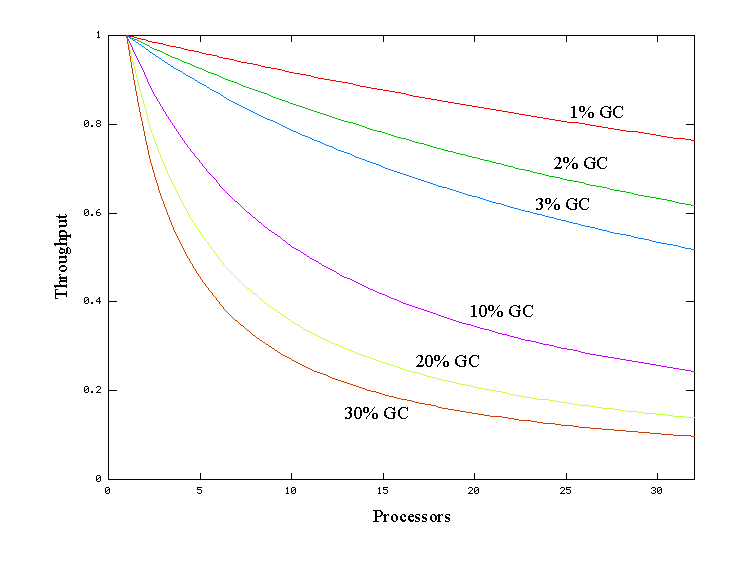
### In the J2SE platform version 1.4.2 there were four garbage collectors from which to choose but without an explicit choice by the user the serial garbage collector was always chosen. In version 5.0 the choice of the collector is based on the class of the machine on which the application is started.

The graph below models an ideal system that is perfectly scalable with the exception of garbage collection. The red line is an application spending only 1% of the time in garbage collection on a uniprocessor system. This translates to more than a 20% loss in throughput on 32 processor systems. At 10% of the time in garbage collection (not considered an outrageous amount of time in garbage collection in uniprocessor applications) more than 75% of throughput is lost when scaling up to 32 processors.



### The serial collector will be adequate for the majority of applications. Each of the other collectors have some added overhead and/or complexity which is the price for specialized behavior. If the application doesn't need the specialized behavior of an alternate collector, use the serial collector. An example of a situation where the serial collector is not expected to be the best choice is a large application that is heavily threaded and run on hardware with a large amount of memory and a large number of processors. For such applications, we now make the choice of the throughput collector.

### Young and old generation

### space usage by generations

At initialization, a maximum address space is virtually reserved but not allocated to physical memory unless it is needed. The complete address space reserved for object memory can be divided into the *young* and *tenured* generations.

The *young* generation consists of *eden* plus two *survivor* spaces . Objects are initially allocated in eden. One *survivor* space is empty at any time, and serves as a destination of the next, copying collection of any live objects in eden and the other survivor space. Objects are copied between survivor spaces in this way until they are old enough to be tenured, or copied to the *tenured* generation.

Other virtual machines, including the production virtual machine for the J2SE Platform version 1.2 for the Solaris Operating System, used two equally sized spaces for copying rather than one large eden plus two small spaces. This means the options for sizing the *young* generation are not directly comparable; see the [Performance FAQ](http://java.sun.com/docs/hotspot/PerformanceFAQ.html) for an example.

A third generation closely related to the *tenured* generation is the *permanent* generation. The *permanent generation* is special because it holds data needed by the virtual machine to describe objects that do not have an equivalence at the Java language level. For example objects describing classes and methods are stored in the *permanent generation*

### Performance Considerations

### Throughput is the percentage of total time not spent in garbage collection, considered over long periods of time. Throughput includes time spent in allocation (but tuning for speed of allocation is generally not needed.)

### Pauses are the times when an application appears unresponsive because garbage collection is occurring.

### Footprint: a footprint is the amount of space a particular unit of hardware or software occupies

### Promptness is the time between when an object becomes dead and when the memory becomes available.

### Measurement

The command line argument -verbose:gc prints information at every collection. Note that the format of the -verbose:gc output is subject to change between releases of the J2SE platform. For example, here is output from a large server application:

  [GC 325407K->83000K(776768K), 0.2300771 secs]   
  [GC 325816K->83372K(776768K), 0.2454258 secs]   
  [Full GC 267628K->83769K(776768K), 1.8479984 secs]

Here we see two minor collections and one major one. The numbers before and after the arrow

325407K->83000K (in the first line)

indicate the combined size of live objects before and after garbage collection, respectively. After minor collections the count includes objects that aren't necessarily alive but can't be reclaimed, either because they are directly alive, or because they are within or referenced from the *tenured* generation. The number in parenthesis

(776768K)(in the first line)

is the total available space, not counting the space in the *permanent* generation, which is the total heap minus one of the survivor spaces. The minor collection took about a quarter of a second.

