Diagnostics Guide

prev next contents index view as PDF get Adobe Reader

Tuning the Memory Management System

Memory management is all about allocation of objects. One part of the memory management system finds a free spot for the new object, while another part garbage collects old objects to create more free space for new objects. The more objects a Java application allocates, the more resources will be used for memory management. A correctly tuned memory management system minimizes the overhead inflicted by garbage collection and makes object allocation fast. You can read more about how memory management in the Oracle JRockit JVM works in Understanding Memory Management. This section covers the most important options available for tuning the memory management system in the JVM. The following topics are covered:

- Optimizing Memory Allocation Performance
- Selecting and Tuning a Garbage Collector
- Tuning the Compaction of Memory
- Optimizing Memory Allocation Performance

Setting the Heap and Nursery Size

The heap is the area where the Java objects reside. The heap size has an impact on the JVM's performance, and thus also on the Java application's performance.

When the JVM uses a generational garbage collection strategy, a part of the heap is reserved for the nursery. All small objects are allocated in the *nursery*, also known as *young space*. When the nursery becomes full, a *young collection* is performed, where objects that have lived long enough in the nursery are moved to *old space*, which is the rest of the heap.

To distinguish between recently allocated objects and objects that have been around for a while in the nursery, the JVM uses a *keep area*. The keep area contains the most recently allocated objects in the nursery and is not garbage collected until the next young collection.

Setting the Heap Size

Command line options: **`-Xms:<min size> -Xmx:<max size>`**

The heap size has an impact on allocation speed, garbage collection frequency and garbage collection times. A small heap will become full quickly and must be garbage collected more often. It is also prone to more fragmentation, making object allocation slower. A large heap introduces a slight overhead in garbage collection times. A heap that is larger than the available physical memory in the system must be paged out to disk, which leads to long access times or even application freezes, especially during garbage collection.

In general, the extra overhead caused by a larger heap is smaller than the gains in garbage collection frequency and allocation speed, as long as the heap doesn't get paged to disk. Thus a good heap size setting would be a heap that is as large as possible within the available physical memory.

There are two parameters for setting the heap size:

- `-Xms:<size>`, which sets the initial and minimum heap size
- `-Xmx:<size>`, which sets the maximum heap size

For example:

```
java -Xms:1g -Xmx:1g MyApplication
```

This starts up the JVM with a heap size fixed to 1 GB.

For default values and limitations, see the documentation on `-Xms` and `-Xmx`.

If the optimal heap size for the application is known, we recommend that you set `-Xms` and `-Xmx` to the same value. This gives you a controlled environment where you get a good heap size right from the start.

Setting the Heap Size on 64-bit Systems

On 64-bit systems, a memory address is 64 bits long, which makes it possible to address much more memory than with a 32-bit address; on the other hand, each address reference requires twice as much memory. To reduce the memory usage for address references on 64-bit systems, the JRockit JVM can use *compressed references*. Compressed references reduce the address references to 32 bits, and can be used as long as the entire heap can be addressed with 32 bits. So on a 64.bit system, you will usually benefit from setting the maximum heap size below 4 GB as long as the amount of live data is less than 3-4 GB. Compressed references are enabled by default whenever applicable.

Note: When you run the JRockit JVM on a 64-bit system with a heap size less than 4 GB, if native OutOfMemory errors occur despite memory being available, try disabling compressed references by using the -XxcompressedRefs=0 option.

Setting the Nursery and Keep Area Size

Command line option: `-Xns:<nursery size>`

The size of the nursery has an impact on allocation speed, garbage collection frequency and garbage collection times. A small nursery will become full quickly and must be garbage collected more often, while garbage collection of a large nursery takes slightly longer time. A nursery that is so small that few or no objects have died before a young collection is started is of very little use, and neither is a nursery that is so large that no young collections are performed between garbage collections of the whole heap that are triggered due to allocation of large objects in old space.

