Oracle® JRockit JVM

Command Line Reference

R27.6

June 2008

ORACLE

Oracle JRockit JVM Command Line Reference, R27.6

Contents

1. Introduction

Introduction to Command-line Options 1-1

Standard Options 1-2

JRockit JVM Non-standard Options 1-3

Other JRockit JVM Start-up Commands 1-3

Case-sensitivity with Command-Line Options 1-4

Introduction to System Properties 1-4

2. -X Command-line Options

-Xbootclasspath 2-2

Operation 2-2

Flags or Other Options Affected 2-2

Exceptions 2-2

-Xbootclasspath/a 2-3

Operation 2-3

Flags or Other Options Affected 2-3

Exceptions 2-3

-Xbootclasspath/p 2-4

Operation 2-4

Flags or Other Options Affected 2-4

Exceptions 2-4

-Xcheck:jni 2-5

Operation 2-5
Flags or Other Options Affected 2-5
None 2-5
Exceptions 2-5
-XclearType 2-6
Operation 2-6
Default Value 2-6
Flags or Other Options Affected 2-7
Exceptions 2-7
-Xdebug 2-8
Operation 2-8
Flags or Other Options Affected 2-8
None 2-8
Exceptions 2-8
-Xgc 2-9
Operation 2-9
Default Value 2-11
Flags or Other Options Affected 2-11
Exceptions 2-12
-XgcPause 2-13
Operation 2-13
Flags or Other Options Affected 2-13
Exceptions 2-13
-XgcPrio 2-14
Operation 2-14
Default 2-15
Flags or Other Options Affected 2-15
Exceptions 2-16

-XgcReport 2-17
Operation 2-17
Flags or Other Options Affected 2-18
Exceptions 2-18
-XlargePages 2-19
Operation 2-19
How to Configure Large Pages 2-19
Failure to Acquire Large Pages 2-20
Default Values 2-20
Other Options or Flags Affected 2-20
Exceptions 2-20
-Xmanagement 2-21
Operation 2-21
Default Value 2-23
R26.4 and Earlier 2-23
5.0 R27.1 2-23
Flags or Other Options Affected 2-23
Exceptions 2-23
-Xms 2-24
Operation 2-24
Default Values 2-24
Flags or Other Options Affected 2-24
Exceptions and Recommendations 2-25
-Xmx 2-26
Operation 2-26
Known Issue for Linux Users 2-26
Default Value 2-27
Flags or Other Options Affected 2-27

Exceptions 2-27
-XnoClassGC 2-28
Operation 2-28
Flags or Other Options Affected 2-28
Exceptions 2-28
-Xnohup 2-29
Operation 2-29
Flags or Other Options Affected 2-29
Exceptions 2-29
-XnoOpt 2-30
Operation 2-30
Default Value 2-30
Flags or Other Options Affected 2-30
Exceptions 2-30
-Xns 2-31
Operation 2-31
Default Value 2-31
Flags or Other Options Affected 2-32
Exceptions 2-32
-XpauseTarget 2-33
Operation 2-33
Default Value 2-33
Flags or Other Options Affected 2-33
Exceptions 2-34
-Xrs 2-35
Operation 2-35
Flags or Other Options Affected 2-35
Exceptions 2-35

-Xrunjdwp 2-36
Operation 2-36
Flags or Other Options Affected 2-39
Exceptions 2-39
-Xss 2-40
Operation 2-40
Default Value 2-40
Flags or Other Options Affected 2-40
Exceptions 2-40
-XstrictFP 2-41
Operation 2-41
Flags or Other Options Affected 2-41
Exceptions 2-41
-Xverbose 2-42
Operation 2-42
Log Levels 2-53
Other Flags and Options Affected 2-53
Exceptions 2-53
-XverboseDecorations 2-54
Operation 2-54
Default Value 2-55
Flags or Other Options Affected 2-55
Exceptions 2-55
-XverboseLog 2-56
Operation 2-56
Flags or Other Options Affected 2-56
Exceptions 2-56
-XverboseTimeStamp 2-57

Operation 2-57
Flags or Other Options Affected 2-57
Exceptions 2-57
-Xverify 2-58
Operation 2-58
Default Value 2-58
Other Flags or Options Affected 2-58
Exceptions 2-58

3. -XX Command-line Options

-XXaggressive 3-2
Operation 3-2
Default Value 3-2
Flags or Options Affected 3-3
Exceptions 3-3
-XXallocClearChunks 3-4
Operation 3-4
Default Value 3-4
Flags or Other Options Affected 3-4
Exceptions 3-4
-XXallocClearChunkSize 3-5
Operation 3-5
Default Value 3-5
Flags or Other Options Affected 3-5
Exceptions 3-5
-XXallocPrefetch 3-6
Operation 3-6
Default Value 3-6

Other Flags and Options Affected 3-6
Exceptions 3-6
-XXallocRedoPrefetch 3-7
Operation 3-7
Default Value 3-7
Other Flags and Options Affected 3-7
Exceptions 3-7
-XXcallProfiling 3-8
Operation 3-8
Default Value 3-8
Other Flags and Options Affected 3-8
Exceptions 3-8
-XXcompactRatio 3-9
Operation 3-9
Default Value 3-9
Flags or Other Options Affected 3-9
Exceptions 3-10
-XXcompactSetLimit 3-11
Operation 3-11
Default Values 3-11
Flags or Other Options Affected 3-11
Exceptions 3-12
-XXcompactSetLimitPerObject 3-13
Operation 3-13
Default Value 3-13
Flags or Other Options Affected 3-13
Exceptions 3-13
-XXcompressedRefs 3-14

Operation 3-14
Default Value 3-14
Flags or Other Options Affected 3-14
Exceptions 3-14
-XXdisableFatSpin 3-16
Operation 3-16
Flags or Other Options Affected 3-16
Exceptions 3-16
-XXdisableGCHeuristics 3-17
Operation 3-17
Default Value 3-17
Flags or Other Options Affected 3-17
Exceptions 3-17
-XXdumpFullState 3-18
Operation 3-18
Flags or Other Options Affected 3-18
Exceptions 3-18
-XXdumpSize 3-19
Operation 3-19
Flags or Other Options Affected 3-19
Exceptions 3-19
-XXexitOnOutOfMemory 3-20
Operation 3-20
Flags or Other Options Affected 3-20
Exceptions 3-20
-XXexternalCompactRatio 3-21
Operation 3-21
Default Value 3-21

Flags or Other Options Affected 3-21
Exceptions 3-21
-XXfullCompaction 3-23
Operation 3-23
Flags or Other Options Affected 3-23
Exceptions 3-23
-XXfullSystemGC 3-24
Operation 3-24
Flags or Other Options Affected 3-24
Exceptions 3-24
-XXgcThreads. 3-25
Operation 3-25
Default Value 3-25
Flags and Other Options Affected 3-25
Exceptions 3-25
-XXgcTrigger 3-26
Operation 3-26
Default Value 3-26
Other Flags and Options Affected 3-26
Exceptions 3-27
-XXheapParts 3-28
Operation 3-28
Default Value 3-28
Flags or Other Options Affected 3-28
Exceptions 3-28
-XXhpm 3-29
Operation 3-29
Other Flags Affected 3-29

Exceptions 3-29
-XXinitialPointerVectorSize 3-30
Operation 3-30
Default Value 3-30
Flags or Other Options Affected 3-30
Exceptions 3-30
-XXinternalCompactRatio 3-31
Operation 3-31
Default Value 3-31
Flags or Other Options Affected 3-31
Exceptions 3-31
-XXjra 3-33
Operation 3-33
Avoid Using Multiple Options 3-36
Flags or Other Options Affected 3-37
Exceptions 3-37
-XXkeepAreaRatio 3-38
Operation 3-38
Default Value 3-38
Flags or Other Options Affected 3-38
Exceptions 3-38
-XXlargeObjectLimit 3-39
Operation 3-39
Default Value 3-39
Flags or Other Options Affected 3-39
Exceptions 3-39
-XXlargePages 3-40
-XXlazyUnlocking 3-41

Operation 3-41
Default Value 3-41
Other Flags and Options Affected 3-41
Exceptions 3-41
-XXmaxPooledPointerVectorSize 3-42
Operation 3-42
Default Value 3-42
Flags or Other Options Affected 3-42
Exceptions 3-42
-XXmme 3-43
Operation 3-43
Default Value 3-43
Flags or Other Options Affected 3-43
Exceptions 3-43
-XXminBlockSize. 3-44
Operation 3-44
Default Value 3-44
Flags or Other Options Affected 3-44
Exceptions 3-44
Default Value 3-44
Flags or Other Options Affected 3-45
Exceptions 3-45
-XXnoCompaction 3-46
Operation 3-46
Flags or Other Options Affected 3-46
Exceptions 3-46
-XXnoJITInline. 3-47
Operation 3-47