0.2300771 secs (in the first line)

The format for the major collection in the third line is similar. The flag -XX:+PrintGCDetails prints additional information about the collections. The additional information printed with this flag is liable to change with each version of the virtual machine. The additional output with the -XX:+PrintGCDetails flag in particular changes with the needs of the development of the Java Virtual Machine. An example of the output with -XX:+PrintGCDetails for the J2SE Platform version 1.5 using the serial garbage collector is shown here.

[GC [DefNew: 64575K->959K(64576K), 0.0457646 secs] 196016K->133633K(261184K), 0.0459067 secs]]

indicates that the minor collection recovered about 98% of the *young* generation,

DefNew: 64575K->959K(64576K)

and took about 46 milliseconds.

0.0457646 secs

The usage of the entire heap was reduced to about 51%

196016K->133633K(261184K)

and that there was some slight additional overhead for the collection (over and above the collection of the *young* generation) as indicated by the final time:

0.0459067 secs

The flag -XX:+PrintGCTimeStamps will additionally print a time stamp at the start of each collection.

111.042: [GC 111.042: [DefNew: 8128K->8128K(8128K), 0.0000505 secs]111.042: [Tenured: 18154K->2311K(24576K), 0.1290354 secs] 26282K->2311K(32704K), 0.1293306 secs]

The collection starts about 111 seconds into the execution of the application. The minor collection starts at about the same time. Additionally the information is shown for a major collection delineated by Tenured. The *tenured* generation usage was reduced to about 10%

18154K->2311K(24576K)

and took about .13 seconds.

0.1290354 secs

### Sizing the Generations

NewRatio: NewRatio denotes the relative size of the tenured generation to the young generation

NewSize : Minimum young generation size.

MaxNewSize : Maximum young generation size.

SurvivorRatio: Ratio of each survivor space to eden. (Default 32)

-XX:MinHeapFreeRatio: Ratio of minimum free space to live objects (recommended 40)

-XX:MaxHeapFreeRatio: Ratio of maximum free space to live objects (recommended 70)

With these parameters if the percent of free space in a generation falls below 40%, the size of the generation will be expanded so as to have 40% of the space free, assuming the size of the generation has not already reached its limit. Similarly, if the percent of free space exceeds 70%, the size of the generation will be shrunk so as to have only 70% of the space free as long as shrinking the generation does not decrease it below the minimum size of the generation.

Unless you have problems with pauses, try granting as much memory as possible to the virtual machine. The default size (64MB) is often too small.

Setting -Xms and -Xmx to the same value increases predictability by removing the most important sizing decision from the virtual machine. On the other hand, the virtual machine can't compensate if you make a poor choice.

Be sure to increase the memory as you increase the number of processors, since allocation can be parallelized.

If survivor spaces are too small, copying collection overflows directly into the *tenured* generation. If survivor spaces are too large, they will be uselessly empty. At each garbage collection the virtual machine chooses a threshold number of times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full. An option, -XX:+PrintTenuringDistribution, can be used to show this threshold and the ages of objects in the new generation. It is also useful for observing the lifetime distribution of an application

First decide the total amount of memory you can afford to give the virtual machine. Then graph your own performance metric against *young* generation sizes to find the best setting.

Unless you find problems with excessive major collection or pause times, grant plenty of memory to the *young* generation.

Increasing the *young* generation becomes counterproductive at half the total heap or less (whenever the *young* generation guarantee cannot be met).

Be sure to increase the *young* generation as you increase the number of processors, since allocation can be parallelized

### Types of garbage collection:

### Serial collector

### Throughput garbage collector. (-XX:+UseParallelGC)

### Concurrent low pause collector (XX:+UseConcMarkSweepGC)

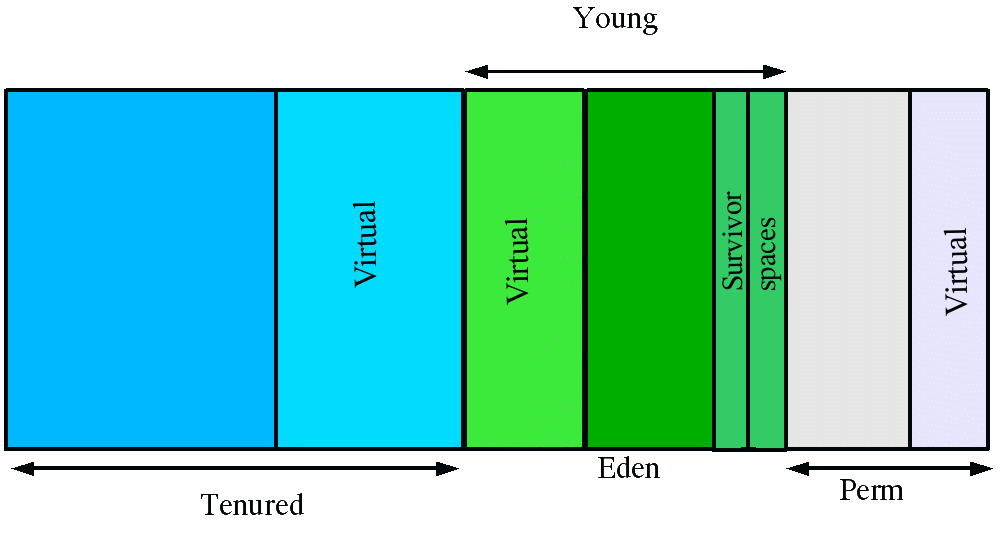
### Incremental collector (-XX:+UseTrainGC)