An optimal nursery size for maximum application throughput is such that as many objects as possible are garbage collected by young collection rather than old collection. This value approximates to about half of the free heap. In the JRockit JVM R27.3.0 and later versions, the dynamic garbage collection mode optimized for throughput, `-Xgcprio:throughput`, and the static generational parallel garbage collector, `-Xgc:genpar`, will dynamically set the nursery size to an approximation of the optimal value.

The optimal nursery size for throughput is often quite large, which may lead to long young collection times. Since all Java threads are paused while the young collection is performed, you may want to reduce the nursery size below the optimal value to reduce the young collection pause times.

The nursery size is set using the command line option `-Xns:<size>`. For example:

```
java -Xns:100m MyApplication
```

This starts up the JVM with a fixed nursery size of 100 MB.

For default values and limitations, see the documentation on `-Xns`.

Keep Area

Command line option: `-XXkeepAreaRatio:<percentage>`

The keep area size has an impact on both old collection and young collection frequency. A large keep area causes more frequent young collections, while a keep area that is too small causes more frequent old collections when objects are promoted prematurely.

An optimal keep area size is as small as possible while maintaining a low promotion ratio. The promotion ratio can be observed in JRA recordings (see Using Oracle JRockit Mission Control Tools for more information) and verbose outputs from `-Xverbose:memory=debug`, as well as in the garbage collection report printed out by `-XgcReport`. By default the keep area is 25% of the nursery.

The keep area size can be changed using the command line option `-XXkeepAreaRatio:<percentage>`, and is defined as a percentage of the nursery. For example:

```
java -XXkeepAreaRatio:10 MyApplication
```

This starts up the JVM with a keep area that is 10% of the nursery size.

Selecting and Tuning a Garbage Collector

Garbage collection of objects is a necessary evil. Without garbage collection the automatic memory management system would not work, and either the application developers would have to somehow recycle the memory themselves or the application would after a while use up all the memory in the system until it can't continue running as further memory allocation becomes impossible.

The impact of garbage collection can be distributed in different ways depending on the choice of the garbage collection method. The JRockit JVM offers several *garbage collection modes*, which use one or several *garbage collection strategies*. The garbage collection modes are either dynamic, which select the best garbage collection strategy for a given goal, or static, allowing the user to select a garbage collection strategy of their choice. You can select a dynamic garbage collection mode by using the command line option `-XgcPrio:<mode>`, or set a static garbage collector with `-Xgc:<strategy>`.

Selecting a Dynamic Garbage Collection Mode

The dynamic garbage collection modes adjust the memory management system in runtime, optimizing for a specific goal depending on which mode is used. There are three dynamic garbage collection modes:

- `throughput`, which optimizes the garbage collector for maximum application throughput
- `pausetime`, which optimizes the garbage collector for short and even pausetimes
- `deterministic`, which optimizes the garbage collector for very short and deterministic pause times

The dynamic garbage collection modes use advanced heuristics to tune the following parameters in runtime:

- Garbage collection strategy
- Nursery size
- Compaction amount and type

Use a dynamic garbage collection mode if you don't want to go through the time consuming process of tuning these parameters manually, or when a static environment isn't optimal for your application.

Throughput Mode

Command line option: `-XgcPrio:throughput`

The dynamic garbage collection mode optimizing over application throughput uses as little CPU resources as possible for garbage collection, thus giving the Java application as many CPU cycles as possible. The JRockit JVM achieves this by using a parallel garbage collection strategy that stops the Java application during the whole garbage collection duration and uses all CPUs available to perform the garbage collection. Each individual garbage collection pause may be long, but in total the garbage collector takes as little CPU time as possible.

Use throughput mode for applications that demand a high throughput but are not very sensitive to the occasional long garbage collection pause.

Throughput mode is default when the JVM runs in `-server` mode (which is default), or can be enabled with the command line option `-XgcPrio:throughput`. For example:

```
java -XgcPrio:throughput MyApplication
```

This starts up the JVM with the garbage collection mode optimized for throughput.

For more information, see the documentation on `-XgcPrio`.

