Tuning Garbage Collection with the 5.0 Java ™ Virtual Machine

Table of Contents

Introduction

The Java ™ 2 Platform Standard Edition (J2SE ™ platform) is used for a wide variety of applications from small applets on desktops to web services on large servers. In the J2SE platform version 1.4.2 there were four garbage collectors from which to choose but without an explicit choice by the user the serial garbage collector was always chosen. In version 5.0 the choice of the collector is based on the class of the machine on which the application is started.

This “smarter choice” of the garbage collector is generally better but is not always the best. For the user who wants to make their own choice of garbage collectors, this document will provide information on which to base that choice. This will first include the general features of the garbage collections and tuning options to take the best advantage of those features. The examples are given in the context of the serial, stop-the-world collector. Then specific features of the other collectors will be discussed along with factors that should considered when choosing one of the other collectors.

When does the choice of a garbage collector matter to the user? For many applications it doesn't. That is, the application can perform within its specifications in the presence of garbage collection with pauses of modest frequency and duration. An example where this is not the case (when the serial collector is used) would be a large application that scales well to large number of threads, processors, sockets, and a large amount of memory.

Amdahl observed that most workloads cannot be perfectly parallelized; some portion is always sequential and does not benefit from parallelism. This is also true for the J2SE platform. In particular, virtual machines for the Java ™ platform up to and including version 1.3.1 do not have parallel garbage collection, so the impact of garbage collection on a multiprocessor system grows relative to an otherwise parallel application.

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

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 well worthwhile to choose the right garbage collector and to tune it if necessary.

The serial collector will be adequate for the majority of applications. Each of the other collectors have some added overhead and/or complexity which is the price for specialized behavior. If the application doesn't need the specialized behavior of an alternate collector, use the serial collector. An example of a situation where the serial collector is not expected to be the best choice is a large application that is heavily threaded and run on hardware with a large amount of memory and a large number of processors. For such applications, we now make the choice of the throughput collector (see the discussion of ergonomics in section 2).

This document was written using the J2SE Platform version 1.5, on the Solaris ™ Operating System (SPARC (R) Platform Edition) as the base platform, because it provides the most scalable hardware and software for the J2SE platform.

However, the descriptive text applies to other supported platforms, including Linux, Microsoft Windows, and the Solaris Operating System (x86 Platform Edition), to the extent that scalable hardware is available. Although command line options are consistent across platforms, some platforms may have defaults different than those described here.

• Ergonomics

New in the J2SE Platform version 1.5 is a feature referred to here as ergonomics. The goal of ergonomics is to provide good performance from the JVM with a minimum of command line tuning. Ergonomics attempts to match the best selection of

- Garbage collector
- Heap size
- Runtime compiler

for an application. This selection assumes that the class of the machine on which the application is run is a hint as to the characteristics of the application (i.e., large applications run on large machines). In addition to these selections is a simplified way of tuning garbage collection. With the throughput collector the user can specify goals for a maximum pause time and a desired throughput for an application. This is in contrast to specifying the size of the heap that is needed for good performance. This is intended to particularly improve the performance of large applications that use large heaps. The more general ergonomics is described in the document entitled “Ergonomics in the 1.5 Java Virtual Machine”. It is recommended that the ergonomics as presented in this latter document be tried before using the more detailed controls explained in this document.

Included in this document under the throughput collector are the ergonomics features that are provided as part of the new adaptive size policy. This includes the new options to specify goals for the performance of garbage collection and additional options to fine tune that performance.

Generations

One strength of the J2SE platform is that it shields the developer from the complexity of memory allocation and garbage collection. However, once garbage collection is the principal bottleneck, it is worth understanding 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 simply iterate over every reachable object. Any objects left over are then 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.

Beginning with the J2SE Platform version 1.2, the virtual machine incorporated 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 avoid extra work.

The most important of these observed properties is *infant mortality*. The blue area in the diagram below 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 (i.e., have "died") shortly after being allocated. Iterator objects, for example, are often alive for the duration of a single loop.

