Preface

The *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* describes the garbage collection methods included in the Java HotSpot Virtual Machine (Java HotSpot VM) and helps you determine which one is the best for your needs.

Audience

This document is intended for application developers and system administrators who want to improve the performance of applications, especially those that handle large amounts of data, use many threads, and have high transaction rates.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents:

- Java Virtual Machine Technology
  `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/index.html`
- Java SE HotSpot at a Glance
  `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`
- HotSpot VM Frequently Asked Questions (FAQ)

  `http://www.oracle.com/technetwork/java/hotspotfaq-138619.html`
- How to Handle Java Finalization's Memory-Retention Issues: Covers finalization pitfalls and ways to avoid them.

  `http://www.devx.com/Java/Article/30192`
- Richard Jones and Rafael Lins, *Garbage Collection: Algorithms for Automated Dynamic Memory Management*, Wiley and Sons (1996), ISBN 0-471-94148-4

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction

A wide variety of applications use Java Platform, Standard Edition (Java SE), from small applets on desktops to web services on large servers. In support of this diverse range of deployments, the Java HotSpot virtual machine implementation (Java HotSpot VM) provides multiple garbage collectors, each designed to satisfy different requirements. This is an important part of meeting the demands of both large and small applications. Java SE selects the most appropriate garbage collector based on the class of the computer on which the application is run. However, this selection may not be optimal for every application. Users, developers, and administrators with strict performance goals or other requirements may need to explicitly select the garbage collector and tune certain parameters to achieve the desired level of performance. This document provides information to help with these tasks. First, general features of a garbage collector and basic tuning options are described in the context of the serial, stop-the-world collector. Then specific features of the other collectors are presented along with factors to consider when selecting a collector.

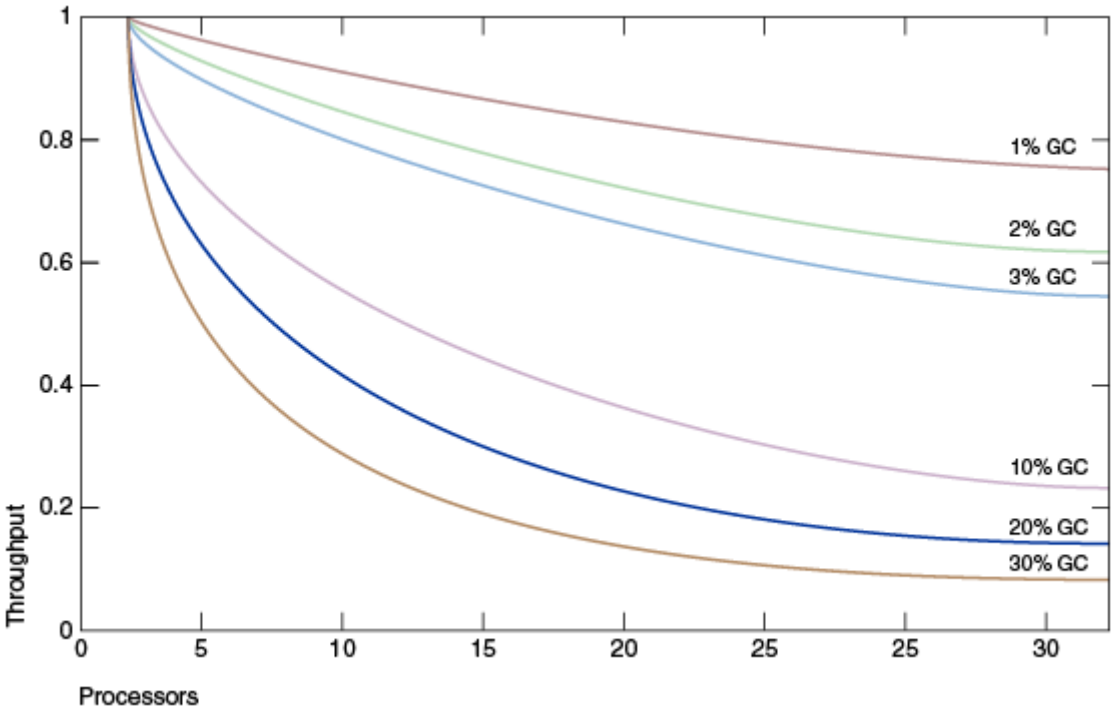
A garbage collector (GC) is a memory management tool. It achieves automatic memory management through the following operations:

- Allocating objects to a young generation and promoting aged objects into an old generation.
- Finding live objects in the old generation through a concurrent (parallel) marking phase. The Java HotSpot VM triggers the marking phase when the total Java heap occupancy exceeds the default threshold. See the sectionsConcurrent Mark Sweep (CMS) Collector and Garbage-First Garbage Collector.
- Recovering free memory by compacting live objects through parallel copying. See the sections The Parallel Collector and Garbage-First Garbage Collector

When does the choice of a garbage collector matter? For some applications, the answer is never. That is, the application can perform well in the presence of garbage collection with pauses of modest frequency and duration. However, this is not the case for a large class of applications, particularly those with large amounts of data (multiple gigabytes), many threads, and high transaction rates.

Amdahl's law (parallel speedup in a given problem is limited by the sequential portion of the problem) implies that most workloads cannot be perfectly parallelized; some portion is always sequential and does not benefit from parallelism. This is also true for the Java platform. In particular, virtual machines from Oracle for the Java platform prior to Java SE 1.4 do not support parallel garbage collection, so the effect of garbage collection on a multiprocessor system grows relative to an otherwise parallel application.

Figure 1-1 Comparing Percentage of Time Spent in Garbage Collection



Description of "Figure 1-1 Comparing Percentage of Time Spent in Garbage Collection"

This shows that negligible speed issues when developing on small systems may become principal bottlenecks when scaling up to large systems. However, small improvements in reducing such a bottleneck can produce large gains in performance. For a sufficiently large system, it becomes worthwhile to select the right garbage collector and to tune it if necessary.

The serial collector is usually adequate for most "small" applications (those requiring heaps of up to approximately 100 megabytes (MB (on modern processors). The other collectors have additional overhead or complexity, which is the price for specialized behavior. If the application does not need the specialized behavior of an alternate collector, use the serial collector. One situation where the serial collector is not expected to be the best choice is a large, heavily threaded application that runs on a machine with a large amount of memory and two or more processors. When applications are run on such server-class machines, the parallel collector is selected by default. See the sectionErgonomics.

This document was developed using Java SE 8 on the Solaris operating system (SPARC Platform Edition) as the reference. However, the concepts and recommendations presented here apply to all supported platforms, including Linux, Microsoft Windows, the Solaris operating system (x64 Platform Edition), and OS X. In addition, the command line options mentioned are available on all supported platforms, although the default values of some options may be different on each platform.

2 Ergonomics

Ergonomics is the process by which the Java Virtual Machine (JVM) and garbage collection tuning, such as behavior-based tuning, improve application performance. The JVM provides platform-dependent default selections for the garbage collector, heap size, and runtime compiler. These selections match the needs of different types of applications while requiring less command-line tuning. In addition, behavior-based tuning dynamically tunes the sizes of the heap to meet a specified behavior of the application.

This section describes these default selections and behavior-based tuning. Use these defaults first before using the more detailed controls described in subsequent sections.

Garbage Collector, Heap, and Runtime Compiler Default Selections

A class of machine referred to as a server-class machine has been defined as a machine with the following:

- 2 or more physical processors
- 2 or more GB of physical memory

On server-class machines, the following are selected by default:

- Throughput garbage collector
- Initial heap size of 1/64 of physical memory up to 1 GB
- Maximum heap size of 1/4 of physical memory up to 1 GB
- Server runtime compiler

For initial heap and maximum heap sizes for 64-bit systems, see the section Default Heap Size in The Parallel Collector.

The 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. Table 2-1, "Default Runtime Compiler", shows the choices made for the runtime compiler for different platforms.

Table 2-1 Default Runtime Compiler

Platform	Operating System	Default[Foot1]	Default if Server-Class[Footref1]
i586	Linux	Client	Server
i586	Windows	Client	Client[Foot2]
SPARC (64-bit)	Solaris	Server	Server[Foot3]
AMD (64-bit)	Linux	Server	Server[Foot3]
AMD (64-bit)	Windows	Server	Server[Foot3]

[Footnote1] Client means the client runtime compiler is used. Server means the server runtime compiler is used.

[Footnote2] The policy was chosen to use the client runtime compiler even on a server class machine. This choice was made because historically client applications (for example, interactive applications) were run more often on this combination of platform and operating system.

[Footnote3] Only the server runtime compiler is supported.

Behavior-Based Tuning

For the parallel collector, Java SE provides two garbage collection tuning parameters that are based on achieving a specified behavior of the application: maximum pause time goal and application throughput goal; see the sectionThe Parallel Collector.

(These two options are not available in the other collectors.) Note that these behaviors cannot always be met. The application requires a heap large enough to at least hold all of the live data. In addition, a minimum heap size may preclude reaching these desired goals.

Maximum Pause Time Goal

The pause time is the duration during which the garbage collector stops the application and recovers space that is no longer in use. The intent of the maximum pause time goal is to limit the longest of these pauses. An average time for pauses and a variance on that average is maintained by the garbage collector. The average is taken from the start of the execution but is weighted so that more recent pauses count more heavily. If the average plus the variance of the pause times is greater than the maximum pause time goal, then the garbage collector considers that the goal is not being met.

The maximum pause time goal is specified with the command-line option `-XX:MaxGCPauseMillis=<nnn>`. This is interpreted as a hint to the garbage collector that pause times of `<nnn>` milliseconds or less are desired. The garbage collector will adjust the Java heap size and other parameters related to garbage collection in an attempt to keep garbage collection pauses shorter than `<nnn>` milliseconds. By default there is no maximum pause time goal. These adjustments may cause garbage collector to occur more frequently, reducing the overall throughput of the application. The garbage collector tries to meet any pause time goal before the throughput goal. In some cases, though, the desired pause time goal cannot be met.

Throughput Goal

The throughput goal is measured in terms of the time spent collecting garbage and the time spent outside of garbage collection (referred to as *application time*). The goal is specified by the command-line option `-XX:GCTimeRatio=<nnn>`. The ratio of garbage collection time to application time is 1 / (1 + `<nnn>`). For example, `-XX:GCTimeRatio=19` sets a goal of 1/20th or 5% of the total time for garbage collection.

The time spent in garbage collection is the total time for both the young generation and old generation collections combined. If the throughput goal is not being met, then the sizes of the generations are increased in an effort to increase the time that the application can run between collections.