Flags and Other Options Affected 3-47
Exceptions 3-47
-XXnoSystemGC 3-48
Operation 3-48
Flags or Other Options Affected 3-48
Exceptions 3-48
-XXoptThreads 3-49
Operation 3-49
Default Value 3-49
Other Options or Flags Affected 3-49
Exceptions 3-49
-XXpointerMatrixLinearSeekDistance 3-50
Operation 3-50
Default Value 3-50
Flags or Other Options Affected 3-50
Exceptions 3-50
-XXprintSystemGC 3-51
Operation 3-51
Flags or Other Options Affected 3-51
Exceptions 3-51
-XXsetGC 3-52
Operation 3-52
Flags or Other Options Affected 3-53
Exceptions 3-53
-XXstaticCompaction 3-54
Operation 3-54
Flags or Other Options Affected 3-54
Exceptions 3-54

-XXthroughputCompaction 3-56
Operation 3-56
Other Flags or Options Affected 3-56
Exceptions 3-56
-XXtlaSize 3-57
Operation 3-57
Default Value 3-58
Flags or Other Options Affected 3-59
Exceptions 3-59
-XXtsf 3-60
Operation 3-60
Default Value 3-60
Flags or Other Options Affected 3-60
Exceptions 3-60
-XXusePointerMatrix 3-61
Operation 3-61
Flags or Other Options Affected 3-61
Exceptions 3-61
-XX:MaximumNurseryPercentage 3-62
Operation 3-62
Default Value 3-62
Exceptions 3-62
-XX:(+|-)UseNewHashFunction 3-63
Operation 3-63
Default Value 3-63
Flags or Other Options Affected 3-63
Exceptions 3-63
-XX:(+|-)UseThreadPriorities 3-64

Operation 3-64
Default Value 3-64
Flags or Other Options Affected 3-64
Exceptions 3-64

4. Oracle JRockit JVM System Properties

java.vendor 4-3
java.vendor.url 4-4
java.vendor.url.bug 4-5
java.version 4-6
java.runtime.version 4-7
java.vm.name 4-8
java.vm.vendor 4-9
java.vm.vendor.url 4-10
java.vm.version 4-11
java.vm.specification.version 4-12
java.vm.specification.vendor 4-13
java.vm.specification.name 4-14
os.name 4-15
os.arch 4-16
os.version 4-17

CHAPTER 1

Introduction

Welcome to the *Oracle JRockit JVM Command-Line Reference*. This document lists and describes how to use the command-line options and system properties valid for use with Oracle JRockit JVM.

This Introduction includes information on the following subjects:

- Introduction to Command-line Options
- Introduction to System Properties

Introduction to Command-line Options

Command-line options, also called start-up commands or start-up options, are self-descripting tags that you enter either at the command line or include in start-up scripts for applications running on a JVM. These options are used to override the JVM's default settings and otherwise define to the JVM how you want your application to run; for example, you can use the command-line option `-Xmx` to set the maximum heap size or use `-XXnoCompaction` to disable heap compaction after a garbage collection.

Command-line options can be either standard—that is, valid for any JVM regardless of manufacturer—or non-standard—particular to a specific brand of JVM. The options described in this reference manual are non-standard and apply only to JRockit JVM.

Standard Options

While a large volume of standard options for JVMs has been created, JRockit JVM does not recognize all of them. The standard options JRockit JVM accepts are listed in Table 1-1:

Table 1-1 Standard Options fro JVMs

Option (Alternate Usage)	Description
`-agentlib`	When used with a specified library (-agentlib:<*`libname`*>), loads a native agent library; for example: • `-agentlib:hprof` • `-agentlib:jdwp=help` • `-agentlib:hprof=help` For more information, see JVMTI Agent Command Line Options.
`-agentpath`	When uses with a full pathname (`-agentpath:`<*`path-to-agent`*>), loads a native agent library by full pathname. For more information, see JVMTI Agent Command Line Options.
`-client`	Selects the JRockit JVM Client JVM.
`-javaagent`	Loads a Java programming language agent, see java.lang.instrument.
`-jrockit`	Selects the Oracle JRockit server JVM. This is equivalent to `-server` and is the default.
`-version`	Displays version information and then exits the application.
`-showversion`	Displays version information and continues the application run.
`-verbose:<area>[,<area>]`	For more information on how this option works, see `-Xverbose`.
`-cp or -classpath`	Specifies a list of directories, JAR archives, and ZIP archives to search for class files. Class path entries are separated by semicolons (;) in Windows or colons (:) in Linux. Specifying `-classpath` or `-cp` overrides any setting of the `CLASSPATH` environment variable.
`-ea (-enableassertions)`	Enables assertions, which are disabled by default. Depending upon the arguments included with the option, `-ea (-enableassertions)` will either simply enable assertions, enable assertions in the specified package and any subpackages, enable assertions in the unnamed package in the current working directory, or enable assertions in the specified class.

Table 1-1 Standard Options fro JVMs

Option (Alternate Usage)	Description
`-da (-disableassertions)`	Disables assertions. Depending upon the arguments included with the option, `-da` (`-disableassertions`) will either simply disable assertions, disable assertions in the specified package and any subpackages, disable assertions in the unnamed package in the current working directory, or disable assertions in the specified class.
`-esa (-enablesystemassertions)`	Enable asserts in all system classes by setting the default assertion status for system classes to true.
`-dsa (-disablesystemassertions)`	Disables asserts in all system classes.

For documentation on these standard command-line options, please refer to Standard Options in the Sun Microsystems document Java - The Java Application Launcher at:

`http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/java.html#standard`

JRockit JVM Non-standard Options

JRockit JVM uses a set of non-standard options to control JVM behavior. Since these options are non-standard, they will not work with other JVMs. If you attempt to use them with another JVM, you will either receive results that can be erroneous or create an error condition and receive no results at all. Conversely, JRockit JVM will not recognize another JVM's non-standard options.

JRockit JVM's non-standard options are divided into two groups:

- -X Command-line Options, which are the most commonly used non-standard options.
- -XX Command-line Options, which are often experimental options that have specific system requirements for their implementation.

Since all of the options described in this manual are non-standard, they are subject to change or deprecation at any time.

Other JRockit JVM Start-up Commands

Occasionally, you might encounter JRockit JVM internal properties set with the `-D` option. For example, you might see something that looks like this:

```
-Djrockit.lockprofiling=true
```

The `-D` option is used to set properties and their associated values, thus sending those parameters to Java programs. In the Oracle JRockit JVM, some of those parameters are read by the JVM and change how the JVM works. Because these `-D` properties are for internal use, they are not discussed in this reference manual; however, as their number and efficacy increase, such documentation might be added.

Case-sensitivity with Command-Line Options

As of Oracle JRockit JDK 5.0 R25, command-line options are no longer case-sensitive unless explicitly stated; however, for this guide, to ensure readability of the commands, "camel notation" is used to spell out a command (for example, `-XgcPrio` or `-XXnoCompaction`). Although you don't have to use camel notation, if it makes the commands easier to read in the context of your code, you can still do so.

Introduction to System Properties

System properties define traits or attributes of the current working environment. When the Java application first starts up, the system properties are initialized with information about the runtime environment, including information about the current user, the current version of the Java runtime, and even the product vendor's bug report URL.

CHAPTER 2

-X Command-line Options

Non-standard, or `-X`, command line options are options that are exclusive to Oracle JRockit JVM that change the behavior of JRockit JVM to better suit the needs of different Java applications. These options are all preceded by `-X` and will not work on other JVMs (conversely, the non-standard options used by other JVMs won't work with JRockit JVM).

Note: Since these options are non-standard, they are subject to change at any time.

This chapter is the complete reference to all `-X` startup flags that can be set in JRockit JVM. Each option is listed in alphabetical order.

-Xbootclasspath

This option specifies a semicolon-separated list of directories, `JAR` archives, and `ZIP` archives to search for boot class files. These are used in place of the boot class files included in the Java 2 SDK.

Note: Applications that use this option to override a class in `rt.jar` should not be deployed. Doing so would contravene the Java 2 Runtime Environment binary code license.

Operation

Format: `-Xbootclasspath <directories and zips/jars separated by ; (Windows) or : (Linux)>`

Enter this option at startup to create the default classpath for bootstrap classes and resources. This option must be entered in lower case, not camel notation, as shown in the example.

Flags or Other Options Affected

- -Xbootclasspath/a
- -Xbootclasspath/p

Exceptions

None

-Xbootclasspath/a

This option is similar to `-Xbootclasspath` in that it specifies a semicolon-separated path of directories, JAR archives, and ZIP archives; however, this list is appended to the default bootstrap class path.

Operation

Format: `-Xbootclasspath/a <directories and zips/jars separated by ; (Windows) or : (Linux)>`

Enter this option at startup to append a list of directories, JAR archives, and ZIP archives to the default classpath for bootstrap classes and resources. This option must be entered in lower case, not camel notation, as shown in the example.

Flags or Other Options Affected

- -Xbootclasspath
- -Xbootclasspath/p

Exceptions

None

-Xbootclasspath/p

This option is similar to `-Xbootclasspath` in that it specifies a semicolon-separated path of directories, JAR archives, and ZIP archives; however, this list is prepended to the default bootstrap class path.