Some objects do live longer, and so the distribution stretches out to the 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 infant mortality 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,* or memory pools holding objects of different ages. Garbage collection occurs in each generation when the generation fills up. Objects are allocated in a generation for younger objects or the *young* generation, and because of infant mortality most objects die there. When the *young* generation fills up it causes a *minor collection.* Minor collections can be optimized assuming a high infant mortality rate.

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. Some surviving objects are moved to a *tenured* generation. When the *tenured* generation needs to be collected there is a *major collection* that is often much slower because it involves all live objects.

The diagram below shows *minor collections* occurring at intervals long enough to allow many of the objects to die between collections. It is well-tuned in the sense that the *young* generation is large enough (and thus the period between minor collections long enough) that the minor collection can take advantage of the high infant mortality rate. This situation can be upset by applications with unusual lifetime distributions, or by poorly sized generations that cause collections to occur before objects have had time to die.

As noted in section 2 ergonomics nows makes different choice of the garbage collector in order to provide good performance on a variety of applications. The serial garbage collector is meant to be used by small applications. Its default parameters were designed to be effective for most small applications. The throughput garbage collector is meant to be used by large applications. 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 for many applications but do not always work. This leads to the central tenet of this document:

If the garbage collector has become a bottleneck, you may wish to customize the generation sizes. Check the verbose garbage collector output, and then explore the sensitivity of your individual performance metric to the garbage collector parameters.

The default arrangement of generations (for all collectors with the exception of the throughput collector) looks something like this.

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

The *young* generation consists of *eden* plus two *survivor* spaces . Objects are initially allocated in eden. One *survivor* space is empty at any time, and serves as a destination of the next, copying collection of any live objects in eden and the

other survivor space. Objects are copied between survivor spaces in this way until they are old enough to be tenured, or copied to the *tenured* generation.

Other virtual machines, including the production virtual machine for the J2SE Platform version 1.2 for the Solaris Operating System, used two equally sized spaces for copying rather than one large eden plus two small spaces. This means the options for sizing the *young* generation are not directly comparable; see the Performance FAQ for an example.

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

3.1 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, since 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, a particular generation sizing chooses 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. To a first approximation, the sizing of one generation does not affect the collection frequency and pause times for another

generation.

There is no one right way to size generations. The best choice is determined by the way the application uses memory as well as user requirements. For this reason the virtual machine's choice of a garbage collectior are not always optimal, and may be overridden by the user in the form of command line options, described below.

3.2 Measurement

Throughput and footprint are best measured using metrics particular to the application. For example, throughput of a web server may be tested using a client load generator, while footprint of the server might be measured on the Solaris Operating System using the pmap command. On the other hand, pauses due to garbage collection are easily estimated by inspecting the diagnostic output of the virtual machine itself.

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

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

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

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

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

(776768K)(in the first line)

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

0.2300771 secs (in the first line)

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

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

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

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

and took about 46 milliseconds.

0.0457646 secs

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

```
196016K->133633K(261184K)
```

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

0.0459067 secs

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

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

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

```
18154K->2311K(24576K)
```

and took about .13 seconds.

0.1290354 secs

- Sizing the Generations

A number of parameters affect generation size. The following diagram 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, 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 (*permanent* generation, *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. These parameters are discussed below.

The discussion that follows regarding the growing and shrinking of the heap does not apply to the throughput collector. The resizing of the heap for the throughput collector is governed by the ergonomics discussed in section 5.2.2. The

parameters that control the total size of the heap and the sizes of the generations do apply to the throughput collector.

4.1 Total Heap

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

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 and above by -Xmx . The default parameters for the 32-bit Solaris Operating System (SPARC Platform Edition) are shown in this table:

-XX:MinHeapFreeRatio=	40
-XX:MaxHeapFreeRatio=	70
-Xms	3670k
-Xmx	64m

Default values of heap size parameters on 64-bit systems have been scaled up by approximately 30%. This increase is meant to compensate for the larger size of objects on a 64-bit system.

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

space free as long as shrinking the generation does not decrease it below the minimum size of the generation.