Footprint Goal

If the throughput and maximum pause time goals have been met, then the garbage collector reduces the size of the heap until one of the goals (invariably the throughput goal) cannot be met. The goal that is not being met is then addressed.

Tuning Strategy

Do not choose a maximum value for the heap unless you know that you need a heap greater than the default maximum heap size. Choose a throughput goal that is sufficient for your application.

The heap will grow or shrink to a size that will support the chosen throughput goal. A change in the application's behavior can cause the heap to grow or shrink. For example, if the application starts allocating at a higher rate, the heap will grow to maintain the same throughput.

If the heap grows to its maximum size and the throughput goal is not being met, the maximum heap size is too small for the throughput goal. Set the maximum heap size to a value that is close to the total physical memory on the platform but which does not cause swapping of the application. Execute the application again. If the throughput goal is still not met, then the goal for the application time is too high for the available memory on the platform.

If the throughput goal can be met, but there are pauses that are too long, then select a maximum pause time goal. Choosing a maximum pause time goal may mean that your throughput goal will not be met, so choose values that are an acceptable compromise for the application.

It is typical that the size of the heap will oscillate as the garbage collector tries to satisfy competing goals. This is true even if the application has reached a steady state. The pressure to achieve a throughput goal (which may require a larger heap) competes with the goals for a maximum pause time and a minimum footprint (which both may require a small heap).

3 Generations

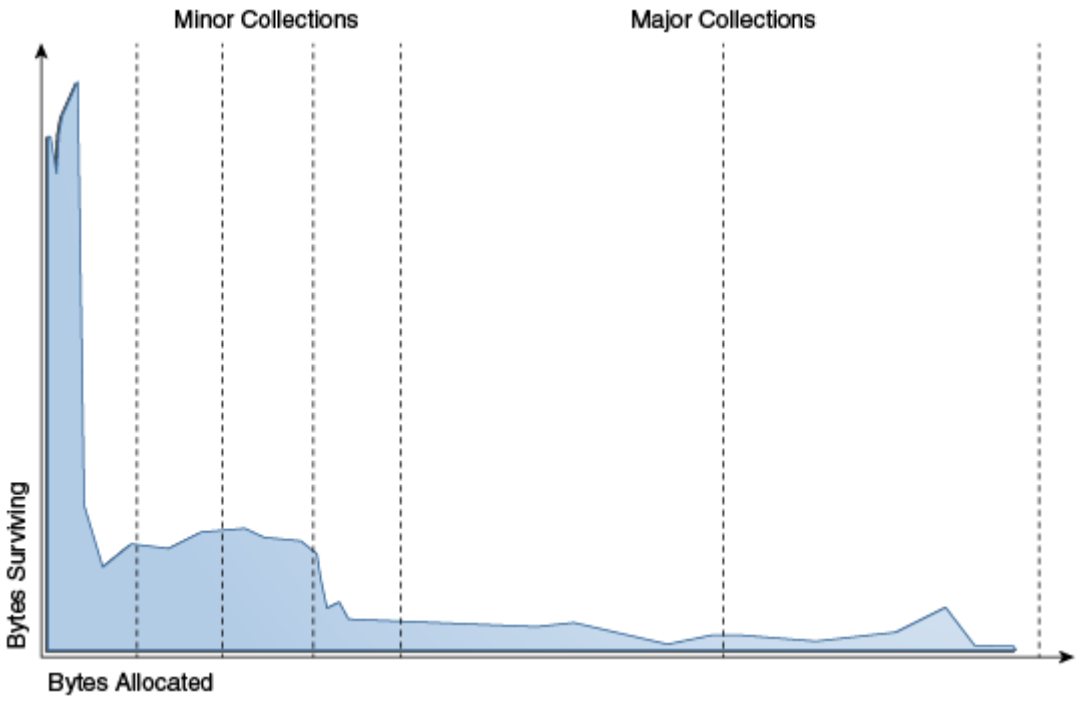
One strength of the Java SE platform is that it shields the developer from the complexity of memory allocation and garbage collection. However, when garbage collection is the principal bottleneck, it is useful to understand some aspects of this hidden implementation. Garbage collectors make assumptions about the way applications use objects, and these are reflected in tunable parameters that can be adjusted for improved performance without sacrificing the power of the abstraction.

An object is considered garbage when it can no longer be reached from any pointer in the running program. The most straightforward garbage collection algorithms iterate over every reachable object. Any objects left over are considered garbage. The time this approach takes is proportional to the number of live objects, which is prohibitive for large applications maintaining lots of live data.

The virtual machine incorporates a number of different garbage collection algorithms that are combined using *generational collection*. While naive garbage collection examines every live object in the heap, generational collection exploits several empirically observed properties of most applications to minimize the work required to reclaim unused (garbage) objects. The most important of these observed properties is the weak *generational hypothesis*, which states that most objects survive for only a short period of time.

The blue area in Figure 3-1, "Typical Distribution for Lifetimes of Objects" is a typical distribution for the lifetimes of objects. The x-axis is object lifetimes measured in bytes allocated. The byte count on the y-axis is the total bytes in objects with the corresponding lifetime. The sharp peak at the left represents objects that can be reclaimed (in other words, have "died") shortly after being allocated. Iterator objects, for example, are often alive for the duration of a single loop.

Figure 3-1 Typical Distribution for Lifetimes of Objects



Description of "Figure 3-1 Typical Distribution for Lifetimes of Objects"

Some objects do live longer, and so the distribution stretches out to the right. For instance, there are typically some objects allocated at initialization that live until the process exits. Between these two extremes are objects that live for the duration of some intermediate computation, seen here as the lump to the right of the initial peak. Some applications have very different looking distributions, but a surprisingly large number possess this general shape. Efficient collection is made possible by focusing on the fact that a majority of objects "die young."

To optimize for this scenario, memory is managed in *generations* (memory pools holding objects of different ages). Garbage collection occurs in each generation when the generation fills up. The vast majority of objects are allocated in a pool dedicated to young objects (the *young generation*), and most objects die there. When the young generation fills up, it causes a *minor collection* in which only the young generation is collected; garbage in other generations is not reclaimed. Minor collections can be

optimized, assuming that the weak generational hypothesis holds and most objects in the young generation are garbage and can be reclaimed. The costs of such collections are, to the first order, proportional to the number of live objects being collected; a young generation full of dead objects is collected very quickly. Typically, some fraction of the surviving objects from the young generation are moved to the tenured generation during each minor collection. Eventually, the *tenured generation* will fill up and must be collected, resulting in a *major collection*, in which the entire heap is collected. Major collections usually last much longer than minor collections because a significantly larger number of objects are involved.

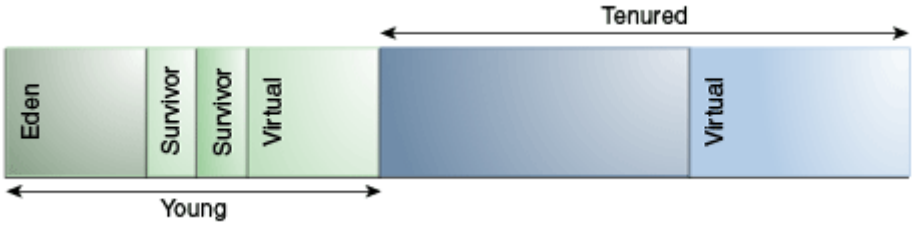
As noted in the section Ergonomics, ergonomics selects the garbage collector dynamically to provide good performance on a variety of applications. The serial garbage collector is designed for applications with small data sets, and its default parameters were chosen to be effective for most small applications. The parallel or throughput garbage collector is meant to be used with applications that have medium to large data sets. The heap size parameters selected by ergonomics plus the features of the adaptive size policy are meant to provide good performance for server applications. These choices work well in most, but not all, cases, which leads to the central tenet of this document:

Note:

If garbage collection becomes a bottleneck, you will most likely have to customize the total heap size as well as the sizes of the individual generations. Check the verbose garbage collector output and then explore the sensitivity of your individual performance metric to the garbage collector parameters.

Figure 3-2, "Default Arrangement of Generations, Except for Parallel Collector and G1" shows the default arrangement of generations (for all collectors with the exception of the parallel collector and G1):

Figure 3-2 Default Arrangement of Generations, Except for Parallel Collector and G1



Description of "Figure 3-2 Default Arrangement of Generations, Except for Parallel Collector and G1"

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 and two survivor spaces. Most objects are initially allocated in eden. One survivor space is empty at any time, and serves as the destination of any live objects in eden; the other survivor space is the destination during the next copying collection. Objects are copied between survivor spaces in this way until they are old enough to be tenured (copied to the tenured generation).

Performance Considerations

There are two primary measures of garbage collection performance:

- *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.

Users have different requirements of garbage collection. For example, some consider the right metric for a web server to be throughput because pauses during garbage collection may be tolerable or simply obscured by network latencies. However, in an interactive graphics program, even short pauses may negatively affect the user experience.

Some users are sensitive to other considerations. *Footprint* is the working set of a process, measured in pages and cache lines. On systems with limited physical memory or many processes, footprint may dictate scalability.*Promptness* is the time between when an object becomes dead and when the memory becomes available, an important consideration for distributed systems, including Remote Method Invocation (RMI).

In general, choosing the size for a particular generation is a trade-off between these considerations. For example, a very large young generation may maximize throughput, but does so at the expense of footprint, promptness, and pause times. Young generation pauses can be minimized by using a small young generation at the expense of throughput. The sizing of one generation does not affect the collection frequency and pause times for another generation.

There is no one right way to choose the size of a generation. The best choice is determined by the way the application uses memory as well as user requirements. Thus the virtual machine's choice of a garbage collector is not always optimal and may be overridden with command-line options described in the section Sizing the Generations.

Measurement

Throughput and footprint are best measured using metrics particular to the application. For example, the throughput of a web server may be tested using a client load generator, whereas the footprint of the server may be measured on the Solaris operating system using the `pmap` command. However, pauses due to garbage collection are easily estimated by inspecting the diagnostic output of the virtual machine itself.