Operation

Format: `-Xbootclasspath/p <directories and zips/jars separated by ; (Windows) or : (Linux)>`

Enter this option at startup to prepend a list of directories, `JAR` archives, and `ZIP` archives to the default classpath for bootstrap classes and resources. This option must be entered in lower case, not camel notation, as shown in the example.

Flags or Other Options Affected

- -Xbootclasspath
- -Xbootclasspath/a

Exceptions

None

-Xcheck:jni

Enables additional checks for JNI functions.

Operation

Format: `-Xcheck:jni`

Include this option at startup.

Flags or Other Options Affected

None

Exceptions

None

-XclearType

Deprecated

Use this option to define when the memory occupied by an object that has been garbage collected will be cleared. You can set objects clearing at allocation time, garbage collection time, or when a thread-local area is allocated.

Note: This option is deprecated as of the JRockit JVM R25. If you use it with that or any subsequent releases, the JVM will accept the option without throwing an exception, but nothing will happen.

The following information for `-XclearType` applies only to the JRockit JVM R24 and earlier.

Operation

Format: `-XclearType:<param>`

Set the desired parameter (`<param>`) as described in Table 2-1 to define the when object clearing will occur.

Table 2-1 Valid Parameters for -XclearType

<param>	Description
`alloc`	Starts clearing objects at allocation time.
`gc`	Starts clearing objects when the garbage collection is running.
`local`	Starts clearing objects when a thread-local area is allocated.

Default Value

If `-XclearType` is not set, object clearing will default to the following:

- IA32 systems: `alloc`
- IA64 systems:
  - JRockit JVM R24: `local`
  - JRockit JVM R23: `gc`

Flags or Other Options Affected

None

Exceptions

None

-Xdebug

`-Xdebug` enables debugging capabilities in the JVM which are used by the Java Virtual Machine Tools Interface (JVMTI). JVMTI is a low-level debugging interface used by debuggers and profiling tools. With it, you can inspect the state and control the execution of applications running in the JVM.

The subset of JVMTI that is most typically used by profilers is always available. However, the functionality used by debuggers to be able to step through the code and set breakpoints has some overhead associated with it and is not always available. To enable this functionality you must use the `-Xdebug` option.

WARNING: When running with -Xdebug the JVM is not running at its full speed. Thus, the option should not be used for applications when running in production environments.

Operation

Format: `-Xdebug`

Include this option at startup.

For Example:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n myApp
```

Flags or Other Options Affected

None

Exceptions

None

-Xgc

Use `-Xgc` to set a static garbage collector. The static garbage collectors are classified as in Table 2-2.

Table 2-2 Static Garbage Collectors

	Single-Spaced	Generational
Concurrent	Single-spaced concurrent `-Xgc:singlecon`	Generational concurrent `-Xgc:gencon`
Parallel	Single-spaced parallel `-Xgc:singlepar` (since R27.2) or `-Xgc:parallel`	Generational parallel `-Xgc:genpar` (since R27.2)

Under some circumstances, the performance of these static garbage collectors might meet your needs better than the dynamic garbage collection modes or the default collectors available with the `-server` or `-client` flags. Additionally, if you want to use scripts written for the earlier versions of JRockit JVM that implement these collectors, those scripts will continue to work without requiring any modification—unless they use the generational copy garbage collector, which is no longer available.

Operation

Format: `-Xgc:<gcType>`

Use `-Xgc` with one of the garbage collection types (`<gcType>`) listed in Table 2-3 to get the desired garbage collector algorithm:

Table 2-3 Valid Garbage Collection Types for -Xgc

<gcType>	Description
`singlecon`	Sets a single-spaced (non-generational) concurrent garbage collector. This is a mostly concurrent garbage collector, which means that it will perform most of its garbage collection chores concurrently with the Java application. All objects are maintained in a single space, or "generation." The singlecon garbage collection trades lower application throughput for minimal pause times.
`gencon`	Sets a generational, concurrent garbage collector. With this type of garbage collector, objects are allocated in the young *generation* (the nursery). When the nursery is full, JRockit JVM stops all Java threads and moves the live objects in the young generation to the *old generation*. This is a mostly concurrent garbage collector, which means that it will perform most of its garbage collection chores concurrently with the Java application. The gencon garbage collector is better than the singlecon garbage collector for most applications that allocate a lot of small, short-lived objects. The gencon garbage collection trades minimal pause times for larger heap sizes and lower application throughput.

Table 2-3 Valid Garbage Collection Types for -Xgc

<gcType>	Description
`singlepar` `parallel`	Sets a single-spaced, parallel garbage collector. A parallel garbage collector stops all Java threads when the heap is full and uses every CPU to perform a complete garbage collection of the entire heap. A parallel collector can have longer pause times than concurrent collectors, but it maximizes throughput. Even on single CPU machines, this maximized performance makes parallel the recommended garbage collector, provided that your application can tolerate the longer pause times. The `singlepar` synonym is available in R27.2 and later releases.
`genpar`	Sets a generational garbage collector. With this type of garbage collector, objects are first allocated in the young generation. When the nursery is full, JRockit JVM stops all Java threads and performs a parallel nursery collection, that is, it uses all CPU resources available and moves all live objects in the young generation to the old generation. The old generation collector stops all Java threads when the heap is full and a complete parallel garbage collection is performed. This collector will prioritize throughput to pause times. This collector is generally better than the singlepar garbage collector for applications that allocate a lot of short-lived objects. While it performs more garbage collections than the singlepar collector, the individual pause times of the genpar collector are shorter and it creates less fragmentation in the old generation space. This option is available in R27.2 and later releases.

Default Value

These defaults apply to `-Xgc`:

- `singlecon` is the default garbage collector when JRockit JVM is run in `-client` mode
- The default garbage collector when JRockit JVM is run in `-server` mode is the dynamic garbage collection mode optimized for throughput, and thus not one of the static garbage collectors that you can set with `-Xgc`.

Flags or Other Options Affected

When you specify `-Xgc`, the following options are affected:

- Setting `-XXsetGC` will override `-Xgc` and vice versa. The option specified first on the command line will be ignored.
- Setting `-Xgc` will override part of the effect of `-server` and `-client`.

Exceptions

When using `-Xgc`, be aware of the following exceptions:

- If you set a static garbage collector, you will not be able to fully use the management API; that is, some functions of the API will not be available.
- You cannot use `-Xgc` together with either of the following options:
  - `-XgcPrio`
  - `-XpauseTarget`

-XgcPause

The `-Xgcpause` option prints out the pause times caused by the garbage collector during a run. The pause times are shown during runtime on your screen during the running of the application.

The effect of this option is identical to `-Xverbose:gcpause`.

Operation

Format: `-XgcPause`

Use this option at startup. As pauses are encountered, they will print a report to your screen, as shown in Listing 2-1.

Listing 2-1 Output from -Xgcpause used with the Static Garbage Collector -Xgc:gencon

```
[memory ] old collection phase 0-2 pause time: 100.794255 ms
[memory ] nursery collection pause time: 4.121775 ms
[memory ] nursery collection pause time: 185.137069 ms
[memory ] old collection phase 4-5 pause time: 147.672697 ms
[memory ] (pause includes yc: 142.537 ms, compaction: 1.317 ms, update ref: 1.41
3 ms)
[memory ] nursery collection pause time: 7.075705 ms
[memory ] old collection phase 5 pause time: 0.300176 ms
```

Flags or Other Options Affected

None

Exceptions

None

-XgcPrio

`-XgcPrio` sets a dynamic garbage collection mode. This garbage collector combines all types of garbage collection heuristics and optimizes the performance accordingly. When running this garbage collector, you only need to determine whether your application responds best to optimal memory throughput during collection or minimized pause times. The dynamic garbage collector will then adapt its choice of collector type, in runtime, to what is suiting your application the best.

Note: This command line option is supported with JRockit JDK 1.4.2_10 R26.2 and later versions, as well as all versions of JRockit JDK 5.0 and JRockit JDK 6.

Operation

Format: `-XgcPrio:<gcType>`

Combine `-XgcPrio` with one of the garbage collection types (`<gcType>`) described in Table 2-4:

Table 2-4 Garbage Collection Types Valid for -XgcPrio

<gcType>	Description
`throughput`	The garbage collector is optimized for application throughput. This means that the garbage collector works as effectively as possible, giving as much CPU resources to the Java threads as possible. This may, however, cause non-deterministic pauses when the garbage collector stops all Java threads for garbage collection.The throughput priority should be used when non-deterministic pauses do not impact the application's behavior.

Table 2-4 Garbage Collection Types Valid for -XgcPrio

<gcType>	Description
`pausetime`	The garbage collector is optimized for short pauses. This means that the garbage collection will work concurrently with the Java application when necessary, in order to avoid pausing the Java threads. This inflicts a slight performance overhead to the application, as the concurrent garbage collector demands more system resources (CPU time and memory) than the parallel garbage collector that is used for optimal throughput. The target pause time is by default 200 ms. To change the default pause target, see -XpauseTarget.
deterministic	The garbage collector is optimized for very short and deterministic pause times.The garbage collector will aim on keeping the garbage collection pauses below a 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`.

Default

- The default garbage collector in -server mode is the dynamic garbage collection mode optimized for `throughput` (`-Xgcprio:throughput`).
- The default garbage collector in -client mode is static, using the single generational concurrent garbage collection strategy (`-Xgc:singlecon`)

Flags or Other Options Affected

Use of -XgcPrio will affect certain options as described here:

- Setting `-XgcPrio` will override part of the effect of `-server` and `-client`.

- If you have set `-Xns` it overrides the dynamic nursery sizing; see `-Xns`.
- You can use `-XpauseTarget` to set pause times for `-XgcPrio:pausetime` and `XgcPrio:deterministic`.

Exceptions

You cannot combine the dynamic garbage collector with a static garbage collector set with `-Xgc` or `-XXsetGC`.

If you set `-XXdisableGCHeuristics`, then there will be no change of garbage collection strategy as a result of the `-XgcPrio` option.

-XgcReport

The `-XgcReport` option generates an end-of-run report that shows garbage collection statistics. You can use this report to determine if you're using the most effective garbage collector for your application or not.

The report divides the statistics into "young collections" and "old collections", and for each of the types the following information is printed:

- Number of collections: The total number of garbage collections of this type during the run.
- Total promoted: The total number of objects and amount of bytes promoted from young space to old space by this type of garbage collections during the run.
- Max promoted: The maximum number of objects and amount of bytes promoted by any single garbage collection of this type during the run.
- Total GC time: The total time spent in this type of garbage collections during the run. For concurrent garbage collections the total garbage collection time and the total garbage collection pause time will differ.
- Mean GC time: The average time spent in a single garbage collection of this type during the run. For concurrent garbage collections the garbage collection time and the garbage collection pause time will differ.
- Maximum GC pauses: The three longest garbage collection pauses caused by this type of garbage collection during the run.

The effect of this option is identical to `-Xverbose:gcreport`.

Operation

Format: `-XgcReport`

As shown in Listing 2-2, the `-Xgcreport` shows a detailed profile of collections on both the young generation and the old generation.

Listing 2-2 Dynamic Xgcprio:pausetime + Xgcreport (at the end of the application run):