Large server applications often experience two problems with these defaults. One is slow startup, because the initial heap is small and must be resized over many *major* collections. A more pressing problem is that the default maximum heap size is unreasonably small for most server applications. The rules of thumb for server applications are:

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

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

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

A description of other virtual machine options can be found at

Link

4.2 The *Young* Generation

The second most influential knob 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 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 equal to one another fixes the *young* generation, just as setting -Xms and -Xmx equal fixes the total heap size. This is useful for tuning the *young* generation at a finer granularity than the integral multiples allowed by NewRatio.

4.2.1 *Young* Generation Guarantee

In an ideal minor collection the live objects are copied from one part of the *young* generation (the eden space plus the first survivor space) to another part of the *young* generation (the second survivor space). However, there is no guarantee that all the live objects will fit into the second survivor space. To ensure that the minor collection can complete even if all the objects are live, enough free memory must be reserved in the *tenured* generation to accommodate all the live objects. In the worst case, this reserved memory is equal to the size of eden plus the objects in non-empty survivor space. When there isn't enough memory available in the *tenured* generation for this worst case, a major collection will occur instead. This policy is fine for small applications, because the memory reserved in the *tenured* generation is typically only virtually committed but not actually used. But for applications needing the largest possible heap, an eden bigger than half the virtually committed size of the heap is useless: only major collections would occur. Note that the *young* generation guarantee applies only to serial collector . The throughput collector and the concurrent collector will proceed with a *young* generation collection, and if the *tenured* generation cannot accommodate all the promotions from the *young* generation, both generations are collected.

If desired, the parameter SurvivorRatio can be used to tune the size of the survivor spaces, but this is often not as important for performance. For example, -XX:SurvivorRatio=6 sets the ratio between each survivor space and eden to be 1:6. In other words, each survivor space will be one eighth 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 of times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full. An option, -XX:+PrintTenuringDistribution, can be used to show this threshold and the ages of objects in the new generation. It is also useful for observing the lifetime distribution of an application.

Here are the default values for the 32-bit Solaris Operating System (SPARC Platform Edition):

NewRatio	2 *(client JVM: 8)*
NewSize	2228k
MaxNewSize	Not limited
SurvivorRatio	32

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

The rules of thumb for server applications are:

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

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

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

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

Types of Collectors

The discussion to this point has been about the serial collector. In the J2SE Platform version 1.5 there are three additional collectors. Each is a generational collector which has been implemented to emphasize the throughput of the application or low garbage collection pause times.

- The *throughput* collector: this collector uses a parallel version of the *young* generation collector. It is used if the

-XX:+UseParallelGC option is passed on the command line. The *tenured* generation collector is the same as the serial collector.

- The *concurrent* low pause collector: this collector is used if the -Xincgc ™ or -XX:+UseConcMarkSweepGC is passed on the command line. The concurrent collector is used to collect the *tenured* generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection. A parallel version of the *young* generation copying collector is used with the concurrent collector. The concurrent low pause collector is used if the option -XX:+UseConcMarkSweepGC is passed on the command line.
- The *incremental* (sometimes called *train*) low pause collector: this collector is used only if -XX:+UseTrainGC is passed on the command line. This collector has not changed since the J2SE Platform version 1.4.2 and is currently not under active development. It will not be supported in future releases. Please see the 1.4.2 GC Tuning Document for information on this collector.

Note that -XX:+UseParallelGC should not be used with -XX:+UseConcMarkSweepGC . The argument parsing in the J2SE Platform starting with version 1.4.2 should only allow legal combinations of command line options for garbage collectors, but earlier releases may not detect all illegal combinations and the results for illegal combinations are unpredictable.

Always try the collector chosen by the JVM on your application before explicitly selecting another collector. Tune the heap size for your application and then consider what requirements of your application are not being met. Based on the latter, consider using one of the other collectors.