The command-line option `-verbose:gc` causes information about the heap and garbage collection to be printed at each collection. 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]
```

The output shows two minor collections followed by one major collection. The numbers before and after the arrow (for example, `325407K->83000K` from the first line) indicate the combined size of live objects before and after garbage collection, respectively. After minor collections, the size includes some objects that are garbage (no longer alive) but cannot be reclaimed. These objects are either contained in the tenured generation or referenced from the tenured generation.

The next number in parentheses (for example, `(776768K)` again from the first line) is the committed size of the heap: the amount of space usable for Java objects without requesting more memory from the operating system. Note that this number only includes one of the survivor spaces. Except during a garbage collection, only one survivor space will be used at any given time to store objects.

The last item on the line (for example, `0.2300771 secs`) indicates the time taken to perform the collection, which is in this case approximately a quarter of a second.

The format for the major collection in the third line is similar.

Note:

The format of the output produced by `-verbose:gc` is subject to change in future releases.

The command-line option `-XX:+PrintGCDetails` causes additional information about the collections to be printed. An example of the output with `-XX:+PrintGCDetails` using the serial garbage collector is shown here.

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

This indicates that the minor collection recovered about 98% of the young generation, `DefNew: 64575K->959K(64576K)` and took `0.0457646 secs` (about 45 milliseconds).

The usage of the entire heap was reduced to about 51% (`196016K->133633K(261184K)`), and there was some slight additional overhead for the collection (over and above the collection of the young generation) as indicated by the final time of `0.0459067 secs`.

> Note:
>
> **The format of the output produced by `-XX:+PrintGCDetails` is subject to change in future releases.**

The option `-XX:+PrintGCTimeStamps` adds a time stamp at the start of each collection. This is useful to see how frequently garbage collections occur.

```
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]
```

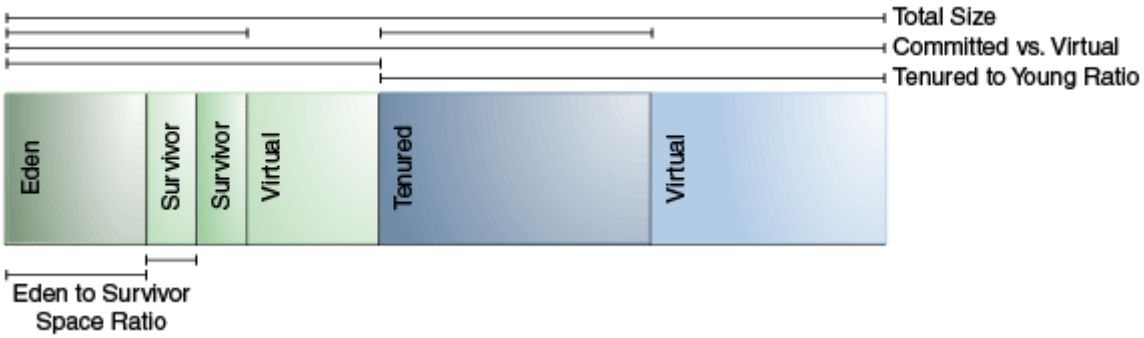
The collection starts about 111 seconds into the execution of the application. The minor collection starts at about the same time. Additionally, the information is shown for a major collection delineated by Tenured. The tenured generation usage was reduced to about 10% (`18154K->2311K(24576K)`) and took `0.1290354 secs` (approximately 130 milliseconds).

4 Sizing the Generations

A number of parameters affect generation size. Figure 4-1, "Heap Parameters" illustrates the difference between committed space and virtual space in the heap. At initialization of the virtual machine, the entire space for the heap is reserved. The size of the space reserved can be specified with the `-Xmx` option. If the value of the `-Xms` parameter is smaller than the value of the `-Xmx` parameter, than not all of the space that is reserved is immediately committed to the virtual machine. The uncommitted space is labeled "virtual" in this figure. The different parts of the heap (tenured generation and young generation) can grow to the limit of the virtual space as needed.

Some of the parameters are ratios of one part of the heap to another. For example the parameter `NewRatio` denotes the relative size of the tenured generation to the young generation.

Figure 4-1 Heap Parameters



Description of "Figure 4-1 Heap Parameters"

Total Heap

The following discussion regarding growing and shrinking of the heap and default heap sizes does not apply to the parallel collector. (See the section Parallel Collector Ergonomics in Sizing the Generations for details on heap resizing and default heap sizes with the parallel collector.) However, the parameters that control the total size of the heap and the sizes of the generations do apply to the parallel collector.

The most important factor affecting garbage collection performance is total available memory. Because collections occur when generations fill up, throughput is inversely proportional to the amount of memory available.

By default, the virtual machine grows or shrinks the heap at each collection to try to keep the proportion of free space to live objects at each collection within a specific range. This target range is set as a percentage by the parameters `-XX:MinHeapFreeRatio=<minimum>` and `-XX:MaxHeapFreeRatio=<maximum>`, and the total size is bounded below by `-Xms<min>` and above by `-Xmx<max>`. The default parameters for the 64-bit Solaris operating system (SPARC Platform Edition) are shown in Table 4-1, "Default Parameters for 64-Bit Solaris Operating System":

Table 4-1 Default Parameters for 64-Bit Solaris Operating System

Parameter	Default Value
`MinHeapFreeRatio`	`40`
`MaxHeapFreeRatio`	`70`
`-Xms`	`6656k`
`-Xmx`	`calculated`

With these parameters, if the percent of free space in a generation falls below 40%, then the generation will be expanded to maintain 40% free space, up to the maximum allowed size of the generation. Similarly, if the free space exceeds 70%, then the generation will be contracted so that only 70% of the space is free, subject to the minimum size of the generation.

As noted in Table 4-1, "Default Parameters for 64-Bit Solaris Operating System", the default maximum heap size is a value that is calculated by the JVM. The calculation used in Java SE for the parallel collector and the server JVM are now used for all the

garbage collectors. Part of the calculation is an upper limit on the maximum heap size that is different for 32-bit platforms and 64-bit platforms. See the section Default Heap Size in The Parallel Collector. There is a similar calculation for the client JVM, which results in smaller maximum heap sizes than for the server JVM.

The following are general guidelines regarding heap sizes for server applications:

- Unless you have problems with pauses, try granting as much memory as possible to the virtual machine. The default size 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. However, the virtual machine is then unable to compensate if you make a poor choice.
- In general, increase the memory as you increase the number of processors, since allocation can be parallelized.

The Young Generation

After total available memory, the second most influential factor affecting garbage collection performance is the proportion of the heap dedicated to the young generation. The bigger the young generation, the less often minor collections occur. However, for a bounded heap size, a larger young generation implies a smaller tenured generation, which will increase the frequency of major collections. The optimal choice depends on the lifetime distribution of the objects allocated by the application.

By default, the young generation size is controlled by the parameter `NewRatio`. For example, setting `-XX:NewRatio=3` means that the ratio between the young and tenured generation is 1:3. In other words, the combined size of the eden and survivor spaces will be one-fourth of the total heap size.

The parameters `NewSize` and `MaxNewSize` bound the young generation size from below and above. Setting these to the same value fixes the young generation, just as setting `-Xms` and `-Xmx` to the same value fixes the total heap size. This is useful for tuning the young generation at a finer granularity than the integral multiples allowed by `NewRatio`.

Survivor Space Sizing

You can use the parameter `SurvivorRatio` can be used to tune the size of the survivor spaces, but this is often not important for performance. For example, `-XX:SurvivorRatio=6` sets the ratio between eden and a survivor space to 1:6. In other words, each survivor space will be one-sixth the size of eden, and thus one-eighth the size of the young generation (not one-seventh, because there are two survivor spaces).

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, which is the number times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full. The command line option `-XX:+PrintTenuringDistribution` (not available on all garbage collectors) 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.

Table 4-2, "Default Parameter Values for Survivor Space Sizing" provides the default values for 64-bit Solaris:

Table 4-2 Default Parameter Values for Survivor Space Sizing

Parameter	Server JVM Default Value
`NewRatio`	`2`
`NewSize`	`1310M`
`MaxNewSize`	`not limited`
`SurvivorRatio`	`8`

The maximum size of the young generation will be calculated from the maximum size of the total heap and the value of the `NewRatio` parameter. The "not limited" default value for the `MaxNewSize` parameter means that the calculated value is not limited by `MaxNewSize` unless a value for `MaxNewSize` is specified on the command line.

The following are general guidelines for server applications:

- First decide the maximum heap size you can afford to give the virtual machine. Then plot your performance metric against young generation sizes to find the best setting.
    - Note that the maximum heap size should always be smaller than the amount of memory installed on the machine to avoid excessive page faults and thrashing.
- If the total heap size is fixed, then increasing the young generation size requires reducing the tenured generation size. Keep the tenured generation large enough to hold all the live data used by the application at any given time, plus some amount of slack space (10 to 20% or more).
- Subject to the previously stated constraint on the tenured generation:
    - Grant plenty of memory to the young generation.
    - Increase the young generation size as you increase the number of processors, because allocation can be parallelized.

5 Available Collectors

The discussion to this point has been about the serial collector. The Java HotSpot VM includes three different types of collectors, each with different performance characteristics.

- The serial collector uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads. It is best-suited to single processor machines, because it cannot take advantage of multiprocessor hardware, although it can be useful on multiprocessors for applications with small data sets (up to approximately 100 MB). The serial collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option `-XX:+UseSerialGC`.

- The parallel collector (also known as the *throughput collector*) performs minor collections in parallel, which can significantly reduce garbage collection overhead. It is intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware. The parallel collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option `-XX:+UseParallelGC`.

    - Parallel compaction is a feature that enables the parallel collector to perform major collections in parallel. Without parallel compaction, major collections are performed using a single thread, which can significantly limit scalability. Parallel compaction is enabled by default if the option `-XX:+UseParallelGC` has been specified. The option to turn it off is `-XX:-UseParallelOldGC`.

- The mostly concurrent collector performs most of its work concurrently (for example, while the application is still running) to keep garbage collection pauses short. It is designed for applications with medium-sized to large-sized data sets in which response time is more important than overall throughput because the techniques used to minimize pauses can reduce application performance. The Java HotSpot VM offers a choice between two mostly concurrent collectors; see The Mostly Concurrent Collectors. Use the option `-XX:+UseConcMarkSweepGC` to enable the CMS collector or `-XX:+UseG1GC` to enable the G1 collector.