```
[memory ] Memory usage report
[memory ]
```

```
[memory ] young collections
[memory ]     number of collections = 50
[memory ]     total promoted =        4628936 (size 260614072)
[memory ]     max promoted =          108102 (size 6083128)
[memory ]     total GC time =         4.381 s
[memory ]     mean GC time =          87.623 ms
[memory ]     maximum GC Pauses =      125.264 , 153.122, 244.682 ms

[memory ]

[memory ] old collections
[memory ]     number of collections = 6
[memory ]     total promoted =        0 (size 0)
[memory ]     max promoted =          0 (size 0)
[memory ]     total GC time =         0.701 s (pause 0.701 s)
[memory ]     mean GC time =          116.833 ms (pause 116.830 ms)
[memory ]     maximum GC Pauses =      134.081 , 190.973, 215.381 ms

[memory ]

[memory ]     number of parallel mark phases    = 6
[memory ]     number of parallel sweep phases   = 4
[memory ]     number of concurrent sweep phases = 2
Mon Nov  1 16:50:34 WEST 2004
```

Flags or Other Options Affected

None

Exceptions

None

-XlargePages

This option tells the JRockit JVM to use large pages, if they are available, for the Java heap and other areas in the JVM. Large pages allow your application to more effectively use the translation look-aside buffer (TLB) in the processor.

Note: While this option duplicates the functionality of `-XXlargePages`, this is the preferred option for enabling large pages.

Operation

Format: `-XlargePages`

Windows, Linux and Solaris all support multiple page sizes on x86, IPF and SPARC architectures. x86 supports 4 KB and 4 MB (2 KB and 2 MB in PAE mode), while IPF and SPARC support a wider range of different sizes, from 4 KB to 256M, depending on model.

Format: `-XlargePages:exitOnFailure=true`

By default the JVM will continue running without large pages if large pages cannot be acquired when `-XlargePages` is enabled. Use this extended option to override this behavior and force the JVM to exit if enough large pages can't be acquired. This extended option is supported in JRockit JVM R27.5 and later releases.

How to Configure Large Pages

If you use this option, you will need to configure large pages on you machine. To do so, use one of the following procedures.

On Linux

1. Reserve memory to be used for large pages by executing the following command:

   `echo` *nn* `> /proc/sys/vm/nr_hugepages`

   Where *nn* is the number of desired pages.

   You should do this step as soon as possible after the machine has been started since ongoing memory usage creates fragmentation and Linux might be unable to allocate the number of specified pages.

Note: The following steps might need to be completed by your system administrator, unless you have administrative privileges.

2. To enable the JRockit JVM to allocate large pages, make a `hugetblfs` file system available by using this command:

   ```
   mount -t hugetlbfs nodev /mnt/hugepages
   ```

3. Grant the user executing the Java application read and write permission to the file system. You can do this with either the `mount` command or with the `chmod` and `chown` commands.

For a more thorough description of large pages on Linux, read the file `vm/hugetlbpage.txt` available in the documentation for the Linux kernel.

On Windows

1. As Administrator, give the user who will run the application the permission to lock pages in memory by opening the Start menu and selecting:

   Control Panel>**Administrative**>**Tools**>**Local Security Policy**>**Local Policies**>**User Rights**>**Assignments**>**Lock pages in memory**.

2. Select **Lock pages in memory**.

3. Make enough free consecutive memory available by either logging off your computer or rebooting it.

On Solaris

Nothing has to be configured in the O/S to enable an application to use large pages.

Failure to Acquire Large Pages

If the JRockit JVM fails to acquire large pages, it will print a warning and continue; for example:

```
[WARN ] Unable to acquire large pages for the heap, using normal pages
```

Default Values

`-XXlargePages` is disabled by default.

Other Options or Flags Affected

None

Exceptions

None

-Xmanagement

This option starts the JRockit JVM concurrently with the management server and allows you to either enable and configure or explicitly disable security features such as SSL encryption and authentication.

Operation

Format: `-Xmanagement[:<argument1>=<value1>[,<argument2>=<value2>]]`

where *argumentn* and *valuen* is as defined in Table 2-5.

Table 2-5 -Xmanagement Arguments

Argument	Description
`autodiscovery=<true|false>`	Enables or disables autodiscovery, which allows Oracle JRockit Mission Control to automatically discover running JRockit JVM instances through the multicast-based JRockit Discovery Protocol. Valid with JRockit JVM 5.0.
`ssl=<true|false>`	Enables or disables SSL encryption. Valid with JRockit JVM 5.0.
`authenticate=<true|false>`	Enables or disables authentication. Valid with JRockit JVM 5.0.
`port=<portNumber>`	Identifies the port that the management server will open for remote access. Valid with JRockit JVM 5.0.
`class=`	This option loads the class and causes its empty constructor to be called early in JVM startup. From the constructor, a new thread is then started, from which your management client is run. Further arguments cannot be given to `-Xmanagement` after the `class` argument.

For Example:

- `java -Xmanagement:ssl=false,authenticate=false myApplication`

Disables SSL encryption and authentication.

- `java -Xmanagement:autodiscovery=true` *`myApplication`*

  Enables autodiscovery.

- `java -Xmanagagement:port=1234` *`myApplication`*

  Tells the management server to open port 1234.

The implementation of this option in JRockit JVM R27.1.0 represents a change from previous versions of the product. In past versions, this option enabled the management server running:

- A JMX agent (for JRockit JVM versions based on J2SE 5.0).
- Oracle's proprietary RMP protocol (for JRockit JVM versions based on J2SE 1.4.2).

The original RMP implementation had no security features. JRockit JVM 5.0 contains new security features—authentication and SSL encryption—that were introduced in J2SE 5.0, but similar security features were not enabled in the JRockit JVM 1.4.2.

Due to the security risks and the mission-critical nature of most JRockit JVM deployments, the new default behavior of the JRockit JVM requires that you either disable security explicitly or configure and enable security. If you don't take these steps, the management server will not open a port for remote access and may cause the JVM startup to halt with an error message concerning the security configuration.

Specifying `-Xmanagement` also enables a local in-memory agent to improve the user experience from a developer perspective. For example, a developer running a WLS instance on JRockit JVM on a one machine can specify `–Xmanagement` to enable the local in-memory agent to connect to it from an Oracle JRockit Mission Control Client on another machine. On the other hand, the developer would not need to specify `–Xmanagement` to get local access from JRockit Mission Control: the in-memory agent is always accessible locally. Thus, if you have a number of JRockit JVM instances running on your machine and you start a JRockit Mission Control Client, it will automatically discover and allow access to those JVMs. Security is enforced by only allowing this type of local access if the JRockit JVM instance and the JRockit Mission Control Client are being run by the same user. Note that this will only work if the monitored JRockit JVM is R27.1 and later.

To enable the management agent without security you must now specify that SSL and authentication should be disabled. Also, the JMX server port must now be specified explicitly with the JRockit JVM 5.0.

For maximum usability, you should also enable the autodiscovery mechanism, which allows JRockit Mission Control to automatically discover running JRockit JVM instances through the

multicast based JRockit Discovery Protocol. Note that this will normally only work on the local subnet.

To limit the number of RMP connections running at the same time, you can set `-Djrockit.managementserver.maxconnect`.

Default Value

The default value is based on the version of the JRockit JVM you are running:

R26.4 and Earlier

J2SE 5.0:

- RJMX server is enabled on port 7091.
- SSL and authentication off by default.

J2SE 1.4.2:

- RMP server is enabled on port 7090
- SSL and authentication are not available.

5.0 R27.1

- Local agent is enabled.
- Remote agent is enabled on if security is enabled or explicitly disabled.

Flags or Other Options Affected

None

Exceptions

None

-Xms

The `-Xms` option sets the initial and minimum Java heap size. The Java heap (the "heap") is the part of the memory where blocks of memory are allocated to objects and freed during garbage collection.

Note: `-Xms` *does not* limit the total amount of memory that the JVM can use.

Operation

Format: `-Xms<size>[g|G|m|M|k|K]`

Combine `-Xms` with a memory value and add a unit.

For Example:

`java` **`-Xms:64m`** *myApp*

sets the initial and minimum java heap to 64 MB.

If you do not add a unit, you will get the exact value you state; for example, 64 will be interpreted as 64 bytes, not 64 megabytes or 64 kilobytes.

For best performance, set `-Xms` to the same size as the maximum heap size, for example:

`java -Xgcprio:throughput -Xmx:64m` **`-Xms:64m`** *myApp*

Default Values

If you do not set this, the minimum Java heap size defaults to (depending on which mode you are running):

- **`-server` mode:** 25% of the amount of free physical memory in the system, up to 64 MB and at least 8 MB
- **`-client` mode:** 25% of the amount of free physical memory in the system, up to 16 MB and at least 8 MB
- If the nursery size is set with `-Xns`, the default initial heap size will be scaled up to at least twice the nursery size.

Flags or Other Options Affected

None

Exceptions and Recommendations

The initial Java heap cannot be set to a smaller value than 8 MB, which is the minimum Java heap size. If you attempt to set it to a smaller value, JRockit JVM defaults to 8 MB.

The `-Xms` value cannot exceed the value set for `-Xmx` (the maximum Java heap size).

-Xmx

This option sets the maximum Java heap size. The Java heap (the "heap") is the part of the memory where blocks of memory are allocated to objects and freed during garbage collection. Depending upon the kind of operating system you are running, the maximum value you can set for the Java heap can vary.

Note: `-Xmx` *does not* limit the total amount of memory that the JVM can use.

Operation

Format: `-Xmx<size>[g|G|m|M|k|K]`

Combine `-Xmx` with a memory value

For Example:

`java` **`-Xmx:1g`** *myApp*

sets the maximum java heap to 1 gigabyte.

If you do not add a unit, you will get the exact value you state; for example, 64 will be interpreted as 64 bytes, not 64 megabytes or 64 kilobytes.

The `-Xmx` option and `-Xms` option in combination are used to limit the Java heap size. The Java heap can never grow larger than `-Xmx`. Also, the `-Xms` value can be used as "minimum heap size" to set a fixed heap size by setting `-Xms = -Xmx` when, for example, you want to run benchmark tests.

Known Issue for Linux Users

The JRockit JVM R26.0.0 on Linux IA32 can experience problems setting up memory for object allocation. When this happens, you will receive the following message:

```
[JRockit] ERROR: Fatal error in JRockit during memory setup phase.
Try to reduce the heap size using -Xmx:<size>m, i.e. "-Xmx:16m". Could
not create the Java virtual machine.
```

and JRockit JVM will be exited.

The workaround for this situation is to try different `-Xmx` values until you find a heap size that is setup correct.

Note: This known issue is valid for R26.0.0.

Default Value

If you do not set this, the maximum java heap size depends on the platform and the amount of memory in the system as described in Table 2-6.

Table 2-6 Default Maximum Heap Sizes

Release	Platform	Default Maximum Heap Size
R27.2 and older	Windows	75% of total physical memory up to 1 GB
R27.2 and older	Linux, Solaris	50% of available physical memory up to 1 GB
R27.3 and newer	Windows on a 64 bit platform	75% of total physical memory up to 2 GB
R27.3 and newer	Linux or Solaris on a 64 bit platform	50% of available physical memory up to 2 GB
R72.3 and newer	Windows on a 32 bit platform	75% of total physical memory up to 1 GB
R27.3 and newer	Linux on a 32 bit platform	50% of available physical memory up to 1 GB

Flags or Other Options Affected

None.

Exceptions

When using-Xmx, be aware of the following exceptions:

- If both `-Xmx` and `-Xms` are specified the value of `-Xmx` must be larger than or equal to that of `-Xms`.
- If both `-Xmx` and `-Xns` are specified the value of `-Xmx` must be larger than or equal to that of `-Xns`.
- The minimum value for `-Xmx` is 16 MB.

-XnoClassGC

This option disables garbage collection of classes. Using `-XnoClassGC` can save some garbage collection time, which will shorten interruptions during the application run.

Operation

Format: `-XnoClassGC`

When you specify `-XnoClassGC` at startup, the class objects in the application specified by *myApp* will be left untouched during garbage collection and will always be considered live. This can result in more memory being permanently occupied which, if not used carefully, will throw an out of memory exception.

Flags or Other Options Affected

None

Exceptions

None

-Xnohup

This option helps to prevent possible interference when JRockit JVM is running as a service and receives `CTRL_LOGOFF_EVENT` or `SIGHUP`. Upon receiving such events, the VM tries to initiate a shutdown but this shutdown will fail, since the operating system will not actually terminate the process.

Note: `-Xnohup` is similar to the Sun Microsystems command-line option -Xrs, developed for their HotSpot JVM. The JRockit JVM supports both `-Xra` and `-Xnohup`. If you use `-Xra`, you will see the same behavior that you would see with `-Xnohup`.

Operation

Format: `-Xnohup`

If you are running JRockit JVM as a service (for example, the servlet engine for a web server), enter the command at startup to prevent the JVM from watching for or processing `CTRL_LOGOFF_EVENT` or `SIGHUP` events.

Flags or Other Options Affected

None

Exceptions

If you specify `-Xnohup`, be aware of the following:

- Pressing Ctrl-Break to create a thread dump does not work.
- User code is responsible for causing shutdown hooks to run, for example, by calling `System.exit()` when JRockit JVM is to be terminated.

-XnoOpt

This option turns off adaptive optimization. While optimized code generally runs faster than code that hasn't been optimized, occasionally, the time required to optimize code results in undesirable delays processing. `-XnoOpt` lets you avoid these delays by turning off optimization. This option is also helpful when you suspect that a JVM or application problem, such as a system crash or poor startup performance, might be related to optimization. You can turn optimization off and retry your application. If it then runs successfully, you can safely assume that the problem lies with code optimization.

Operation

Format: `-XnoOpt`

Note that if you use `-XnoOpt`, you will continue to compile code that might be inefficient and might have a deleterious affect on application performance.

Default Value

If `-XnoOpt` is not set, the JVM will optimize code as usual.

Flags or Other Options Affected

If you use `-XXnoJITInline`, you must also use `-XnoOpt`. `-XXnoJITInline` only disables inlining the first time a method is compiled. Unless you set `-XnoOpt` as well, methods can still be inlined when code is optimized.

Exceptions

None

-Xns

`-Xns` sets the nursery size. the JRockit JVM uses a nursery when the generational garbage collection model is used, that is, when the dynamic garbage collector has determined that the generational garbage collection model should be used or when the static generational concurrent garbage collector (`-Xgc:gencon`) has been selected. You can also use `-Xns` to set a static nursery size when running a dynamic garbage collector (-XgcPrio).

Operation

Format: `-Xns:<size>[g|G|m|M|k|K]`

Combine `-Xns` with a memory value

For Example:

`java` **`-Xns:10m`** *myApp*

sets the nursery to 10 MB of the heap.

The nursery size value cannot exceed the maximum value set for the heap.

Default Value

Default value depends upon whether you use a dynamic garbage collector (the default garbage collector) or the select a static garbage collector, see Table 2-7

Table 2-7 Default Nursery Sizes

Releases	Options used	Default value
All	-server (default)	Dynamic, since the default garbage collector is dynamic.
All	-client	None, since the default garbage collector in -client mode is single spaced.
All	-Xgc:gencon or -Xgc:genpar together with -client	2 MB
All	-Xgc:gencon	10 MB * the number of hardware threads up to 25% of the heap size
R27.3 and later	-Xgc:genpar	Dynamic

Table 2-7 Default Nursery Sizes

Releases	Options used	Default value
All	-Xgcprio:throughput	Dynamic
All	-Xgcprio:pausetime	Dynamic

Flags or Other Options Affected

None

Exceptions

When using `-Xns`, be aware of the following exceptions:

- You can set `-Xns` only when using a dynamic garbage collector (`-XgcPrio`) or a static generational garbage collector (`-Xgc:gencon` or `-Xgc:genpar`).
- The value must be at least four times the size of `-XXlargeObjectLimit` and it may not be greater than `-Xmx`.

-XpauseTarget

This option sets a pause target for the dynamic garbage collection mode optimizing for short pauses (`-XgcPrio:pausetime`) and the dynamic garbage collection mode optimizing for deterministic pauses (`-XgcPrio:deterministic`). The target value is used as a pause time goal. The target helps the dynamic garbage collector to more precisely configure itself to keep pauses near the target value. Using this option allows you to specify the pause target to be between 1 ms and 5 seconds. If you are using the deterministic garbage collector, you can set values below 200 ms you must have a valid license for `-XgcPrio:deterministic`.

Operation

Format: `-XpauseTarget=<target value>`

Or

`-XpauseTarget=<target value> -Xgcprio:pausetime`

Remember that the target set by this option is considered a soft goal; that is, if specifying the target to 100 ms, the garbage collector will try to tune itself towards a configuration that will make the pauses become as near 100 ms as possible. However, if you have an application and heap configuration that will not be able to meet this target however much the garbage collector is tuned (statically or dynamically), the target will be missed. This option only specified the desired pause times, not the maximum allowed pause time.

Used with care this option will improve pause times, however if used with less care, it might stress the garbage collector to less appreciated behavior.

Default Value

If you are using `-XpauseTarget` with `-XgcPrio:pausetime`, the default setting for the target is 500 ms. If you are using `-XgcPrio:deterministic`, the default value is 30 ms.

Flags or Other Options Affected

Normally, this option requires that you use it with a dynamic pause optimizing garbage collection mode (`-XgcPrio:pausetime` or `-XgcPrio:deterministic)`. If you don't specify a garbage collector, this option will change from the default garbage collector to the pause time optimizing garbage collector (the same collector used when specifying `-XgcPrio:pausetime`).

If you are using Oracle JRockit Real Time, set `-XgcPauseTarget` to below 200 ms, and don't specify a garbage collector, the garbage collector will be set to `-XgcPrio:deterministic`.

Exceptions

When using `-XpauseTarget`, be aware of the following exceptions:

- If you have specified garbage collector other than a dynamic pause-time optimizing garbage collector, the option for pause target cannot be used, as you can't optimize for pause times and be static or throughput-optimizing at the same time.
- If you are usimg the deterministic garbage collector, you can specify pause targets below 200 ms as well.

-Xrs

Note: `-Xrs` is a non-standard option developed by Sun Microsystems for their HotSpot JVM. JRockit JVM continues to support this option; however the JRockit JVM non-standard option `-Xnohup` provides the same functionality.

`-Xrs` reduces usage of operating-system signals by the JVM. If the JVM is run as a service (for example, the servlet engine for a web server), it can receive `CTRL_LOGOFF_EVENT` but should not initiate shutdown since the operating system will not actually terminate the process. To avoid possible interference such as this, the `-Xrs` command-line option does not install a console control handler, implying that it does not watch for or process `CTRL_C_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, or `CTRL_SHUTDOWN_EVENT`.

Operation

Format: `-Xrs`

If you are running JRockit JVM as a service (for example, the servlet engine for a web server), enter the command at startup to prevent the JVM from watching for or processing `CTRL_LOGOFF_EVENT` or `SIGHUP` events.

Flags or Other Options Affected

None

Exceptions

If you specify `-Xnohup`, be aware of the following:

- Pressing Ctrl-Break to create a thread dump does not work.
- User code is responsible for causing shutdown hooks to run, for example, by calling `System.exit()` when JRockit JVM is to be terminated.

-Xrunjdwp

This option loads the JPDA reference implementation of JDWP. This library resides in the target VM and uses JVMDI and JNI to interact with it. It uses a transport and the JDWP protocol to communicate with a separate debugger application.

Operation

Format: `-Xrunjdwp:<name1>[=<value1>],<name2>[=<value2>]...`

The `-Xrunjdwp` option can be further qualified by specifying one of the sub-options listed in Table 2-8.

Table 2-8 -Xrunjdwp Sub-options

Name	Required?	Default Value	Description
help	no	N/A	Prints a brief help message and exits the VM.
transport	yes	none	Name of the transport to use in connecting to debugger application.
server	no	“n”	If “y”, listen for a debugger application to attach; otherwise, attach to the debugger application at the specified `address`. If “y” and no address is specified, choose a transport address at which to listen for a debugger application, and print the address to the standard output stream.
address	yes, if `server=n` no, otherwise	““	Transport address for the connection. If server=n, attempt to attach to debugger application at this address. If server=y, listen for a connection at this address.

Table 2-8 -Xrunjdwp Sub-options

launch	no	none	At completion of JDWP initialization, launch the process given in this string. This option is used in combination with `onthrow` and/or `onuncaught` to provide "Just-In-Time debugging" in which a debugger process is launched when a particular event occurs in this VM. Note that the launched process is not started in its own window. In most cases the launched process should be a small application which in turns launches the debugger application in its own window. The following strings are appended to the string given in this argument (space-delimited). They can aid the launched debugger in establishing a connection with this VM. The resulting string is executed. The value of the `transport` sub-option. The value of the `address` sub-option (or the generated address if one is not given)
onthrow	no	none	Delay initialization of the JDWP library until an exception of the given class is thrown in this VM. The exception class name must be package-qualified.Connection establishment is included in JDWP initialization, so it will not begin until the exception is thrown.
onuncaught	no	"n"	If "y", delay initialization of the JDWP library until an uncaught exception is thrown in this VM. Connection establishment is included in JDWP initialization, so it will not begin until the exception is thrown. See the JDI specification for com.sun.jdi.ExceptionEvent for a definition of uncaught exceptions.
stdalloc	no	"n"	By default, the JDWP reference implementation uses an alternate allocator for its memory allocation. If "y", the standard C runtime library allocator will be used. This option is mainly for testing; use it with care. Deadlocks can occur in this VM if the alternative allocator is disabled.

Table 2-8 -Xrunjdwp Sub-options

strict	no	“n”	If “y”, assume strict JVMDI conformance. This will disable all workarounds to known bugs in JVMDI implementations. This option is mainly for testing and should be used with care.
suspend	no	“y”	If “y”, VMStartEvent has a suspend Policy of SUSPEND_ALL. If “n”, VMStartEvent has a suspend policy of SUSPEND_NONE.

For example:

`java` **`-Xrunjdwp:transport=dt_socket,server=y,address=8000`** *`myApp`*

This command:

- Listens for a socket connection on port 8000.
- Suspends this VM before main class loads (`suspend=y` by default).
- Once the debugger application connects, it can send a JDWP command to resume the VM.

`-Xrunjdwp:transport=dt_shmem,server=y,suspend=n`

This command:

- Chooses an available shared memory transport address and print it to `stdout`.
- Listens for a shared memory connection at that address.
- Allows the VM to begin executing before the debugger application attaches.

`-Xrunjdwp:transport=dt_socket,address=myhost:8000`

This command:

- Attaches to a running debugger application via socket on host myhost at port 8000.
- Suspends this VM before main class loads.

`-Xrunjdwp:transport=dt_shmem,address=mysharedmemory`

This command:

- Attaches to a running debugger application via shared memory at transport address `mysharedmemory`.
- Suspends this VM before main class loads.

```
-Xrunjdwp:transport=dt_socket,server=y,address=8000,onthrow=java.io.IOExceptio
n,launch=/usr/local/bin/debugstub
```

This command:

- Waits for an instance of `java.io.IOException` to be thrown in this VM.
- Suspends the VM (`suspend=y` by default).
- Listens for a socket connection on port 8000.
- Executes the following:

  ```
  /usr/local/bin/debugstub dt_socket myhost:8000
  ```

  This program can launch a debugger process in a separate window which will attach to this VM and begin debugging it.