### Throughput garbage collector

Use the throughput collector when you want to improve the performance of your application with larger numbers of processors. In the serial collector garbage collection is done by one thread, and therefore garbage collection adds to the serial execution time of the application. The throughput collector uses multiple threads to execute a minor collection and so reduces the serial execution time of the application. A typical situation is one in which the application has a large number of threads allocating objects. In such an application it is often the case that a large *young* generation is needed.

Number of the gc threads can be controlled by (-XX:ParallelGCThreads=<desired number>).

#### Generations in the throughput collector

As mentioned earlier the arrangement of the generations is different in the throughput collector. That arrangement is shown in the figure below.

The maximum pause time goals is specified with the command line flag

-XX:MaxGCPauseMillis=<nnn>

This is interpreted as a hint to the throughput collector that pause times of <nnn> milliseconds or less are desired. By default there is no maximum pause time goal. The throughput collector will adjust the Java heap size and other garbage collection related parameters in an attempt to keep garbage collection pauses shorter than <nnn> 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. By default no maximum pause time goal is set.

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 flag

-XX:GCTimeRatio=<nnn>

The ratio of garbage collection time to application time is

1 / (1 + <nnn>)

### The Concurrent Low Pause Collector

The concurrent low pause collector is a generational collector similar to the serial collector. The *tenured* generation is collected concurrently with this collector.

This collector attempts to reduce the pause times needed to collect the *tenured* generation. It uses a separate garbage collector thread to do parts of the major collection concurrently with the applications threads. The concurrent collector is enabled with the command line option -XX:+UseConcMarkSweepGC. For each major collection the concurrent collector will pause all the application threads for a brief period at the beginning of the collection and toward the middle of the collection. The second pause tends to be the longer of the two pauses and multiple threads are used to do the collection work during that pause. The remainder of the collection is done with a garbage collector thread that runs concurrently with the application. The minor collections are done in a manner similar to the serial collector although multiple threads are used to do the collection. See "Parallel Minor Collection Options with the Concurrent Collector" below for information on using multiple threads with the concurrent low pause collector.

### *Young* Generation Guarantee

### Guarantee that when a young generation collection starts is enough space in the tenured generation to hold all the objects that require promotion from the young generation.

### Other considerations when scaling to a large number of threads

Besides the threading model, there are other things you may want to consider when moving to a large number of threads, namely:

* The thread stack size
* Thread local heap
* Garbage collection affects
* Using Intimate Shared Memory

The default thread stack size is quite large: 512kb on Sparc and 256kb on Intel for 1.3 and 1.4 32-bit VMs, 1mb with the 64-bit Sparc 1.4 VM; and 128k for 1.2 VMs. If you have many threads (in the thousands) then you can waste a significant amount of stack space. The minimum setting in 1.3 and 1.4 is 64k, and in 1.2 is 32k, which you can change via the *-Xss* flag.

HotSpot Server VM

The Java HotSpot Server VM is designed for maximum program execution speed for applications running in a server environment. The Java HotSpot Server VM is invoked by using the -

### How are Java Stack Traces Generated?

There are three popular ways to generate a Java stack trace: sending a signal to the Java Virtual Machine; the Java Virtual Machine generates a stack trace for you; or using debugging tools or Java API calls.

**Sending a signal to the Java Virtual Machine**

On UNIX platforms you can send a signal to a program by using the kill command. This is the quit signal, which is handled by the JVM. For example, on Solaris you can use the command kill -QUIT process\_id, where process\_id is the process number of your Java program.

Alternatively you can enter the key sequence <ctrl>\ in the window where the Java program was started. Sending this signal instructs a signal handler in the JVM, to recursively print out all the information on the threads and monitors inside the JVM.

Java VM parameters.

-XX:MinHeapFreeRatio = minimum free space / Live objects

-XX:MaxHeapFreeRatio= maximum free space / Live objects

-Xms = Initial heap size

-Xmx = Maximum heap size.