Selecting a Collector

Unless your application has rather strict pause time requirements, first run your application and allow the VM to select a collector. If necessary, adjust the heap size to improve performance. If the performance still does not meet your goals, then use the following guidelines as a starting point for selecting a collector.

- If the application has a small data set (up to approximately 100 MB), then

  select the serial collector with the option `-XX:+UseSerialGC`.

- If the application will be run on a single processor and there are no pause time requirements, then let the VM select the collector, or select the serial collector with the option `-XX:+UseSerialGC`.

- If (a) peak application performance is the first priority and (b) there are no pause time requirements or pauses of 1 second or longer are acceptable, then let the VM select the collector, or select the parallel collector with `-XX:+UseParallelGC`.

- If response time is more important than overall throughput and garbage collection pauses must be kept shorter than approximately 1 second, then select the concurrent collector with `-XX:+UseConcMarkSweepGC` or `-XX:+UseG1GC`.

These guidelines provide only a starting point for selecting a collector because performance is dependent on the size of the heap, the amount of live data maintained by the application, and the number and speed of available processors. Pause times are particularly sensitive to these factors, so the threshold of 1 second mentioned previously is only approximate: the parallel collector will experience pause times longer than 1 second on many data size and hardware combinations; conversely, the concurrent collector may not be able to keep pauses shorter than 1 second on some combinations.

If the recommended collector does not achieve the desired performance, first attempt to adjust the heap and generation sizes to meet the desired goals. If performance is still inadequate, then try a different collector: use the concurrent collector to reduce pause times and use the parallel collector to increase overall throughput on multiprocessor hardware.

6 The Parallel Collector

The parallel collector (also referred to here as the *throughput collector*) is a generational collector similar to the serial collector; the primary difference is that multiple threads are used to speed up garbage collection. The parallel collector is enabled with the command-line option `-XX:+UseParallelGC`. By default, with this option, both minor and major collections are executed in parallel to further reduce garbage collection overhead.

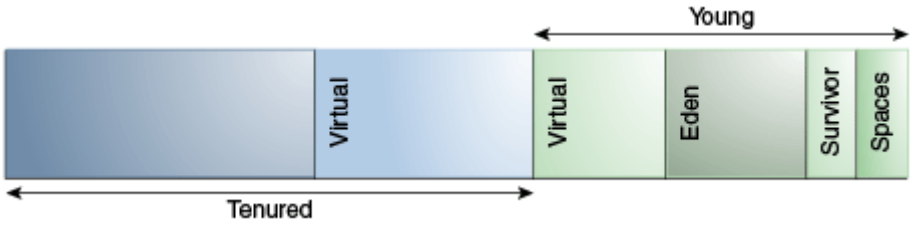
On a machine with *N* hardware threads where *N* is greater than 8, the parallel collector uses a fixed fraction of *N* as the number of garbage collector threads. The fraction is approximately 5/8 for large values of *N*. At values of *N*below 8, the number used is *N*. On selected platforms, the fraction drops to 5/16. The specific number of garbage collector threads can be adjusted with a command-line option (which is described later). On a host with one processor, the parallel collector will likely not perform as well as the serial collector because of the overhead required for parallel execution (for example, synchronization). However, when running applications with medium-sized to large-sized heaps, it generally outperforms the serial collector by a modest amount on machines with two processors, and usually performs significantly better than the serial collector when more than two processors are available.

The number of garbage collector threads can be controlled with the command-line option `-XX:ParallelGCThreads=<N>`. If explicit tuning of the heap is being done with command-line options, then the size of the heap needed for good performance with the parallel collector is the same as needed with the serial collector. However, enabling the parallel collector should make the collection pauses shorter. Because multiple garbage collector threads are participating in a minor collection, some fragmentation is possible due to promotions from the young generation to the tenured generation during the collection. Each garbage collection thread involved in a minor collection reserves a part of the tenured generation for promotions and the division of the available space into these "promotion buffers" can cause a fragmentation effect. Reducing the number of garbage collector threads and increasing the size of the tenured generation will reduce this fragmentation effect.

Generations

As mentioned earlier, the arrangement of the generations is different in the parallel collector. That arrangement is shown in Figure 6-1, "Arrangement of Generations in the Parallel Collector":

Figure 6-1 Arrangement of Generations in the Parallel Collector



Description of "Figure 6-1 Arrangement of Generations in the Parallel Collector"

Parallel Collector Ergonomics

The parallel collector is selected by default on server-class machines. In addition, the parallel collector uses a method of automatic tuning that allows you to specify specific behaviors instead of generation sizes and other low-level tuning details. You can specify maximum garbage collection pause time, throughput, and footprint (heap size).

- **Maximum Garbage Collection Pause Time**: The maximum pause time goal is specified with the command-line option `-XX:MaxGCPauseMillis=<N>`. This is interpreted as a hint that pause times of `<N>` milliseconds or less are desired; by default, there is no maximum pause time goal. If a pause time goal is specified, the heap size and other parameters related to garbage collection are adjusted in an attempt to keep garbage collection pauses shorter than the specified value. These adjustments may cause the garbage collector to reduce the overall throughput of the application, and the desired pause time goal cannot always be met.

- **Throughput**: The throughput goal is measured in terms of the time spent doing garbage collection versus the time spent outside of garbage collection (referred to as application time). The goal is specified by the command-line option `-XX:GCTimeRatio=<N>`, which sets the ratio of garbage collection time to application time to `1 / (1 + <N>)`.

For example, `-XX:GCTimeRatio=19` sets a goal of 1/20 or 5% of the total time in garbage collection. The default value is 99, resulting in a goal of 1% of the time in garbage collection.

- **Footprint**: Maximum heap footprint is specified using the option `-Xmx<N>`. In addition, the collector has an implicit goal of minimizing the size of the heap as long as the other goals are being met.

Priority of Goals

The goals are addressed in the following order:

1. Maximum pause time goal
2. Throughput goal
3. Minimum footprint goal

The maximum pause time goal is met first. Only after it is met is the throughput goal addressed. Similarly, only after the first two goals have been met is the footprint goal considered.

Generation Size Adjustments

The statistics such as average pause time kept by the collector are updated at the end of each collection. The tests to determine if the goals have been met are then made and any needed adjustments to the size of a generation is made. The exception is that explicit garbage collections (for example, calls to `System.gc()`) are ignored in terms of keeping statistics and making adjustments to the sizes of generations.

Growing and shrinking the size of a generation is done by increments that are a fixed percentage of the size of the generation so that a generation steps up or down toward its desired size. Growing and shrinking are done at different rates. By default a generation grows in increments of 20% and shrinks in increments of 5%. The percentage for growing is controlled by the command-line option `-XX:YoungGenerationSizeIncrement=<Y>` for the young generation and `-XX:TenuredGenerationSizeIncrement=<T>` for the tenured generation. The percentage by which a generation shrinks is adjusted by the command-line flag `-XX:AdaptiveSizeDecrementScaleFactor=<D>`. If the growth increment is *X* percent, then the decrement for shrinking is *X*/*D* percent.

If the collector decides to grow a generation at startup, then there is a supplemental percentage is added to the increment. This supplement decays with the number of collections and has no long-term effect. The intent of the supplement is to increase startup performance. There is no supplement to the percentage for shrinking.

If the maximum pause time goal is not being met, then the size of only one generation is shrunk at a time. If the pause times of both generations are above the goal, then the size of the generation with the larger pause time is shrunk first.

If the throughput goal is not being met, the sizes of both generations are increased. Each is increased in proportion to its respective contribution to the total garbage collection time. For example, if the garbage collection time of the young generation is 25% of the total collection time and if a full increment of the young generation would be by 20%, then the young generation would be increased by 5%.

Default Heap Size

Unless the initial and maximum heap sizes are specified on the command line, they are calculated based on the amount of memory on the machine.

Client JVM Default Initial and Maximum Heap Sizes

The default maximum heap size is half of the physical memory up to a physical memory size of 192 megabytes (MB) and otherwise one fourth of the physical memory up to a physical memory size of 1 gigabyte (GB).

For example, if your computer has 128 MB of physical memory, then the maximum heap size is 64 MB, and greater than or equal to 1 GB of physical memory results in a maximum heap size of 256 MB.

The maximum heap size is not actually used by the JVM unless your program creates enough objects to require it. A much smaller amount, called the *initial heap size*, is allocated during JVM initialization. This amount is at least 8 MB and otherwise 1/64th of physical memory up to a physical memory size of 1 GB.

The maximum amount of space allocated to the young generation is one third of the total heap size.

Server JVM Default Initial and Maximum Heap Sizes

The default initial and maximum heap sizes work similarly on the server JVM as it does on the client JVM, except that the default values can go higher. On 32-bit JVMs, the default maximum heap size can be up to 1 GB if there is 4 GB or more of physical memory. On 64-bit JVMs, the default maximum heap size can be up to 32 GB if there is 128 GB or more of physical memory. You can always set a higher or lower initial and maximum heap by specifying those values directly; see the next section.

Specifying Initial and Maximum Heap Sizes

You can specify the initial and maximum heap sizes using the flags `-Xms` (initial heap size) and `-Xmx` (maximum heap size). If you know how much heap your application needs to work well, you can set `-Xms` and `-Xmx` to the same value. If not, the JVM will start by using the initial heap size and will then grow the Java heap until it finds a balance between heap usage and performance.

Other parameters and options can affect these defaults. To verify your default values, use the `-XX:+PrintFlagsFinal` option and look for `MaxHeapSize` in the output. For example, on Linux or Solaris, you can run the following:

```
java -XX:+PrintFlagsFinal <GC options> -version | grep MaxHeapSize
```

Excessive GC Time and OutOfMemoryError

The parallel collector throws an `OutOfMemoryError` if too much time is being spent in garbage collection (GC): If more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an`OutOfMemoryError` is thrown. This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. If necessary, this feature can be disabled by adding the option `-XX:-UseGCOverheadLimit` to the command line.

Measurements

The verbose garbage collector output from the parallel collector is essentially the same as that from the serial collector.

7 The Mostly Concurrent Collectors

Java Hotspot VM has two mostly concurrent collectors in JDK 8:

- Concurrent Mark Sweep (CMS) Collector: This collector is for applications that prefer shorter garbage collection pauses and can afford to share processor resources with the garbage collection.

- Garbage-First Garbage Collector: This server-style collector is for multiprocessor machines with large memories. It meets garbage collection pause time goals with high probability while achieving high throughput.

Overhead of Concurrency

The mostly concurrent collector trades processor resources (which would otherwise be available to the application) for shorter major collection pause times. The most visible overhead is the use of one or more processors during the concurrent parts of the collection. On an *N* processor system, the concurrent part of the collection will use *K*/*N* of the available processors, where 1<=*K*<=ceiling{*N*/4}. (Note that the precise choice of and bounds on *K* are subject to change.) In addition to the use of processors during concurrent phases, additional overhead is incurred to enable concurrency. Thus while garbage collection pauses are typically much shorter with the concurrent collector, application throughput also tends to be slightly lower than with the other collectors.

On a machine with more than one processing core, processors are available for application threads during the concurrent part of the collection, so the concurrent garbage collector thread does not "pause" the application. This usually results in shorter pauses, but again fewer processor resources are available to the application and some slowdown should be expected, especially if the application uses all of the processing cores maximally. As *N* increases, the reduction in processor resources due to concurrent garbage collection becomes smaller, and the benefit from concurrent collection increases. The section Concurrent Mode Failure in Concurrent Mark Sweep (CMS) Collector discusses potential limits to such scaling.

Because at least one processor is used for garbage collection during the concurrent phases, the concurrent collectors do not normally provide any benefit on a uniprocessor (single-core) machine. However, there is a separate mode available for CMS (not G1) that can achieve low pauses on systems with only one or two processors; see Incremental Mode in Concurrent Mark Sweep (CMS) Collector for details. This feature is being deprecated in Java SE 8 and may be removed in a later major release.

Additional References

The Garbage-First Garbage Collector:

`http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html`

Garbage-First Garbage Collector Tuning:

`http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html`

8 Concurrent Mark Sweep (CMS) Collector

The Concurrent Mark Sweep (CMS) collector is designed for applications that prefer shorter garbage collection pauses and that can afford to share processor resources with the garbage collector while the application is running. Typically applications that have a relatively large set of long-lived data (a large tenured generation) and run on machines with two or more processors tend to benefit from the use of this collector. However, this collector should be considered for any application with a low pause time requirement. The CMS collector is enabled with the command-line option `-XX:+UseConcMarkSweepGC`.

Similar to the other available collectors, the CMS collector is generational; thus both minor and major collections occur. The CMS collector attempts to reduce pause times due to major collections by using separate garbage collector threads to trace the reachable objects concurrently with the execution of the application threads. During each major collection cycle, the CMS collector pauses all the application threads for a brief period at the beginning of the collection and again toward the middle of the collection. The second pause tends to be the longer of the two pauses. Multiple threads are used to do the collection work during both pauses. The remainder of the collection (including most of the tracing of live objects and sweeping of unreachable objects is done with one or more garbage collector threads that run concurrently with the application. Minor collections can interleave with an ongoing major cycle, and are done in a manner similar to the parallel collector (in particular, the application threads are stopped during minor collections).

Concurrent Mode Failure

The CMS collector uses one or more garbage collector threads that run simultaneously with the application threads with the goal of completing the collection of the tenured generation before it becomes full. As described previously, in normal operation, the CMS collector does most of its tracing and sweeping work with the application threads still running, so only brief pauses are seen by the application threads. However, if the CMS collector is unable to finish reclaiming the unreachable objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped. The inability to complete a collection concurrently is referred to as *concurrent mode failure* and indicates the need to adjust the CMS collector parameters. If a concurrent collection is interrupted by an explicit garbage collection (`System.gc()`) or for a garbage collection needed to provide information for diagnostic tools, then a concurrent mode interruption is reported.

Excessive GC Time and OutOfMemoryError

The CMS collector throws an `OutOfMemoryError` if too much time is being spent in garbage collection: if more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an`OutOfMemoryError` is thrown. This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. If necessary, this feature can be disabled by adding the option `-XX:-UseGCOverheadLimit` to the command line.

The policy is the same as that in the parallel collector, except that time spent performing concurrent collections is not counted toward the 98% time limit. In other words, only collections performed while the application is stopped count toward excessive GC time. Such collections are typically due to a concurrent mode failure or an explicit collection request (for example, a call to `System.gc`).

Floating Garbage

The CMS collector, like all the other collectors in Java HotSpot VM, is a tracing collector that identifies at least all the reachable objects in the heap. In the parlance of Richard Jones and Rafael D. Lins in their publication *Garbage Collection: Algorithms for Automated Dynamic Memory*, it is an incremental update collector. Because application threads and the garbage collector thread run concurrently during a major collection, objects that are traced by the garbage collector thread may subsequently become unreachable by the time collection process ends. Such unreachable objects that have not yet been reclaimed are referred to as floating garbage. The amount of *floating garbage*depends on the duration of the concurrent collection cycle and on the frequency of reference updates, also known as *mutations*, by the application. Furthermore, because the young generation and the tenured generation are collected independently, each acts a source of roots to the other. As a rough guideline, try increasing the size of the tenured generation by 20% to account for the floating garbage. Floating garbage in the heap at the end of one concurrent collection cycle is collected during the next collection cycle.

Pauses

The CMS collector pauses an application twice during a concurrent collection cycle. The first pause is to mark as live the objects directly reachable from the roots (for example, object references from application thread stacks and registers, static objects and so on) and from elsewhere in the heap (for example, the young generation). This first pause is referred to as the *initial mark pause*. The second pause comes at the end of the concurrent tracing phase and finds objects that were missed by the concurrent tracing due to updates by the application threads of references in an object after the CMS collector had finished tracing that object. This second pause is referred to as the *remark pause*.

Concurrent Phases

The concurrent tracing of the reachable object graph occurs between the initial mark pause and the remark pause. During this concurrent tracing phase one or more concurrent garbage collector threads may be using processor resources that would otherwise have been available to the application. As a result, compute-bound applications may see a commensurate fall in application throughput during this and other concurrent phases even though the application threads are not paused. After the remark pause, a concurrent sweeping phase collects the objects identified as unreachable. Once a collection cycle completes, the CMS collector waits, consuming almost no computational resources, until the start of the next major collection cycle.

Starting a Concurrent Collection Cycle

With the serial collector a major collection occurs whenever the tenured generation becomes full and all application threads are stopped while the collection is done. In contrast, the start of a concurrent collection must be timed such that the collection can finish before the tenured generation becomes full; otherwise, the application would observe longer pauses due to concurrent mode failure. There are several ways to start a concurrent collection.

Based on recent history, the CMS collector maintains estimates of the time remaining before the tenured generation will be exhausted and of the time needed for a concurrent collection cycle. Using these dynamic estimates, a concurrent collection cycle is started with the aim of completing the collection cycle before the tenured generation is exhausted. These estimates are padded for safety, because concurrent mode failure can be very costly.

A concurrent collection also starts if the occupancy of the tenured generation exceeds an initiating occupancy (a percentage of the tenured generation). The default value for this initiating occupancy threshold is approximately 92%, but the value is subject to change from release to release. This value can be manually adjusted using the command-line option `-XX:CMSInitiatingOccupancyFraction=<N>`, where `<N>` is an integral percentage (0 to 100) of the tenured generation size.

Scheduling Pauses

The pauses for the young generation collection and the tenured generation collection occur independently. They do not overlap, but may occur in quick succession such that the pause from one collection, immediately followed by one from the other collection, can appear to be a single, longer pause. To avoid this, the CMS collector attempts to schedule the remark pause roughly midway between the previous and next young generation pauses. This scheduling is currently not done for the initial mark pause, which is usually much shorter than the remark pause.

Incremental Mode

Note that the incremental mode is being deprecated in Java SE 8 and may be removed in a future major release.

The CMS collector can be used in a mode in which the concurrent phases are done incrementally. Recall that during a concurrent phase the garbage collector thread is using one or more processors. The incremental mode is meant to lessen the effect of long concurrent phases by periodically stopping the concurrent phase to yield back the processor to the application. This mode, referred to here as *i-cms*, divides the work done concurrently by the collector into small chunks of time that are scheduled between young generation collections. This feature is useful when applications that need the low pause times provided by the CMS collector are run on machines with small numbers of processors (for example, 1 or 2).

The concurrent collection cycle typically includes the following steps:

- Stop all application threads, identify the set of objects reachable from roots, and then resume all application threads.
- Concurrently trace the reachable object graph, using one or more processors, while the application threads are executing.
- Concurrently retrace sections of the object graph that were modified since the tracing in the previous step, using one processor.
- Stop all application threads and retrace sections of the roots and object graph that may have been modified since they were last examined, and then resume all application threads.
- Concurrently sweep up the unreachable objects to the free lists used for allocation, using one processor.
- Concurrently resize the heap and prepare the support data structures for the next collection cycle, using one processor.

Normally, the CMS collector uses one or more processors during the entire concurrent tracing phase, without voluntarily relinquishing them. Similarly, one processor is used for the entire concurrent sweep phase, again without relinquishing it. This overhead can be too much of a disruption for applications with response time constraints that might otherwise have used the processing cores, particularly when run on systems with just one or two processors. Incremental mode solves this problem by breaking up the concurrent phases into short bursts of activity, which are scheduled to occur midway between minor pauses.

The i-cms mode uses a duty cycle to control the amount of work the CMS collector is allowed to do before voluntarily giving up the processor. The *duty cycle* is the percentage of time between young generation collections that the CMS collector is allowed to run. The i-cms mode can automatically compute the duty cycle based on the behavior of the application (the recommended method, known as *automatic pacing*), or the duty cycle can be set to a fixed value on the command line.

Command-Line Options

Table 8-1, "Command-Line Options for i-cms" list command-line options that control the i-cms mode. The section Recommended Options suggests an initial set of options.

Table 8-1 Command-Line Options for i-cms

Option	Description	Default Value, Java SE 5 and Earlier	Default Value, Java SE 6 and Later
`-XX:+CMSIncrementalMode`	Enables incremental mode. Note that the CMS collector must also be enabled (with `-XX:+UseConcMarkSweepGC`) for this option to work.	disabled	disabled
`-XX:+CMSIncrementalPacing`	Enables automatic pacing. The incremental mode duty cycle is automatically adjusted based on statistics collected while the JVM is running.	disabled	disabled
`-XX:CMSIncrementalDutyCycle=<N>`	The percentage (0 to 100) of time between minor collections that the CMS collector is allowed to run. If `CMSIncrementalPacing` is enabled, then this is just the initial value.	50	10
`-XX:CMSIncrementalDutyCycleMin=<N>`	The percentage (0 to 100) that is the lower bound on the duty cycle when `CMSIncrementalPacing` is enabled.	10	0

Option	Description	Default Value, Java SE 5 and Earlier	Default Value, Java SE 6 and Later
`-XX:CMSIncrementalSafetyFactor=<N>`	The percentage (0 to 100) used to add conservatism when computing the duty cycle	10	10
`-XX:CMSIncrementalOffset=<N>`	The percentage (0 to 100) by which the incremental mode duty cycle is shifted to the right within the period between minor collections.	0	0
`-XX:CMSExpAvgFactor=<N>`	The percentage (0 to 100) used to weight the current sample when computing exponential averages for the CMS collection statistics.	25	25

Recommended Options

To use i-cms in Java SE 8, use the following command-line options:

```
-XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode \
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps
```

The first two options enable the CMS collector and i-cms, respectively. The last two options are not required; they simply cause diagnostic information about garbage collection to be written to standard output, so that garbage collection behavior can be seen and later analyzed.

For Java SE 5 and earlier releases, Oracle recommends using the following as an initial set of command-line options for i-cms:

```
-XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode \
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps \
-XX:+CMSIncrementalPacing -XX:CMSIncrementalDutyCycleMin=0
-XX:CMSIncrementalDutyCycle=10
```

The same values are recommended for JavaSE8 although the values for the three options that control i-cms automatic pacing became the default in JavaSE6.

Basic Troubleshooting

The i-cms automatic pacing feature uses statistics gathered while the program is running to compute a duty cycle so that concurrent collections complete before the heap becomes full. However, past behavior is not a perfect predictor of future behavior and the estimates may not always be accurate enough to prevent the heap from becoming full. If too many full collections occur, then try the steps in Table 8-2, "Troubleshooting the i-cms Automatic Pacing Feature", one at a time.

Table 8-2 Troubleshooting the i-cms Automatic Pacing Feature

Step	Options
1. Increase the safety factor.	`-XX:CMSIncrementalSafetyFactor=<N>`
2. Increase the minimum duty cycle.	`-XX:CMSIncrementalDutyCycleMin=<N>`
3. Disable automatic pacing and use a fixed duty cycle.	`-XX:-CMSIncrementalPacing -XX:CMSIncrementalDutyCycle=<N>`

Measurements

Example 8-1, "Output from the CMS Collector" is the output from the CMS collector with the options `-verbose:gc` and `-XX:+PrintGCDetails`, with a few minor details removed. Note that the output for the CMS collector is interspersed with the

output from the minor collections; typically many minor collections occur during a concurrent collection cycle. CMS-initial-mark indicates the start of the concurrent collection cycle, CMS-concurrent-mark indicates the end of the concurrent marking phase, and CMS-concurrent-sweep marks the end of the concurrent sweeping phase. Not discussed previously is the precleaning phase indicated by CMS-concurrent-preclean. Precleaning represents work that can be done concurrently in preparation for the remark phase CMS-remark. The final phase is indicated by CMS-concurrent-reset and is in preparation for the next concurrent collection.

Example 8-1 Output from the CMS Collector