5.1 When to Use the Throughput Collector

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

5.2 The Throughput Collector

The throughput collector is a generational collector similar to the serial collector but with multiple threads used to do the minor collection. The major collections are essentially the same as with the serial collector. By default on a host with *N*

CPUs, the throughput collector uses *N* garbage collector threads in the minor collection. The number of garbage collector threads can be controlled with a command line option (see below). On a host with 1 CPU the throughput collector will likely not perform as well as the serial collector because of the additional overhead for the parallel execution (e.g., synchronization costs). On a host with 2 CPUs the throughput collector generally performs as well as the serial garbage collector and a reduction in the minor garbage collector pause times can be expected on hosts with more than 2 CPUs.

The throughput collector can be enabled by using command line flag -XX:+UseParallelGC. The number of garbage collector threads can be controlled with the ParallelGCThreads command line option (`-XX:ParallelGCThreads=<desired number>`). If explicit tuning of the heap is being done with command line flags the size of the heap needed for good performance with the throughput collector is to first order the same as needed with the serial collector. Turning on the throughput collector should just make the minor collection pauses shorter. Because there are multiple garbage collector threads participating in the minor collection there is a small possibility of fragmentation due to promotions from the *young* generation to the *tenured* generation during the collection. Each garbage collection thread 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 will reduce this fragmentation effect as will increasing the size of the *tenured* generation. 5.

5.2.1 Generations in the throughput collector

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

5.2.2 Ergonomics in the throughput collector

In the J2SE Platform version 1.5 the throughput collector will be chosen as the garbage collector on server class machines. The document Ergonomics in the 5 Java Virtual Machine discusses this selection of the garbage collector. For the throughput collector a new method of tuning has been added which is based on a desired behavior of the application with respect to garbage collection. The following command line flags can be used to specify the desired behavior in terms of goals for the maximum pause time and the throughput for the application.

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

```
-XX:MaxGCPauseMillis=<nnn>
```

This is interpreted as a hint to the throughput collector that pause times of `<nnn>` milliseconds or less are desired. By default there is no maximum pause time goal. The throughput collector will adjust the Java heap size and other garbage collection related parameters in an attempt to keep garbage collection pauses shorter than `<nnn>` milliseconds. These adjustments may cause the garbage collector to reduce overall throughput of the application and in some cases the desired pause time goal cannot be met. By default no maximum pause time goal is set.

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

```
-XX:GCTimeRatio=<nnn>
```

The ratio of garbage collection time to application time is

```
1 / (1 + <nnn>)
```

For example -XX:GCTimeRatio=19 sets a goal of 5% of the total time for garbage collection. By default the goal for total time for garbage collection is 1%.

Additionally, as an implicit goal the throughput collector will try to met the other goals in the smallest heap that it can.

5.2.2.1 Priority of goals

The goals are addressed in the following order

- Maximum pause time goal
- Throughput goal
- 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.

5.2.2.2 Adjusting Generation Sizes

The statistics (e.g., average pause time) kept by the collector are updated at the end of a 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 (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. 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 flag `-XX:YoungGenerationSizeIncrement=<nnn >` for the young generation and `-XX:TenuredGenerationSizeIncrement=<nnn>` for the tenured generation. The percentage by which a generation shrinks is adjusted by the command line flag `-XX: AdaptiveSizeDecrementScaleFactor=<nnn >.` If the size of an increment for growing is XXX percent, the size of the decrement for shrinking will be XXX / nnn percent.

If the collector decides to grow a generation at startup, there is a supplemental percentage added to the increment. This supplement decays with the number of collections and there is no long term affect of this supplement. 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, the size of only one generation is shrunk at a time. If the pause times of both generations are above the goal, 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%.

5.2.2.3 Heap Size

If not otherwise set on the command line, the sizes of the initial heap and maximum heap are calculated based on the size of the physical memory. If phys_mem is the size of the physical memory on the platform, the initial heap size will be set to phys_mem / DefaultInitialRAMFraction. DefaultInitialRAMFraction is a command line option with a default value of 64. Similarly the maximum heap size will be set to phys_mem / DefaultMaxRAM. DefaultMaxRAMFraction has a default value of 4.

5.2.3 Out-of-Memory Exceptions

The throughput collector will throw an out-of-memory exception if too much time is being spent doing garbage collection. For example, if the JVM is spending more than 98% of the total time doing garbage collection and is recovering less than 2% of the heap, it will throw an out-of-memory expection. The implementation of this feature has changed in 1.5. The policy is the same but there may be slight differences in behavior due to the new implementation.

