generation is shrunk so as to have only 60% of the space free. |
| -XX:NewSize=n | Platform-dependent | Default initial size of the new (young) generation, in bytes. |
| -XX:NewRatio=n | 2 on client JVM, 8 on server JVM | Ratio between the young and old generations. For example, if n is 3, then the ratio is 1:3 and the combined size of Eden and the survivor spaces is one fourth of the total size of the young and old generations. |
| -XX:SurvivorRatio=n | 32 | Ratio between each survivor space and Eden. For example, if n is 7, each survivor space is one-ninth of the young generation (not one-eighth, because there are two survivor spaces). |
| -XX:MaxPermSize=n | Platform-dependent | Maximum size of the permanent generation. Set it to 128m. (keep initial and maximum permanent generation size same to discourage address map swapping) |

Other useful command line options

| Option | Default | Description |
|------------------------|---------|---------------------------------|
| -XX:+DisableExplicitGC | - | Ignore all calls to System.gc() |