Pausetime Mode

Command line option: `-XgcPrio:pausetime`

The dynamic garbage collection mode optimizing over pause times aims to keep the garbage collection pauses below a given pause target while maintaining as high throughput as possible. The JRockit JVM achieves this by choosing between a mostly concurrent garbage collection strategy that allows the Java application to continue running during large portions of the garbage collection duration, and a parallel garbage collection strategy that stops the Java application during the entire garbage collection duration. The mostly concurrent garbage collector introduces some extra overhead in keeping track of changes during the concurrent phases, and will also cause more frequent garbage collections. This will lower the overall throughput somewhat, but keeps down the individual garbage collection pauses.

Use pausetime mode for applications that are sensitive to long latencies, for example transaction based systems where transaction times must be stable.

Pausetime mode is enabled with the command line option `-XgcPrio:pausetime`. For example:

```
java -XgcPrio:pausetime MyApplication
```

This starts up the JVM with the garbage collection mode optimized for short pauses.

For more information, see the documentation on `-XgcPrio`.

Setting a Pause Target for Pausetime Mode

Command line option: `-XpauseTarget:<time in ms>`

The pausetime mode uses a pause target for optimizing the pause times. The pause target impacts the application throughput, as a lower pause target will inflict more overhead on the memory management system. Set the pause target as high as your application can tolerate.

The pause target for pausetime mode is by default 500 ms, and can be changed with the command line option `-XpauseTarget:<time in ms>`. For example:

```
java -XgcPrio:pausetime -XpauseTarget:300ms MyApplication
```

This starts up the JVM with the garbage collection optimized for short pauses and a pause target of 300 ms.

For more information, see the documentation on `-XpauseTarget`.

Deterministic Mode

Command line option: `-XgcPrio:deterministic`

The dynamic garbage collection mode optimizing for deterministic pause times is designed to ensure extremely short garbage collection pause times and limit the total pause time within a prescribed window. The JRockit JVM achieves this by using a specially designed mostly concurrent garbage collector, which allows the Java application to continue running as much as possible during the garbage collection.

Use the deterministic mode for applications with strict demands on short and deterministic latencies, for example transaction based applications.

Deterministic mode is enabled with the command line option `-XgcPrio:deterministic`. For example:

```
java -XgcPrio:deterministic MyApplication
```

This starts up the JVM with the garbage collection mode optimized for short and deterministic pauses.

For more information, see the documentation on `-XgcPrio`.

Special Note for WLRT Users

Deterministic garbage collection time can be affected by the JRockit Mission Control Client. While all JRockit Mission Control tools are fully supported when running WLRT with the deterministic garbage collector, you should be aware of some caveats.

- `-Xmanagement` does not prolong deterministic garbage collection pauses by itself, but it does introduce a slightly increased amount of Java code executed by the JVM. This can affect response times and performance compared to not using `-Xmanagement`.
- When making a JRA-recording, disable heap statistics (heapstat) if you run in a latency sensitive situation where you cannot accept the pause for the benefit of the information. Heapstat provides additional bookkeeping of the content of the heap. These statistics are collected at the beginning and at the end of a JRA-recording, inside a pause. You can disable heapstat by using specific arguments when requesting the recording. For more information, please see Creating a JRA Recording with JRockit Mission Control 1.0.
- JRA recordings, even with heapstats disabled, might cause deterministic garbage collection pauses to last slightly longer.
- Memory leak trend analysis can cause longer garbage collection pauses, similar to JRA recordings.
- On requests for more information when the Memory Leak Detector is using its graphical user interface or the Ctrl-Break handler—for example to retrieve the number of instances of a type of object or to retrieve the list of references to an instance or to a class—a longer pause can be introduced.

For more information on JRockit Mission Control, please refer to Using Oracle JRockit Mission Control Tools.

Setting a Pause Target for Deterministic Mode

Command line option: `-XpauseTarget:<time in ms>`

The deterministic mode uses a pause target for optimizing the pause times. The pause target impacts the application throughput, as a lower pause target will inflict more overhead on the memory management system. Set the pause target as high as your application can tolerate.