5.2.4 Measurements with the Throughput Collector

The verbose garbage collector output is the same for the throughput collector as with the serial collector.

5.3 When to Use the Concurrent Low Pause Collector

Use the concurrent low pause collector if your application would benefit from shorter garbage collector pauses and can afford to share processor resources with the garbage collector when the application is running. Typically applications which 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. Optimal results have been observed for interactive applications with *tenured* generations of a modest size on a single processor.

5.4 The Concurrent Low Pause Collector

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

collected concurrently with this collector.

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

5.4.1 Overhead of Concurrency

The concurrent collector trades processor resources (which would otherwise be available to the application) for shorter major collection pause times. The concurrent part of the collection is done by a single garbage collection thread. On an *N* processor system when the concurrent part of the collection is running, it will be using 1/ *N*th of the available processor power. On a uniprocessor machine it would be fortuitous if it provided any advantage (see the section on Incremental mode for the exception to this statement). The concurrent collector also has some additional overhead costs that will take away from the throughput of the applications, and some inherent disadvantages (e.g., fragmentation) for some types of applications. On a two processor machine there is a processor available for applications threads while the concurrent part of the collection is running, so running the concurrent garbage collector thread does not "pause" the application. There may be reduced pause times as intended for the concurrent collector but again less processor resources are available to the application and some slowdown of the application should be expected. As *N* increases, the reduction in processor resources due to the running of the concurrent garbage collector thread becomes less, and the advantages of the concurrent collector become more.

5.4.2 *Young* Generation Guarantee

Prior to J2SE Platform version 1.5 the concurrent collector had to satisfy the young generation guarantee just as the serial

collector does. Starting with J2SE Platform version 1.5 this is no longer true. The concurrent collector can recover if it starts a young generation collection and there is not enough space in the tenured generation to hold all the objects that require promotion from the young generation. This is similar to the throughput collector.

5.4.3 Full Collections

The concurrent collector uses a single garbage collector thread that runs simultaneously with the application threads with the goal of completing the collection of the *tenured* generation before it becomes full. In normal operation, the concurrent collector is able to do most of its work with the application threads still running, so only brief pauses are seen by the application threads. As a fall back, if the concurrent collector is unable to finish before the *tenured* generation fills up, the application is paused and the collection is completed with all the application threads stopped. Such collections with the application stopped are referred to as full collections and are a sign that some adjustments need to be made to the concurrent collection parameters.

5.4.4 Floating Garbage

A garbage collector works to find the live objects in the heap. Because application threads and the garbage collector thread run concurrently during a major collection, objects that are found to be alive by the garbage collector thread may become dead by the time collection finishes. Such objects are referred to as floating garbage. The amount of floating garbage depends on the length of the concurrent collection (more time for the applications threads to discard an object) and on the particulars of the application. As a rough rule of thumb try increasing the size of the *tenured* generation by 20% to account for the floating garbage. Floating garbage is collected at the next garbage collection.

5.4.5 Pauses

The concurrent 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 (e.g., objects on thread stack, static objects and so on) and elsewhere in the heap (e.g., the *young* generation). This first pause is referred to as the initial mark. The second pause comes at the end of the marking phase and finds objects that were missed during the concurrent marking phase due to the concurrent execution of the application threads. The second pause is referred to as the remark.

5.4.6 Concurrent Phases

The concurrent marking occurs between the initial mark and the remark. During the concurrent marking the concurrent garbage collector thread is executing and using processor resources that would otherwise be available to the application. After the remark there is a concurrent sweeping phase which collects the dead objects. During this phase the concurrent garbage collector thread is again taking processor resources from the application. After the sweeping phase the concurrent collector sleeps until the start of the next major collection.

5.4.7 Scheduling a collection

With the serial collector a major collection is started when the tenured generation becomes full and all application threads are stopped while the collection is done. In contrast a concurrent collection should be started at a time such that the collection can finish before the tenured generation becomes full. There are several ways a concurrent collection can be started.