```
[GC [1 CMS-initial-mark: 13991K(20288K)] 14103K(22400K), 0.0023781 secs]
[GC [DefNew: 2112K->64K(2112K), 0.0837052 secs] 16103K->15476K(22400K), 0.0838519
secs]
...
[GC [DefNew: 2077K->63K(2112K), 0.0126205 secs] 17552K->15855K(22400K), 0.0127482
secs]
[CMS-concurrent-mark: 0.267/0.374 secs]
[GC [DefNew: 2111K->64K(2112K), 0.0190851 secs] 17903K->16154K(22400K), 0.0191903
secs]
[CMS-concurrent-preclean: 0.044/0.064 secs]
[GC [1 CMS-remark: 16090K(20288K)] 17242K(22400K), 0.0210460 secs]
[GC [DefNew: 2112K->63K(2112K), 0.0716116 secs] 18177K->17382K(22400K), 0.0718204
secs]
[GC [DefNew: 2111K->63K(2112K), 0.0830392 secs] 19363K->18757K(22400K), 0.0832943
secs]
...
[GC [DefNew: 2111K->0K(2112K), 0.0035190 secs] 17527K->15479K(22400K), 0.0036052
secs]
[CMS-concurrent-sweep: 0.291/0.662 secs]
[GC [DefNew: 2048K->0K(2112K), 0.0013347 secs] 17527K->15479K(27912K), 0.0014231
secs]
[CMS-concurrent-reset: 0.016/0.016 secs]
[GC [DefNew: 2048K->1K(2112K), 0.0013936 secs] 17527K->15479K(27912K), 0.0014814
secs
]
```

The initial mark pause is typically short relative to the minor collection pause time. The concurrent phases (concurrent mark, concurrent preclean and concurrent sweep) normally last significantly longer than a minor collection pause, as indicated by Example 8-1, "Output from the CMS Collector". Note, however, that the application is not paused during these concurrent phases. The remark pause is often comparable in length to a minor collection. The remark pause is affected by certain application characteristics (for example, a high rate of object modification can increase this pause) and the time since the last minor collection (for example, more objects in the young generation may increase this pause).

9 Garbage-First Garbage Collector

The Garbage-First (G1) garbage collector is a server-style garbage collector, targeted for multiprocessor machines with large memories. It attempts to meet garbage collection (GC) pause time goals with high probability while achieving high throughput. Whole-heap operations, such as global marking, are performed concurrently with the application threads. This prevents interruptions proportional to heap or live-data size.

The G1 collector achieves high performance and pause time goals through several techniques.

The heap is partitioned into a set of equally sized heap regions, each a contiguous range of virtual memory. G1 performs a concurrent global marking phase to determine the liveness of objects throughout the heap. After the marking phase completes, G1 knows which regions are mostly empty. It collects these regions first, which often yields a large amount of free space. This is why this method of garbage collection is called Garbage-First. As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage. G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.

G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory. This evacuation is performed in parallel on multiprocessors to decrease pause times and increase throughput. Thus, with each garbage collection, G1 continuously works to reduce fragmentation. This is beyond the capability of both of the previous methods. CMS (Concurrent Mark Sweep) garbage collection does not do compaction. Parallel compaction performs only whole-heap compaction, which results in considerable pause times.

It is important to note that G1 is not a real-time collector. It meets the set pause time target with high probability but not absolute certainty. Based on data from previous collections, G1 estimates how many regions can be collected within the target time. Thus, the collector has a reasonably accurate model of the cost of collecting the regions, and it uses this model to determine which and how many regions to collect while staying within the pause time target.

The first focus of G1 is to provide a solution for users running applications that require large heaps with limited GC latency. This means heap sizes of around 6 GB or larger, and a stable and predictable pause time below 0.5 seconds.

Applications running today with either the CMS or the with parallel compaction would benefit from switching to G1 if the application has one or more of the following traits.

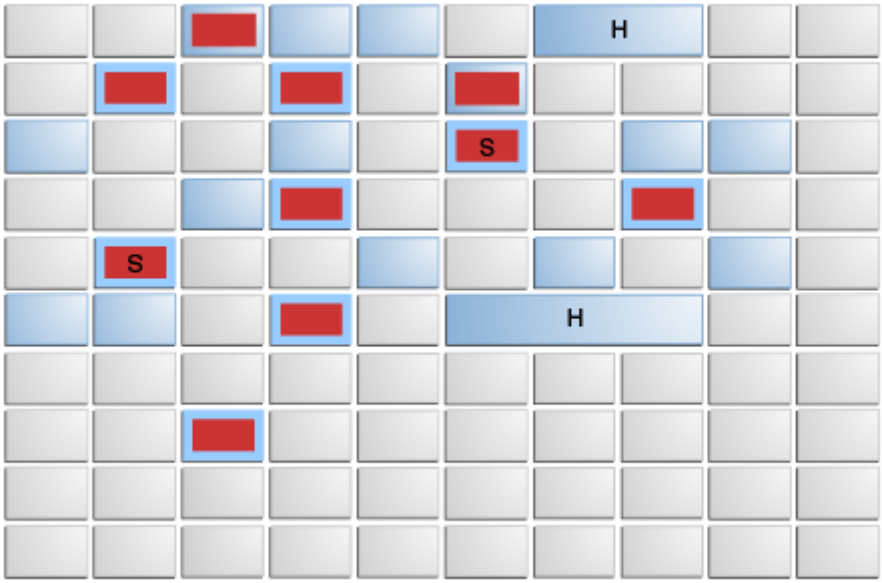
- More than 50% of the Java heap is occupied with live data.
- The rate of object allocation rate or promotion varies significantly.
- The application is experiencing undesired long garbage collection or compaction pauses (longer than 0.5 to 1 second).

G1 is planned as the long-term replacement for the Concurrent Mark-Sweep Collector (CMS). Comparing G1 with CMS reveals differences that make G1 a better solution. One difference is that G1 is a compacting collector. Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.

As with CMS, G1 is designed for applications that require shorter GC pauses.

G1 divides the heap into fixed-sized regions (the gray boxes) as in Figure 9-1, "Heap Division by G1".

Figure 9-1 Heap Division by G1



Description of "Figure 9-1 Heap Division by G1"

G1 is generational in a logical sense. A set of empty regions is designated as the logical young generation. In the figure, the young generation is light blue. Allocations are done out of that logical young generation, and when the young generation is full, that set of regions is garbage collected (a young collection). In some cases, regions outside the set of young regions (old regions in dark blue) can be garbage collected at the same time. This is referred to as a *mixed collection*. In the figure, the regions being collected are marked by red boxes. The figure illustrates a mixed collection because both young regions and old regions are being collected. The garbage collection is a compacting collection that copies live objects to selected, initially empty regions. Based on the age of a surviving object, the object can be copied to a survivor region (marked by "S") or to an old region (not specifically shown). The regions marked by "H" contain humongous objects that are larger than half a region and are treated specially; see the section Humongous Objects and Humongous Allocations in Garbage-First Garbage Collector.

Allocation (Evacuation) Failure

As with CMS, the G1 collector runs parts of its collection while the application continues to run and there is a risk that the application will allocate objects faster than the garbage collector can recover free space. See the sectionConcurrent Mode Failure in Concurrent Mark Sweep (CMS) Collector for the analogous CMS behavior. In G1, the failure (exhaustion of the Java heap) occurs while G1 is copying live data out of one region (evacuating) into another region. The copying is done to compact the live data. If a free (empty) region cannot be found during the evacuation of a region being garbage collected, then an allocation failure occurs (because there is no space to allocate the live objects from the region being evacuated) and a stop-the-world (STW) full collection is done.

Floating Garbage

Objects can die during a G1 collection and not be collected. G1 uses a technique called snapshot-at-the-beginning (SATB) to guarantee that all live objects are found by the garbage collector. SATB states that any object that is live at the start of the concurrent marking (a marking over the entire heap) is considered live for the purpose of the collection. SATB allows floating garbage in a way analogous to that of a CMS incremental update.

Pauses

G1 pauses the application to copy live objects to new regions. These pauses can either be young collection pauses where only young regions are collected or mixed collection pauses where young and old regions are evacuated. As with CMS there is a final marking or remark pause to complete the marking while the application is stopped. Whereas CMS also had an initial marking pause, G1 does the initial marking work as part of an evacuation pause. G1 has a cleanup phase at the end of a collection which is partly STW and partly concurrent. The STW part of the cleanup phase identifies empty regions and determines old regions that are candidates for the next collection.

Card Tables and Concurrent Phases

If a garbage collector does not collect the entire heap (an *incremental collection*), the garbage collector needs to know where there are pointers from the uncollected part of the heap into the part of the heap that is being collected. This is typically for a generational garbage collector in which the uncollected part of the heap is usually the old generation, and the collected part of the heap is the young generation. The data structure for keeping this information (old generation pointers to young generation objects), is a *remembered set*. A *card table* is a particular type of remembered set. Java HotSpot VM uses an array of bytes as a card table. Each byte is referred to as a *card*. A card corresponds to a range of addresses in the heap. *Dirtying a card* means changing the value of the byte to a *dirty value*; a dirty value might contain a new pointer from the old generation to the young generation in the address range covered by the card.

Processing a card means looking at the card to see if there is an old generation to young generation pointer and perhaps doing something with that information such as transferring it to another data structure.

G1 has concurrent marking phase which marks live objects found from the application. The concurrent marking extends from the end of a evacuation pause (where the initial marking work is done) to the remark. The concurrent cleanup phase adds regions emptied by the collection to the list of free regions and clears the remembered sets of those regions. In addition, a concurrent refinement thread runs as needed to process card table entries that have been dirtied by application writes and which may have cross region references.

Starting a Concurrent Collection Cycle

As mentioned previously, both young and old regions are garbage collected in a mixed collection. To collect old regions, G1 does a complete marking of the live objects in the heap. Such a marking is done by a concurrent marking phase. A concurrent marking phase is started when the occupancy of the entire Java heap reaches the value of the parameter `InitiatingHeapOccupancyPercent`. Set the value of this parameter with the command-line option `-XX:InitiatingHeapOccupancyPercent=<NN>`. The default value of `InitiatingHeapOccupancyPercent` is 45.

Pause Time Goal

Set a pause time goal for G1 with the flag `MaxGCPauseMillis`. G1 uses a prediction model to decide how much garbage collection work can be done within that target pause time. At the end of a collection, G1 chooses the regions to be collected in the next collection (the collection set). The collection set will contain young regions (the sum of whose sizes determines the size of the logical young generation). It is partly through the selection of the number of young regions in the collection set that G1 exerts control over the length of the GC pauses. You can specify the size of the young generation on the command line as with the other garbage collectors, but doing so may hamper the ability of G1 to attain the target pause time. In addition to the pause time goal, you can specify the length of the time period during which the pause can occur. You can specify the minimum mutator usage with this time span (`GCPauseIntervalMillis`) along with the pause time goal. The default value for `MaxGCPauseMillis` is 200 milliseconds. The default value for `GCPauseIntervalMillis` (0) is the equivalent of no requirement on the time span.

10 Garbage-First Garbage Collector Tuning

This section describes how to adapt and tune the Garbage-First garbage collector (G1 GC) for evaluation, analysis and performance.

As described in the section Garbage-First Garbage Collector, the G1 GC is a regionalized and generational garbage collector, which means that the Java object heap (heap) is divided into a number of equally sized regions. Upon startup, the Java Virtual Machine (JVM) sets the region size. The region sizes can vary from 1 MB to 32 MB depending on the heap size. The goal is to have no more than 2048 regions. The eden, survivor, and old generations are logical sets of these regions and are not contiguous.

The G1 GC has a pause time target that it tries to meet (soft real time). During young collections, the G1 GC adjusts its young generation (eden and survivor sizes) to meet the soft real-time target. See the sections Pauses and Pause Time Goal in Garbage-First Garbage Collector for information about why the G1 GC takes pauses and how to set pause time targets.

During mixed collections, the G1 GC adjusts the number of old regions that are collected based on a target number of mixed garbage collections, the percentage of live objects in each region of the heap, and the overall acceptable heap waste percentage.

The G1 GC reduces heap fragmentation by incremental parallel copying of live objects from one or more sets of regions (called Collection Sets (CSet)s) into one or more different new regions to achieve compaction. The goal is to reclaim as much heap space as possible, starting with those regions that contain the most reclaimable space, while attempting to not exceed the pause time goal (garbage first).

The G1 GC uses independent Remembered Sets (RSets) to track references into regions. Independent RSets enable parallel and independent collection of regions because only a region's RSet must be scanned for references into that region, instead of the whole heap. The G1 GC uses a post-write barrier to record changes to the heap and update the RSets.

Garbage Collection Phases

Apart from evacuation pauses (see the section Allocation (Evacuation) Failure in Garbage-First Garbage Collector) that compose the stop-the-world (STW) young and mixed garbage collections, the G1 GC also has parallel, concurrent, and multiphase marking cycles. G1 GC uses the snapshot-at-the-beginning (SATB) algorithm, which logically takes a snapshot of the set of live objects in the heap at the start of a marking cycle. The set of live objects also includes objects allocated since the start of the marking cycle. The G1 GC marking algorithm uses a pre-write barrier to record and mark objects that are part of the logical snapshot.

Young Garbage Collections

The G1 GC satisfies most allocation requests from regions added to the eden set of regions. During a young garbage collection, the G1 GC collects both the eden regions and the survivor regions from the previous garbage collection. The live objects from the eden and survivor regions are copied, or evacuated, to a new set of regions. The destination region for a particular object depends upon the object's age; an object that has aged sufficiently evacuates to an old generation region (that is, it is promoted); otherwise, the object evacuates to a survivor region and will be included in the CSet of the next young or mixed garbage collection.

Mixed Garbage Collections

Upon successful completion of a concurrent marking cycle, the G1 GC switches from performing young garbage collections to performing mixed garbage collections. In a mixed garbage collection, the G1 GC optionally adds some old regions to the set of eden and survivor regions that will be collected. The exact number of old regions added is controlled by a number of flags (see "Taming Mixed Garbage Collectors" in the section Recommendations). After the G1 GC collects a sufficient number of old regions (over multiple mixed garbage collections), G1 reverts to performing young garbage collections until the next marking cycle completes.

Phases of the Marking Cycle

The marking cycle has the following phases:

- **Initial marking phase**: The G1 GC marks the roots during this phase. This phase is piggybacked on a normal (STW) young garbage collection.
- **Root region scanning phase**: The G1 GC scans survivor regions marked during the initial marking phase for references to the old generation and marks the referenced objects. This phase runs concurrently with the application (not STW) and must complete before the next STW young garbage collection can start.
- **Concurrent marking phase**: The G1 GC finds reachable (live) objects across the entire heap. This phase happens concurrently with the application, and can be interrupted by STW young garbage collections.
- **Remark phase**: This phase is STW collection and helps the completion of the marking cycle. G1 GC drains SATB buffers, traces unvisited live objects, and performs reference processing.
- **Cleanup phase**: In this final phase, the G1 GC performs the STW operations of accounting and RSet scrubbing. During accounting, the G1 GC identifies completely free regions and mixed garbage collection candidates. The cleanup phase is partly concurrent when it resets and returns the empty regions to the free list.

Important Defaults

The G1 GC is an adaptive garbage collector with defaults that enable it to work efficiently without modification. Table 10-1, "Default Values of Important Options for G1 Garbage Collector" lists of important options and their default values in Java HotSpot VM, build 24. You can adapt and tune the G1 GC to your application performance needs by entering the options in Table 10-1, "Default Values of Important Options for G1 Garbage Collector" with changed settings on the JVM command line.

Table 10-1 Default Values of Important Options for G1 Garbage Collector

Option and Default Value	Option
`-XX:G1HeapRegionSize=n`	Sets the size of a G1 region. The value will be a power of two and can range from 1 MB to 32 MB. The goal is to have around 2048 regions based on the minimum Java heap size.
`-XX:MaxGCPauseMillis=200`	Sets a target value for desired maximum pause time. The default value is 200 milliseconds. The specified value does not adapt to your heap size.
`-XX:G1NewSizePercent=5`	Sets the percentage of the heap to use as the minimum for the young generation size. The default value is 5 percent of your Java heap.[Foot1] This is an experimental flag. See How to Unlock Experimental VM Flags for an example. This setting replaces the `-XX:DefaultMinNewGenPercent` setting.
`-XX:G1MaxNewSizePercent=60`	Sets the percentage of the heap size to use as the maximum for young generation size. The default value is 60 percent of your Java heap.[Footref1] This is an experimental flag. See How to Unlock Experimental VM Flags for an example. This setting replaces the `-XX:DefaultMaxNewGenPercent` setting.
`-XX:ParallelGCThreads=n`	Sets the value of the STW worker threads. Sets the value of `n` to the number of logical processors. The value of `n` is the same as the number of logical processors up to a value of 8. If there are more than eight logical processors, sets the value of *n* to approximately 5/8 of the logical processors. This works in most cases except for larger SPARC systems where the value of *n* can be approximately 5/16 of the logical processors.

Option and Default Value	Option
`-XX:ConcGCThreads=n`	Sets the number of parallel marking threads. Sets `n` to approximately 1/4 of the number of parallel garbage collection threads (ParallelGCThreads).
`-XX:InitiatingHeapOccupancyPercent=45`	Sets the Java heap occupancy threshold that triggers a marking cycle. The default occupancy is 45 percent of the entire Java heap.
`-XX:G1MixedGCLiveThresholdPercent=85`	Sets the occupancy threshold for an old region to be included in a mixed garbage collection cycle. The default occupancy is 85 percent.[Footref1] This is an experimental flag. See How to Unlock Experimental VM Flags for an example. This setting replaces the `-XX:G1OldCSetRegionLiveThresholdPercent` setting.
`-XX:G1HeapWastePercent=5`	Sets the percentage of heap that you are willing to waste. The Java HotSpot VM does not initiate the mixed garbage collection cycle when the reclaimable percentage is less than the heap waste percentage. The default is 5 percent.[Footref1]
`-XX:G1MixedGCCountTarget=8`	Sets the target number of mixed garbage collections after a marking cycle to collect old regions with at most `G1MixedGCLIveThresholdPercent` live data. The default is 8 mixed garbage collections. The goal for mixed collections is to be within this target number.[Footref1]
`-XX:G1OldCSetRegionThresholdPercent=10`	Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle. The default is 10 percent of the Java heap.[Footref1]
`-XX:G1ReservePercent=10`	Sets the percentage of reserve memory to keep free so as to reduce the risk of to-space overflows. The default is 10 percent. When you increase or decrease the percentage, make sure to adjust the total Java heap by the same amount.[Footref1]

[Footnote1] This setting is not available in Java HotSpot VM build 23 or earlier.

How to Unlock Experimental VM Flags

To change the value of experimental flags, you must unlock them first. You can do this by setting `-XX:+UnlockExperimentalVMOptions` explicitly on the command line before any experimental flags. For example:

```
java -XX:+UnlockExperimentalVMOptions -XX:G1NewSizePercent=10 -
XX:G1MaxNewSizePercent=75 G1test.jar
```

Recommendations

When you evaluate and fine-tune G1 GC, keep the following recommendations in mind:

- **Young Generation Size**: Avoid explicitly setting young generation size with the `-Xmn` option or any or other related option such as `-XX:NewRatio`. Fixing the size of the young generation overrides the target pause-time goal.

- **Pause Time Goals**: When you evaluate or tune any garbage collection, there is always a latency versus throughput trade-off. The G1 GC is an incremental garbage collector with uniform pauses, but also more overhead on the application threads. The throughput goal for the G1 GC is 90 percent application time and 10 percent garbage collection time. Compare this to the Java HotSpot VM parallel collector. The throughput goal of the parallel collector is 99 percent application time and 1 percent garbage collection time. Therefore, when you evaluate the G1 GC for throughput, relax your pause time target. Setting too aggressive a goal indicates that you are willing to bear an increase in garbage collection overhead, which has a direct effect on throughput. When you evaluate the G1 GC for latency, you set your desired (soft) real-time goal, and the G1 GC will try to meet it. As a side effect, throughput may suffer. See the section Pause Time Goal in Garbage-First Garbage Collector for additional information.

- **Taming Mixed Garbage Collections**: Experiment with the following options when you tune mixed garbage collections. See the section Important Defaults for information about these options:

  - `-XX:InitiatingHeapOccupancyPercent`: Use to change the marking threshold.

  - `-XX:G1MixedGCLiveThresholdPercent` and `-XX:G1HeapWastePercent`: Use to change the mixed garbage collection decisions.

  - `-XX:G1MixedGCCountTarget` and `-XX:G1OldCSetRegionThresholdPercent`: Use to adjust the CSet for old regions.

Overflow and Exhausted Log Messages

When you see to-space overflow or to-space exhausted messages in your logs, the G1 GC does not have enough memory for either survivor or promoted objects, or for both. The Java heap cannot because it is already at its maximum. Example messages:

- `924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space exhausted), 0.1957310 secs]`

- `924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space overflow), 0.1957310 secs]`

To alleviate the problem, try the following adjustments:

- Increase the value of the `-XX:G1ReservePercent` option (and the total heap accordingly) to increase the amount of reserve memory for "to-space".

- Start the marking cycle earlier by reducing the value of `-XX:InitiatingHeapOccupancyPercent.`

- Increase the value of the `-XX:ConcGCThreads` option to increase the number of parallel marking threads.

See the section Important Defaults for a description of these options.

Humongous Objects and Humongous Allocations

For G1 GC, any object that is more than half a region size is considered a *humongous object*. Such an object is allocated directly in the old generation into *humongous regions*. These humongous regions are a contiguous set of regions. `StartsHumongous` marks the start of the contiguous set and `ContinuesHumongous` marks the continuation of the set.

Before allocating any humongous region, the marking threshold is checked, initiating a concurrent cycle, if necessary.

Dead humongous objects are freed at the end of the marking cycle during the cleanup phase and also during a full garbage collection cycle.

To reduce copying overhead, the humongous objects are not included in any evacuation pause. A full garbage collection cycle compacts humongous objects in place.

Because each individual set of `StartsHumongous` and `ContinuesHumongous` regions contains just one humongous object, the space between the end of the humongous object and the end of the last region spanned by the object is unused. For objects that are just slightly larger than a multiple of the heap region size, this unused space can cause the heap to become fragmented.

If you see back-to-back concurrent cycles initiated due to humongous allocations and if such allocations are fragmenting your old generation, then increase the value of `-XX:G1HeapRegionSize` such that previous humongous objects are no longer humongous and will follow the regular allocation path.

11 Other Considerations

This section covers other situations that affect garbage collection.

Finalization and Weak, Soft, and Phantom References

Some applications interact with garbage collection by using finalization and weak, soft, or phantom references. These features can create performance artifacts at the Java programming language level. An example of this is relying on finalization to close file descriptors, which makes an external resource (descriptors) dependent on garbage collection promptness. Relying on garbage collection to manage resources other than memory is almost always a bad idea.

The section Related Documents in the Preface includes an article that discusses in depth some of the pitfalls of finalization and techniques for avoiding them.

Explicit Garbage Collection

Another way that applications can interact with garbage collection is by invoking full garbage collections explicitly by calling `System.gc()`. This can force a major collection to be done when it may not be necessary (for example, when a minor collection would suffice), and so in general should be avoided. The performance effect of explicit garbage collections can be measured by disabling them using the flag `-XX:+DisableExplicitGC`, which causes the VM to ignore calls to `System.gc()`.

One of the most commonly encountered uses of explicit garbage collection occurs with the distributed garbage collection (DGC) of Remote Method Invocation (RMI). Applications using RMI refer to objects in other virtual machines. Garbage cannot be collected in these distributed applications without occasionally invoking garbage collection of the local heap, so RMI forces full collections periodically. The frequency of these collections can be controlled with properties, as in the following example:

```
java -Dsun.rmi.dgc.client.gcInterval=3600000
    -Dsun.rmi.dgc.server.gcInterval=3600000 ...
```

This example specifies explicit garbage collection once per hour instead of the default rate of once per minute. However, this may also cause some objects to take much longer to be reclaimed. These properties can be set as high as`Long.MAX_VALUE` to make the time between explicit collections effectively infinite if there is no desire for an upper bound on the timeliness of DGC activity.

Soft References

Soft references are kept alive longer in the server virtual machine than in the client. The rate of clearing can be controlled with the command-line option `-XX:SoftRefLRUPolicyMSPerMB=<N>`, which specifies the number of milliseconds (ms) a soft reference will be kept alive (once it is no longer strongly reachable) for each megabyte of free space in the heap. The default value is 1000 ms per megabyte, which means that a soft reference will survive (after the last strong reference to the object has been collected) for 1 second for each megabyte of free space in the heap. This is an approximate figure because soft references are cleared only during garbage collection, which may occur sporadically.

Class Metadata

Java classes have an internal representation within Java Hotspot VM and are referred to as class metadata. In previous releases of Java Hotspot VM, the class metadata was allocated in the so called permanent generation. In JDK 8, the permanent generation was removed and the class metadata is allocated in native memory. The amount of native memory that can be used for class metadata is by default unlimited. Use the option `MaxMetaspaceSize` to put an upper limit on the amount of native memory used for class metadata.

Java Hotspot VM explicitly manages the space used for metadata. Space is requested from the OS and then divided into chunks. A class loader allocates space for metadata from its chunks (a chunk is bound to a specific class loader). When classes are unloaded for a class loader, its chunks are recycled for reuse or returned to the OS. Metadata uses space allocated by `mmap`, not by `malloc`.

If `UseCompressedOops` is turned on and `UseCompressedClassesPointers` is used, then two logically different areas of native memory are used for class metadata. `UseCompressedClassPointers` uses a 32-bit offset to represent the class pointer in a 64-bit process as does `UseCompressedOops` for Java object references. A region is allocated for these compressed class pointers (the 32-bit offsets). The size of the region can be set with`CompressedClassSpaceSize` and is 1 gigabyte (GB) by default. The space for the compressed class pointers is reserved as space allocated by `mmap` at initialization and committed as needed. The `MaxMetaspaceSize` applies to the sum of the committed compressed class space and the space for the other class metadata.

Class metadata is deallocated when the corresponding Java class is unloaded. Java classes are unloaded as a result of garbage collection, and garbage collections may be induced in order to unload classes and deallocate class metadata. When the space committed for class metadata reaches a certain level (a high-water mark), a garbage collection is induced. After the garbage collection, the high-water mark may be raised or lowered depending on the amount of space freed from class metadata. The high-water mark would be raised so as not to induce another garbage collection too soon. The high-water mark is initially set to the value of the command-line option `MetaspaceSize`. It is raised or lowered based on the options `MaxMetaspaceFreeRatio` and `MinMetaspaceFreeRatio`. If the committed space available for class metadata as a percentage of the total committed space for class metadata is greater than `MaxMetaspaceFreeRatio`, then the high-water mark will be lowered. If it is less than `MinMetaspaceFreeRatio`, then the high-water mark will be raised.

Specify a higher value for the option `MetaspaceSize` to avoid early garbage collections induced for class metadata. The amount of class metadata allocated for an application is application-dependent and general guidelines do not exist for the selection of `MetaspaceSize`. The default size of `MetaspaceSize` is platform-dependent and ranges from 12 MB to about 20 MB.

Information about the space used for metadata is included in a printout of the heap. A typical output is shown in Example 11-1, "Typical Heap Printout".

Example 11-1 Typical Heap Printout

```
Heap
  PSYoungGen      total 10752K, used 4419K
    [0xfffffffff6ac00000, 0xfffffffff6b800000, 0xfffffffff6b800000)
    eden space 9216K, 47% used
      [0xfffffffff6ac00000,0xfffffffff6b050d68,0xfffffffff6b500000)
    from space 1536K, 0% used
      [0xfffffffff6b680000,0xfffffffff6b680000,0xfffffffff6b800000)
    to   space 1536K, 0% used
      [0xfffffffff6b500000,0xfffffffff6b500000,0xfffffffff6b680000)
  ParOldGen       total 20480K, used 20011K
      [0xfffffffff69800000, 0xfffffffff6ac00000, 0xfffffffff6ac00000)
    object space 20480K, 97% used
      [0xfffffffff69800000,0xfffffffff6ab8add8,0xfffffffff6ac00000)
  Metaspace       used 2425K, capacity 4498K, committed 4864K, reserved 1056768K
    class space   used 262K, capacity 386K, committed 512K, reserved 1048576K
```

In the line beginning with `Metaspace`, the `used` value is the amount of space used for loaded classes. The `capacity` value is the space available for metadata in currently allocated chunks. The `committed` value is the amount of space available for chunks. The `reserved` value is the amount of space reserved (but not necessarily committed) for metadata. The line beginning with `class space` line contains the corresponding values for the metadata for compressed class pointers.