The garbage collector will aim on keeping the garbage collection pauses below the given pause target. How well it will succeed depends on the application and the hardware. For example, a pause target on 30 ms has been verified on an application with 1 GB heap and an average of 30% live data or less at collection time, running on the following hardware:

- 2 x Intel Xeon 3.6 GHz, 2 MB level 2 cache, 4 GB RAM
- 4 x Intel Xeon 2.0 GHz, 0.5 MB level 2 cache, 8 GB RAM

Running on slower hardware, with a different heap size and/or with more live data might break the deterministic behavior or cause performance degradation over time, while faster hardware or less live data might allow you to set a lower pause target.

The pause target for deterministic mode is by default 30 ms, and can be changed with the command line option `-XpauseTarget:<time>`. For example:

```
java -XgcPrio:deterministic -XpauseTarget:40ms MyApplication
```

This starts up the JVM with the garbage collection optimized for short and deterministic pauses and a pause target of 40ms.

For more information, see the documentation on `-XpauseTarget`.

Selecting a Static Garbage Collection Strategy

Command line option: `-Xgc:<strategy>`

There are four major static garbage collection strategies available.

- `singlepar`, which is a single-generational parallel garbage collector (same as parallel)
- `genpar`, which is a two-generational parallel garbage collector
- `singlecon`, which is a single-generational mostly concurrent garbage collector
- `gencon`, which is a two-generational mostly concurrent garbage collector

When a static garbage collection strategy is selected, the garbage collection strategy will not change automatically in runtime.

Use a static garbage collection strategy if you want a well defined and predictable behavior and are willing to tune the JVM to find the best memory management settings for your application.

Garbage Collector Strategy Selection Workflow

To select the best garbage collection strategy for your application you can follow this workflow:

1. Is your application sensitive to long garbage collection pauses (500 ms or more)?
   - Yes: Select a mostly concurrent garbage collection strategy, `gencon` or `singlecon`
   - No: Select a parallel garbage collection strategy, `genpar` or `singlepar`
1. Does your application allocate a lot of temporary objects?
   - Yes: Select a two-generational garbage collection strategy, `gencon` or `genpar`
   - No: Select a single-generational garbage collection strategy, `singlecon` or `singlepar`

For example, the Oracle WebLogic Sip Server is a transaction based system that allocates new objects for each transaction and has short time-outs for transactions. Long garbage collection pauses would cause transactions to time out, so a mostly concurrent garbage collection should be used. This suggests either `gencon` or `singlecon`. The transactions generate a lot of temporary or short lived objects, which suggests a two-generational garbage collector, `gencon`.

You can set a static garbage collection strategy with the command line option `-Xgc:<strategy>`, for example:

```
java -Xgc:gencon MyApplication
```

This starts up the JVM with the generational concurrent garbage collector.

For more information, see the documentation on `-Xgc`.

Changing Garbage Collection Strategy During Runtime

You can change garbage collector strategies during runtime from the Memory tab of the JRockit Management Console (in JRockit Mission Control) except for when these conditions exist:

- If you are using the dynamic garbage collection mode optimized for deterministic pause times.
- If you are using static single-spaced parallel garbage collection.

For more information, consult the JRockit Management Console's online help.

Tuning the Concurrent Garbage Collection Trigger

Command line option: `-XXgcTrigger:<percentage>`

When you are using a concurrent strategy for garbage collection (in either the mark or the sweep phase, or both), the JRockit JVM dynamically adjusts when to start an old generation garbage collection in order to avoid running out of free heap space during the concurrent phases of the garbage collection. The triggering is based on such characteristics as how much space is available on the heap after previous collections. The JVM dynamically tries to optimize this space and will occasionally run out of free heap during the concurrent garbage collection while it does. When the limit is hit, the verbose printout:

```
[memdbg ] starting parallel sweeping phase
```

appears below the command line (assuming you have set `-Xverbose:memdbg`). This message means that a concurrent sweep could not finish in time and the JVM is using all currently available resources to make up for it. In this case, a parallel sweep is made. If the JVM fails to adapt and the above printout continues to appear, performance is being adversely affected. To avoid this, set the `-XXgcTrigger` option to trigger a garbage collection when there is still X% left of the heap, for example:

```
java -XXgcTrigger=20 MyApplication
```

will trigger an old generation garbage collection when less than 20% of the free heap size is left unused.