The concurrent collector keeps statistics on the time remaining before the tenured generation is full (T-until-full) and on the time needed to do a concurrent collection (T-collect). When the T-until-full approaches T-collect, a concurrent collection is started. This test is appropriately padded so as to start a collection conservatively early.

A concurrent collection will also start if the occupancy of the tenured generation grows above the initiating occupancy (i.e., the percentage of the current heap that is used before a concurrent collection is started). The initiating occupancy by default is set to about 68%. It can be set with the parameter CMSInitiatingOccupancyFraction which can be set on the command line with the flag

```
-XX:CMSInitiatingOccupancyFraction=<nn>
```

The value <nn> is a percentage of the current tenured generation size.

5.4.8 Scheduling pauses

The pauses for the young generation collection and the tenured generation collection occur independently. They cannot

overlap, but they can 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 remark pauses for a concurrent collection are scheduled to be midway between the previous and next young generation pauses. The initial mark pause is typically too short to be worth scheduling.

5.4.9 Incremental mode

The concurrent 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 a processor. The incremental mode is meant to lessen the impact 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 by concurrently by the collector into small chunks of time which are scheduled between young generation collections. This feature is useful when applications that need the low pause times provided by the concurrent collector are run on machines with small numbers of processors (e.g., 1 or 2).

The concurrent collection cycle typically includes the following steps:

- stop all application threads; do the initial mark; resume all application threads
- do the concurrent mark (uses one procesor for the concurrent work)
- do the concurrent pre-clean (uses one processor for the concurrent work)
- stop all application threads; do the remark; resume all application threads
- do the concurrent sweep (uses one processor for the concurrent work)
- do the concurrent reset (uses one processor for the concurrent work)

Normally, the concurrent collector uses one processor for the concurrent work for the entire concurrent mark phase, without (voluntarily) relinquishing it. Similarly, one processor is used for the entire concurrent sweep phase, again without relinquishing it. This processor utilization can be too much of a disruption for applications with pause time constraints, particularly when run on systems with just one or two processors. i-cms solves this problem by breaking up the concurrent phases into short bursts of activity, which are scheduled to occur mid-way between minor pauses.

I-cms uses a "duty cycle" to control the amount of work the concurrent 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 concurrent collector is allowed to run. I-cms can automatically compute the duty cycle based on the behavior of the application (the recommended method), or the duty cycle can be set to a fixed value on the command line.

5.4.9.1 Command line

The following command-line options control i-cms (see below for recommendations for an initial set of options):

`-XX:+CMSIncrementalMode default: disabled`

This flag enables the incremental mode. Note that the concurrent collector must be enabled (with -XX:+UseConcMarkSweepGC) for this option to work.

`-XX:+CMSIncrementalPacing default: disabled`

This flag enables automatic adjustment of the incremental mode duty cycle based on statistics collected while the JVM is running.

`-XX:CMSIncrementalDutyCycle=<N> default: 50`

This is the percentage (0-100) of time between minor collections that the concurrent collector is allowed to run. If CMSIncrementalPacing is enabled, then this is just the initial value.

`-XX:CMSIncrementalDutyCycleMin=<N> default: 10`

This is the percentage (0-100) which is the lower bound on the duty cycle when CMSIncrementalPacing is enabled.

`-XX:CMSIncrementalSafetyFactor=<N> default: 10`

This is the percentage (0-100) used to add conservatism when computing the duty cycle.

```
-XX:CMSIncrementalOffset=<N> default: 0
```

This is the percentage (0-100) by which the incremental mode duty cycle is shifted to the right within the period between minor collections.

```
-XX:CMSExpAvgFactor=<N> default: 25
```

This is the percentage (0-100) used to weight the current sample when computing exponential averages for the concurrent collection statistics.

5.4.9.2 Recommended Options for i-cms

When trying i-cms, we recommend the following as an initial set of command line options:

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

The first three options enable the concurrent collector, i-cms, and i-cms automatic pacing. The next two set the minimum duty cycle to 0 and the initial duty cycle to 10, since the default values (10 and 50, respectively) are too large for a number of applications. The last three options cause diagnostic information on the collection to be written to stdout, so that the behavior of i-cms can be seen and later analyzed.