NewRation= Young generation / Tenured generation

Survivor ratio = Survivor space / Eden space

## 4 Types of Collectors

The discussion to this point has been about the default collector. In the J2SE platform, version 1.4.2 there are three additional collectors. Each is a generational collector which has been implemented to emphasize the throughput of the application or low garbage collection pause times.

1. The*throughput* collector: this collector uses a parallel version of the *young* generation collector. It is used if the -XX:+UseParallelGC option is passed on the command line. The *tenured* generation collector is the same as the default collector.
2. The*concurrent* low pause collector: this collector is used if the -XX:+UseConcMarkSweepGC is passed on the command line. The concurrent collector is used to collect the *tenured* generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection. A parallel version of the *young* generation copying collector is used with the concurrent collector if the combination -XX:+UseConcMarkSweepGC -XX:+UseParNewGC is passed on the command line.
3. The*incremental*(sometimes called *train*) low pause collector: this collector is used only if -Xincgc is passed on the command line. By careful bookkeeping, the incremental garbage collector collects just a portion of the *tenured* generation at each minor collection, trying to spread the large pause of a major collection over many minor collections. However, it is even slower than the default *tenured* generation collector when considering overall throughput.

The performance impact of explicit garbage collections can be measured by disabling explicit garbage collections using the flag -XX:+DisableExplicitGC.

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

### Young generation garbage collection algorithms

* The (original) copying collector (Enabled by default). When this collector kicks in, all application threads are stopped, and the copying collection proceeds using one thread (which means only one CPU even if on a multi-CPU machine). This is known as a stop-the-world collection, because basically the JVM pauses everything else until the collection is completed.
* The parallel copying collector (Enabled using -XX:+UseParNewGC). Like the original copying collector, this is a stop-the-world collector. However this collector parallelizes the copying collection over multiple threads, which is more efficient than the original single-thread copying collector for multi-CPU machines (though not for single-CPU machines). This algorithm potentially speeds up young generation collection by a factor equal to the number of CPUs available, when compared to the original singly-threaded copying collector.
* The parallel scavenge collector (Enabled using -XX:UseParallelGC). This is like the previous parallel copying collector, but the algorithm is tuned for gigabyte heaps (over 10GB) on multi-CPU machines. This collection algorithm is designed to maximize throughput while minimizing pauses. It has an optional adaptive tuning policy which will automatically resize heap spaces. If you use this collector, you can only use the the original mark-sweep collector in the old generation (i.e. the newer old generation concurrent collector cannot work with this young generation collector).

### Old generation garbage collection algorithms

* The (original) mark-sweep collector (Enabled by default). This uses a stop-the-world mark-and-sweep collection algorithm. The collector is single-threaded, the entire JVM is paused and the collector uses only one CPU until completed.
* The concurrent collector (Enabled using -XX:+UseConcMarkSweepGC). This collector tries to allow application processing to continue as much as possible during the collection. Splitting the collection into six phases described shortly, four are concurrent while two are stop-the-world:  
  1. the initial-mark phase (stop-the-world, snapshot the old generation so that we can run most of the rest of the collection concurrent to the application threads);  
  2. the mark phase (concurrent, mark the live objects traversing the object graph from the roots);  
  3. the pre-cleaning phase (concurrent);  
  4. the re-mark phase (stop-the-world, another snapshot to capture any changes to live objects since the collection started);  
  5. the sweep phase (concurrent, recycles memory by clearing unreferenced objects);  
  6. the reset phase (concurrent).  
  If "the rate of creation" of objects is too high, and the concurrent collector is not able to keep up with the concurrent collection, it falls back to the traditional mark-sweep collector.
* The incremental collector (Enabled using -Xincgc). The incremental collector uses a "train" algorithm to collect small portions of the old generation at a time. This collector has higher overheads than the mark-sweep collector, but because small numbers of objects are collected each time, the (stop-the-world) garbage collection pause is minimized at the cost of total garbage collection taking longer. The "train" algorithm does not guarantee a maximum pause time, but pause times are typically less than ten milliseconds.

Clustering for performance and scalability

A WebLogic Server cluster consists of multiple WebLogic Server server instances running simultaneously and working together to provide increased scalability and reliability

A domain is an interrelated set of WebLogic Server resources that are managed as a unit. A domain includes one or more WebLogic Server instances, which can be clustered, non-clustered, or a combination of clustered and non-clustered instances. A domain can include multiple clusters. A domain also contains the application components deployed in the domain, and the resources and services required by those application components and the server instances in the domain. Examples of the resources and services used by applications and server instances include machine definitions, optional network channels, connectors, and startup classes.

A WebLogic Server cluster provides these benefits:

* Scalability
* High-Availability

## Key Capabilities of a Cluster

* Application Failover
* Load balancing is the even distribution of jobs and associated communications across the computing and networking resources in your environment.