If you are using a parallel garbage collection strategy (in both the mark and the sweep phase), then old generation garbage collections are performed whenever the heap is completely full.

Tuning the Compaction of Memory

Compaction is the process of moving chunks of allocated space towards the lower end of the heap, helping create contiguous free memory at the upper end. The JRockit JVM does partial compaction of the heap at each old collection. The size and position of the compaction area as well as the compaction method is selected by advanced heuristics, depending on the garbage collection mode used.

Fragmentation vs. Garbage Collection Pauses

Compaction is performed during garbage collection while all Java threads are paused. Compaction of a large area with many objects will thus increase the garbage collection pause times. On the other hand, insufficient compaction will lead to fragmentation of the heap, which leads to lower performance. If the fragmentation increases over time, the JRockit JVM will eventually be forced to either do a full compaction of the heap, causing a long garbage collection pause, or throw an OutOfMemoryError.

If your application shows performance degradation over time in a periodic manner, such that the performance degrades until it suddenly pops back to excellent, just to start degrading again, you are most likely experiencing fragmentation problems. The heap becomes more and more fragmented for each old collection until finally object allocation becomes impossible and the JVM is forced to do a full compaction of the heap. The full compaction eliminates the fragmentation, but only until the next garbage collection. You can verify this by looking at `-Xverbose:memory` outputs, monitoring the JVM through the Management Console in JRockit Mission Control or by creating a JRA recording and examining the garbage collection data. If you see that the amount of used heap after each old collection keeps increasing over time until it hits the roof, and then drops down again at the next old collection, you are experiencing a fragmentation problem.

Compaction is optimally tuned when the fragmentation is kept on a low and constant level.

Adjusting Compaction

Even though the compaction heuristics in the JRockit JVM are designed to keep the garbage collection pauses low and even, you may sometimes want to limit the compaction ratio further to reduce the garbage collection pauses. In other cases you may want to increase the compaction ratio to keep heap fragmentation in control. There are several ways to adjust the compaction:

- Setting the Compaction Ratio
- Setting the Compact Set Limit
- Turning Off Compaction
- Using Full Compaction

Setting the Compaction Ratio

Command line option: `-XXcompactRatio:<percentage>`

Setting a static compaction ratio will force the JVM to compact a specified percentage of the heap at each old collection. This disables the heuristics for selecting a dynamic compaction ratio that depends on the heap layout. The compact ratio can be defined to a static percentage of the heap using the command line option `-XXcompactRatio:<percentage>`. For example:

```
java -XXcompactRatio:1 MyApplication
```

This starts up the JVM with a static compact ratio of about 1% of the heap.

For more information, see the documentation on `-XXcompactRatio`.

Use this option if you need to force the JVM to use a smaller or larger compaction ratio than it would select by default. You can monitor the compaction ratio in `-Xverbose:memory=debug` outputs and JRA recordings. A high

compaction ratio keeps down the fragmentation on the heap but increases the compaction pause times.

Setting the Compact Set Limit

Command line option: `-XXcompactSetLimit:<references>`

When compaction has moved objects, the references to these objects must be updated. The garbage collector does this before the Java threads are allowed to run again, which increases the garbage collection pause proportionally to the number of references that have been updated. The compact set limit defines how many references there may be from objects outside the compaction area to objects within the compaction area, thus limiting a portion of the compaction pause. If, during a garbage collection, the number of references to the chosen compaction area exceeds the compact set limit, the compaction will be canceled.

The compact set limit depends on the garbage collection mode used, and will for some modes adjust dynamically in runtime. You can set a static compact set limit by using the command line option `-XXcompactSetLimit:<references>`, where "`references`" specifies the maximum number of references to objects within the compaction area. For example:

```
java -XXcompactSetLimit:20000 MyApplication
```

This starts up the JVM with a compact set limit of 20000 references.

For more information, see the documentation for `-XXcompactSetLimit`.