5.4.9.3 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, try the following steps, one at a time:

Increase the safety factor:

```
-XX:CMSIncrementalSafetyFactor=<N>
```

Increase the minimum duty cycle:

```
-XX:CMSIncrementalDutyCycleMin=<N>
```

Disable automatic pacing and use a fixed duty cycle:

```
-XX:-CMSIncrementalPacing -XX:CMSIncrementalDutyCycle=<N>
```

5.4.10 Measurements with the Concurrent Collector

Below is output for -verbose:gc with -XX:+PrintGCDetails (some details have been removed). Note that the output for the concurrent collector is interspersed with the output from the minor collections. Typically many minor collections will occur during a concurrent collection cycle. The CMS-initial-mark: indicates the start of the concurrent collection cycle. The 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 before is the precleaning phase indicated by CMS-concurrent-preclean:. Precleaning represents work that can be done concurrently and is in preparation for the remark phase CMS-remark. The final phase is indicated by the CMS-concurrent-reset: and is in preparation for the next concurrent collection.

COPY

```
[GC [1 CMS-initial-mark:
13991K(20288K)]
    [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 times of the concurrent phases (concurrent mark, concurrent precleaning, and concurrent sweep) may be relatively long (as in the example above) when compared to a minor collection pause but the application is not paused during the concurrent phases. The remark pause is affected by the specifics of the application (e.g., a higher rate of modifying objects can increase this pause) and the time since the last minor collection (i.e., more objects in the *young* generation may increase this pause).

Other Considerations

For most applications the *permanent* generation is not relevant to garbage collector performance. However, some applications dynamically generate and load many classes. For instance, some implementations of JSP ™ pages do this. If necessary, the maximum *permanent* generation size can be increased with MaxPermSize.

Some applications interact with garbage collection by using finalization and weak/soft/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.

Another way applications can interact with garbage collection is by invoking full garbage collections explicitly, such as through the System.gc() call. These calls force major collection, and inhibit scalability on large systems. The performance impact of explicit garbage collections can be measured by disabling explicit garbage collections using the flag -XX:+DisableExplicitGC.

One of the most commonly encountered uses of explicit garbage collection occurs with RMI's distributed garbage collection (DGC). Applications using RMI refer to objects in other virtual machines. Garbage can't be collected in these distributed applications without occasional local collection, so RMI forces periodic full collection. The frequency of these collections can be controlled with properties. For example,

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

specifies explicit 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.

The Solaris 8 Operating System supports an alternate version of libthread that binds threads to light-weight processes (LWPs) directly. Some applications can benefit greatly from the use of the alternate libthread. This is a potential benefit for any threaded application. To try this, set the environment variable LD_LIBRARY_PATH to include /usr/lib/lwp before launching the virtual machine. The alternate libthread is the default libthread in the Solaris 9 Operating System.

Soft references are cleared less aggressively in the server virtual machine than the client. The rate of clearing can be slowed by increasing the parameter SoftRefLRUPolicyMSPerMB with the command line flag -XX:SoftRefLRUPolicyMSPerMB=10000. SoftRefLRUPolicyMSPerMB is a measure of the time that a soft reference survives for a given amount of free space in the heap. The default value is 1000 ms per megabyte. This can be read to mean 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 very approximate.

Conclusion

Garbage collection can become a bottleneck in different applications depending on the requirements of the applications. By understanding the requirements of the application and the garbage collection options, it is possible to minimize the impact of garbage collection.

Other Documentation

8.1 Example of Output

The GC output examples document contains examples for different types of garbage collector behavior. The examples show the diagnostic output from the garbage collector and explain how to recognize various problems. Examples from different collectors are included.

8.2 Frequently Asked Questions

A FAQ is included that contains answers to specific questions. The level of detail in the FAQ is generally greater than in this tuning document.

As used on the web site, the terms "Java Virtual Machine" and "JVM" mean a virtual machine for the Java platform.