Use this option to increase the compact set limit if too many compactions are canceled (aborted), or to decrease the limit if the compaction pause times are too long. You can monitor the compaction behavior in `-Xverbose:memory=debug` outputs and JRA recordings, and compaction pause times in `-Xverbose:gcpause=debug` outputs and JRA recordings.

Note: `-XXcompactSetLimit` has no effect when the `deterministic` or `pausetime` garbage collection modes are used, as these garbage collector modes use other heuristics for adjusting the compaction pausetimes.

Turning Off Compaction

Command line option: `-XXnoCompaction`

Very few applications survive in the long run without any compaction at all, but for those that do you can turn off the compaction entirely.

To turn off compaction entirely, use the command line option `-XXnoCompaction`, for example:

```
java -XXnoCompaction MyApplication
```

For more information, see the documentation for `-XXnoCompaction`.

Using Full Compaction

Command line option: **`-XXfullCompaction`**

Some applications are not sensitive to garbage collection pauses or perform old collections very infrequently. For these applications you may want to try running full compaction, as this maximizes the object allocation performance between the garbage collections. Note however that a full compaction of a large heap with a lot of objects may take several seconds to perform.

To turn on full compaction, use the command line option `-XXfullCompaction`, for example:

```
java -XXfullCompaction MyApplication
```

For more information, see the documentation for `-XXfullCompaction`.

Optimizing Memory Allocation Performance

Apart from optimizing the garbage collection to clear space for object allocation, you can tune the object allocation itself to maximize the application throughput.

Setting the Thread Local Area Size

Command line options: **-XXtlaSize:min=<size>,preferred=<size> -XXlargeObjectLimit:<size> -XXminBlockSize:<size>**

The thread local area (TLA) is a chunk of free space reserved on the heap or in the nursery and given to a thread for its exclusive use. A thread can allocate small objects in its own TLA without synchronizing with other threads. Objects allocated in a TLA are however not thread local. They can be accessed by any thread and will be garbage collected globally. When the TLA gets full the thread simply requests a new TLA.

The thread local area size influences the allocation speed, but can also have an impact on garbage collection frequency. A large TLA size allows each thread to allocate a lot of objects before requesting a new TLA, and in JRockit JVM R27.2 and later it also allows the thread to allocate larger objects in the thread local area. On the other hand, a large TLA size prevents small chunks of free memory from being used for object allocation, which increases the impact of fragmentation. In JRockit JVM R27.1 and later, the TLA size is dynamic depending on the size of the available chunks of free space, and varies between a minimum and a preferred size.

Increasing the preferred TLA size is beneficial for applications where each thread allocates a lot of objects. When a two-generational garbage collection strategy is used, a large minimum and preferred TLA size will also allow larger objects to be allocated in the nursery. Note however that the preferred TLA size should always be less than about 5% of the nursery size.

Increasing the minimum TLA size may improve garbage collection times slightly, as the garbage collector can ignore any free chunks that are smaller than the minimum TLA size.

Decreasing the preferred TLA size is beneficial for applications where each thread allocates only a few objects before it is terminated, so that a larger TLA wouldn't ever become full. A small preferred TLA size is also beneficial for applications with very many threads, where the threads don't have time to fill their TLAs before a garbage collection is performed.

Decreasing the minimum TLA size lessens the impact of fragmentation.

A common setting for the TLA size is a minimum TLA size of 2-4 kB and a preferred TLA size of 16-256 kB.

To adjust the TLA size, you can use the command line option `-XXtlaSize:min=<size>,preferred=<size>`. For example:

```
java -XXtlaSize:min=1k,preferred=512k MyApplication
```

This starts up the JVM with a minimum TLA size of 1 kB and a preferred TLA size of 512 kB.

For more information and default values, see the documentation on `-XXtlaSize`.

Note: If you are using JRockit JVM R27.1 or older and want to adjust the TLA size, you should set `-XXlargeObjectLimit:<size>` and `-XXminBlockSize:<size>` to the same value as the minimum TLA size.

Note: If you are using the Oracle JRockit JVM R27.0 or older the minimum and preferred TLA size will always be the same value. The syntax for setting the TLA size is `-XXtlaSize:<size>`.

back to top previous next