```
-Xrunjdwp:transport=dt_shmem,server=y,onuncaught=y,launch=d:\bin\debugstub.exe
```

This command:

- Waits for an uncaught exception to be thrown in this VM.
- Suspends the VM.
- Selects a shared memory transport address and listen for a connection at that address.
- Executes the following:

  `d:\bin\debugstub.exe dt_shmem <address>`
  where `<address>` is the selected shared memory address.

  This program can launch a debugger process in a separate window which will attach to this VM and begin debugging it.

Flags or Other Options Affected

None

Exceptions

None

-Xss

`-Xss` sets the thread stack size. Thread stacks are memory areas allocated for each Java thread for their internal use. This is where the thread stores its local execution state.

Operation

Format: `-Xss<size>[g|G|m|M|k|K]`

Combine `-Xss` with a memory value

For Example:

`java` **`-Xss:512k`** *`myApp`*

sets the default stack size to 512 kilobytes.

If you do not add a unit, you will get the exact value you state; for example, 64 will be 64 bytes, not 64 megabytes or 64 kilobytes.

Default Value

`-Xss` default values are platform-specific, as defined in Table 2-9.

Table 2-9 -Xss Default Values

Platform	Default
Windows x86	64 KB
Linux x86	128 KB
Windows IA64	320 KB
Linux IA64	1024 KB
Solaris Sparc	256 KB

Flags or Other Options Affected

None

Exceptions

None

-XstrictFP

This option enables strict floating point arithmetics globally for all methods in all classes. With `-XstrictFP` set, the JVM calculates with more precision, and with a greater range of values than the Java specification requires. When you use `-XstrictFP`, the compiler generates code that adheres strictly to the Java specification to ensure identical results on all platforms. Without `-XstrictFP`, the JVM will not be as strict in enforcing floating point values.

This option is similar to the Java keyword `strictfp`; however, that keyword applies at the class level whereas `-XstrictFP` applies globally. See the Java Language Specification for more details on `strictfp`.

Operation

Format: `-XstrictFP`

Flags or Other Options Affected

None

Exceptions

None

-Xverbose

`-Xverbose` lets JRockit JVM output specific information about the system. The output is by default printed to the standard output for error messages (`stderr`) but you can redirect it to a file by using the -XverboseLog command line option. The information displayed depends on the parameter specified with the option; for example, specifying the parameter `cpuinfo` displays information about your CPU and indicates whether or not the JVM can determine if hyper threading is enabled. Table 2-10 lists the parameters available for the `-Xverbose` option.

Operation

Format: `-Xverbose:<param[=level]>`

Where `param` is one of the parameters described in Table 2-10 and `level` the log level, as described in Log Levels.

For Example:

`java` **`-Xverbose:gcpause=debug`** *`myClass`*

enables pause time sampling and information during a run and logs messages with detailed information of JRockit JVM's behavior.

Note: To use more than one parameter, separate them with a comma; for example:

```
-Xverbose:gc,opt
```

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

`class`	`The names of classes loaded; sample output might look like this:`
	`[INFO ][class ] Initializing bootstrap classes...`
	`[INFO ][class ] created: # 0 java/lang/Object (/localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/rt.jar)`
	`[INFO ][class ] 0 java/lang/Object success (0.45 ms)`
	`[INFO ][class ] created: # 2 java/io/Serializable (/localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/rt.jar)`
	`[INFO ][class ] 2 java/io/Serializable success (0.08 ms)`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

codegen	The names of each method that is being compiled. Verbose output for codegen might look like this:
	`[codegen]  0 : 17.9411 ms`
	`[codegen]  0 68592131  1 java.lang.Object.unlockFatReal_jvmpi (Ljava.lang.Object;Ljava.lang.Thread;I)V: 17.94 ms`
	`[codegen]  1 : 2.0262 ms`
	`[codegen]  0 0  2 java.lang.Object.acquireMonitor(Ljava.lang.Object;II)I: 19.97 ms`
	`[codegen]  2 : 4.4926 ms`
	`[codegen]  0 10  3 java.lang.Object.unlockFat(Ljava.lang.Object;Ljava.lang.Thread;I)V: 24.46 ms`
	`[codegen]  3 : 0.3328 ms`

cpuinfo	Technical information about your CPUs. Verbose output for cpuinfo might look like this:
	`[cpuinfo] Vendor:  GenuineIntel`
	`[cpuinfo] Type:  Original OEM`
	`[cpuinfo] Family:  Pentium 4`
	`[cpuinfo] Brand:  Intel(R) Pentium(R) 4 Mobile CPU 1.60GHz`
	`[cpuinfo] Supports: On-Chip FPU`
	`[cpuinfo] Supports: Virtual Mode Extensions`
	`[cpuinfo] Supports: Debugging Extensions`
	`[cpuinfo] Supports: Page Size Extensions`

exceptions	Displays exception types and messages (excluding the common types of exceptions). Verbose output for exceptions might look like this: `[excepti][00002] java/lang/NumberFormatException: null`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

`exceptions=debug`	Displays exception types and messages (excluding the common types of exceptions).It also displays stacktraces; Verbose output for `exceptions=debug` might look like this:
	`[excepti][00002] java/lang/NumberFormatException: null`
	`at java/lang/Integer.parseInt(Ljava/lang/String;I)I(Integer.java:415)`
	`at java/lang/Integer.<init>(Ljava/lang/String;)V(Integer.java:620)`
	`at sun/net/InetAddressCachePolicy.<clinit>()V (InetAddressCachePolicy.java:77)`
	`at jrockit/vm/RNI.c2java(IIII)V(Native Method)`
	`at jrockit/vm/RNI.generateFixedCode(I)I(Native Method)`
	`at java/net/InetAddress.<clinit>()V(InetAddress.java:640)`
	`at jrockit/vm/RNI.c2java(IIII)V(Native Method)`
	`at jrockit/vm/RNI.generateFixedCode(I)I(Native Method)`
	`at java/net/InetSocketAddress.<init>(Ljava/lang/String;I)V (InetSocketAddress.java:124)`
	`at java/net/Socket.<init>(Ljava/lang/String;I)V (Socket.java:178)`
	`at Ex.main([Ljava/lang/String;)V(Ex.java:5)`
	`at jrockit/vm/RNI.c2java(IIII)V(Native Method)`
	`--- End of stack trace`

`exceptions=trace`	The same information as `debug`, but includes the common types of exceptions. Verbose output for `exceptions=trace` will look the same as `-Xverbose:exceptions=debug` but also prints exceptions of types: • `java.util.EmptyStackException` • `java.lang.ClassNotFoundException` • `java.security.PrivilegedActionException`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

`load`	The name of each loaded Java or native library:
	`[INFO ][load ] opened zip /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/rt.jar`
	`[INFO ][load ] opened zip /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/jsse.jar`
	`[INFO ][load ] opened zip /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/jce.jar`
	`[INFO ][load ] opened zip /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/charsets.jar`
	`[INFO ][load ] Loaded native library: /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/i386/libverify.so`
	`[INFO ][load ] Loaded native library: /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/i386/libjava.so`
	`[INFO ][load ] Loaded native library: /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/i386/native_threads/libhpi.so`
	`[INFO ][load ] Loaded native library: /localhome/jrockits/R27.5.0_R27.5.0-110_1.5.0/jre/lib/i386/libzip.so`

`gcpause`	`-Xverbose:gcpause` gives the same output as -XgcPause.
`gcreport`	`-Xverbose:gcreport` gives the same output as -XgcReport.

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

memdbg	Turns on memory printout and adds new special memdbg printouts. Verbose output for memdbg might look like this:
	`[memory ] 12.875: nursery GC 89648K->89716K (89716K), 3.296 ms`
	`[memdbg ] nursery GC 291: promoted 1510 objects (69744 bytes) in 3.296 ms`
	`[memdbg ] Page faults before GC: 36784, page faults after GC: 36800, pages in heap: 22429`
	`[finalizer] (YC) Pending finalizers 0->0`
	`[memdbg ] old collection 7 started`
	`[memdbg ] Compacting 8 heap parts at index 112 (type 2) (exceptional 0)`
	`[memdbg ] starting parallel marking phase`
	`[memdbg ] ending marking phase [memdbg ] current generational GC work score: 0.142956`
	`[memdbg ] last single generational GC work score: 0.081486`
	`[memdbg ] current error: -0.042956`
	`[memdbg ] previous nursery size: 736760`
	`[memdbg ] requested nursery size: 711984`
	`[memdbg ] starting parallel sweeping phase`
	`[memdbg ] ending sweeping phase`
	`[memory ] 11.841-12.025: GC 89716K->67088K (89716K), 184.000 ms`
	`[memdbg ] Page faults before GC: 36827, page faults after GC: 37036, pages in heap: 22429`
	`[finalizer] (OC) Pending finalizers 0->0`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

`memdbg with -XgcPrio: throughput`	Turns on memory printout and adds new special `memdbg` printouts. A report for a JVM running a dynamic garbage collector optimized for application throughput (`-XgcPrio:throughput`) with `memdbg` specified might look like this:
	`[memdbg ] nursery GC 3: promoted 22788 objects (1246K) in 28.472 ms`
	`[memdbg ] Page faults before GC: 24768, page faults after GC: 25288, pages in heap: 19170`
	`[finaliz] (YC) Pending finalizers 0->0`
	`[memdbg ] old collection 2 started`
	`[memdbg ] OC reasons: Large obj: 1 (4021248 bytes), TLA: 1, Promotion: 0, GCTrigger: 1, SystemGC: 0, Other: 0`
	`[memdbg ] Compacting 8 heap parts at index 0 (type 1) (exceptional 0)`
	`[memdbg ] starting parallel marking phase`
	`[memdbg ] ending marking phase`
	`[memdbg ] current generational GC work score: 0.266484`
	`[memdbg ] last single generational GC work score: 0.000000`
	`[memdbg ] current error: -0.166484`
	`[memory ] Changing GC strategy to single generation, parallel mark and parallel sweep`
	`[memdbg ] starting parallel sweeping phase`
	`[memdbg ] ending sweeping phase`
	`[memdbg ] expanding the heap from 74 MB to 87 MB`
	`[memory ] 15.882-16.157: GC 76680K->69203K (89716K), 275.413 ms`
	`[memdbg ] Page faults before GC: 25288, page faults after GC: 25765, pages in heap: 22429`
	`[finaliz] (OC) Pending finalizers 0->0`

`memory; gc`	Information about the memory management system, including: • Start time of collection (seconds since JVM start) • End time of collection (seconds since JVM start) • Memory used by objects before collection (KB) • Memory used by objects after collection (KB) • Size of heap after collection (KB) • Total time of collection (seconds or milliseconds) • Total pause time during collection (milliseconds) The information displayed by `-Xverbose:memory` or `-Xverbose:gc` will vary depending upon the type of garbage collector you are using.

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...
`memory`; `gc`	A report for a JVM running a generational concurrent collector (`-Xgc:gencon`) with `memory` or `gc` specified might look like this:

with `gencon`	`[memory ] GC strategy: gencon`
	`[memory ] heap size: 65536K, maximal heap size: 785672K, nursery size: 16384K`
	`[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms`
	`[memory ] <s/start> - start time of collection (seconds since jvm start)`
	`[memory ] <end> - end time of collection (seconds since jvm start)`
	`[memory ] <before> - memory used by objects before collection (KB)`
	`[memory ] <after> - memory used by objects after collection (KB)`
	`[memory ] <heap> - size of heap after collection (KB)`
	`[memory ] <pause> - total pause time during collection (milliseconds)`
	`[memory ] 1.069: parallel nursery GC 24995K->24810K (65536K), 40.038 ms`
	`[memory ] 1.535: parallel nursery GC 46818K->46701K (65536K), 31.319 ms`
	`[memory ] 8.698-9.247: GC 48085K->46437K (65536K), 230.190 ms`
	`[memory ] 9.252-9.915: GC 49055K->55834K (76680K), 216.105 ms`
	`[memory ] 9.928: parallel nursery GC 63287K->63287K (76680K), 20.282 ms`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...

`memory; gc` with `singlecon`	A report for a JVM running a single generation concurrent collector (`-Xgc:singlecon`) with `memory` or `gc` specified might look like this:
	`[memory ] GC strategy: singlecon`
	`[memory ] heap size: 65536K, maximal heap size: 785672K`
	`[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms`
	`[memory ] <s/start> - start time of collection (seconds since jvm start)`
	`[memory ] <end> - end time of collection (seconds since jvm start)`
	`[memory ] <before> - memory used by objects before collection (KB)`
	`[memory ] <after> - memory used by objects after collection (KB)`
	`[memory ] <heap> - size of heap after collection (KB)`
	`[memory ] <pause> - total pause time during collection (milliseconds)`
	`[memory ] 30.220-30.693: GC 65058K->55006K (76680K), 101.591 ms`
	`[memory ] 30.749-31.290: GC 76680K->73168K (89716K), 90.350 ms`
	`[memory ] 31.297-31.904: GC 79089K->89716K (104968K), 2.386 ms`

`memory; gc` with `parallel`	A report for a JVM running a parallel collector (`-Xgc:parallel`) with `memory` or `gc` specified might look like this:
	`[memory ] GC strategy: parallel`
	`[memory ] heap size: 65536K, maximal heap size: 785672K`
	`[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms`
	`[memory ] <s/start> - start time of collection (seconds since jvm start)`
	`[memory ] <end> - end time of collection (seconds since jvm start)`
	`[memory ] <before> - memory used by objects before collection (KB)`
	`[memory ] <after> - memory used by objects after collection (KB)`
	`[memory ] <heap> - size of heap after collection (KB)`
	`[memory ] <pause> - total pause time during collection (milliseconds)`
	`[memory ] 1.563-1.805: GC 65536K->55018K (76680K), 242.030 ms`
	`[memory ] 1.871-2.114: GC 76680K->73161K (89716K), 242.675 ms`
	`[memory ] 2.167-2.478: GC 89716K->86478K (104968K), 310.974 ms`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...
`memory;` `gc`	A report for a JVM running a dynamic garbage collector optimized for application throughput (-XgcPrio:`throughput`) with `memory` or `gc` specified might look like this:

with `-Xgcprio: throughput`	`[memory ] GC strategy: System optimized over throughput (initial strategy singleparpar)`
	`[memory ] heap size: 65536K, maximal heap size: 785672K`
	`[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms`
	`[memory ] <s/start> - start time of collection (seconds since jvm start)`
	`[memory ] <end> - end time of collection (seconds since jvm start)`
	`[memory ] <before> - memory used by objects before collection (KB)`
	`[memory ] <after> - memory used by objects after collection (KB)`
	`[memory ] <heap> - size of heap after collection (KB)`
	`[memory ] <pause> - total pause time during collection (milliseconds)`
	`[memory ] Changing GC strategy to generational, parallel mark and parallel sweep`
	`[memory ] 1.669-1.904: GC 65536K->54455K (76680K), 234.978 ms`
	`[memory ] 1.923: parallel nursery GC 66150K->67136K (76680K), 105.995 ms`
	`[memory ] 2.039: parallel nursery GC 71236K->71203K (76680K), 58.620 ms`
	`[memory ] 2.107: parallel nursery GC 75303K->76680K (76680K), 36.650 ms`
	`[memory ] Changing GC strategy to single generation, parallel mark and parallel sweep`
	`[memory ] 2.164-2.482: GC 76680K->69340K (89716K), 318.158 ms`

`opt`	Information about all methods that get optimized. Verbose output for `opt` might look like this:
	`[opt ] 280 2434  0 ObjAlloc.main([Ljava.lang.String;)V: 0.00 ms`
	`[opt ] 0 : 9.8996 ms`

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...
`referents`	The reference objects for each old generation garbage collection and the reference to which they point. `-Xverbose:referents` corresponds to the option `-Djrockit.verboserefs` in earlier JRockit JVM versions, prior to R27.2. Each reference type is broken down by reference class and referent. In the case of handles, only referents are shown; there are no references. The different counters tell how many instances of each type exists and how they are reachable (or cleared). The header/footer of the report informs what type of collection took place, how long ago the last old collection happened and how much memory that was free at that time is reported. If any soft references are present the user is told which softly reachable referents are collected based on when they where last looked up through `get()`. The performance overhead of this log module is high.
`refobj`	Information on reference objects and handles at each garbage collection. At info level the output is a summary of reference objects of different types and how many of them are "activated". A reference object is activated when the reachability requirements for the reference object type are fulfilled. Upon activation, the memory management system can clear the reference, enqueue it in a reference queue or enqueue it for finalization, depending on the type of reference. At debug level this module displays an improved version of the information previously displayed by `-Xverbose:referents`. The performance overhead of this log module is low on info level. On debug level, the performance overhead is high. This module is available in JRockit JVM R27.5 and later versions.
`stackoverflow`	Stack overflow errors as they occur. This output will typically be several pages long with the same information repeated *ad nauseum*.This is because, if the stack overflows, the stack trace will be extremely long and this parameter will cause the entire long stacktrace to print.

Table 2-10 -Xverbose Parameters

This Parameter...	Prints to the screen...
`starttime`	The values of `System.currentTimeMillis()` and `System.nanoTime()` at the time JRockit JVM started. These can be used to correlate log output between different processes. Verbose output for `starttime` might look like this: `[startti] VM start time: 1152871839957 millis 171588375730523 nanos` Where: • `millis` is the number of milliseconds elapsed since midnight, January 1, 1970 UTC. This is same value that `System.currentTimeMills`() would render. • `nanos` measures time to the resolution of one-billionth of a second (a nanosecond); however, the time from which `nanoTime()` is measured (the start time) is unspecified so that the most efficient method of measurement for different operating systems can be used. It is the same value that `System.nanoTime()` would render.

`systemgc`	Notifies of garbage collections started by a call to `System.gc()` or for a reason marked as "`other`" in JRA recordings and `-Xverbose:memdbg` outputs, for example a call to a JMAPI function that implicitly starts a garbage collection or to the diagnostics command `runsystemgc`.
	A garbage collection started by a direct call to `System.gc()` will result in a verbose output similar to:
	`[INFO ][sysgc ] GC requested by thread 1`
	The thread number in this output is the thread ID of the thread that requests the garbage collection.
	The output for a garbage collection started by other means will have the reason for the garbage collection printed out, for example:
	`[INFO ][sysgc ] GC triggered for reason: Set Nursery Size`
	Note: In JRockit JVM R27.3 and older versions, all the `-Xverbose:systemgc` outputs look the same. Furthermore, some of the garbage collections printed by `-Xverbose:systemgc` and treated as "`other`" were not printed out.

`timing`	The timer resolution and, on Linux, the method used to get a time value. This is the resolution of the timer used by the `System.nanoTime()` method. Here is an example of a verbose timing report on Windows: `[INFO ][timing ] Counter timer using resolution of 1779720000Hz`
`verboserefs`	Gives the same output as referents.

Log Levels

Log levels identify the levels of information recorded in the log. JRockit JVM uses six logging levels, as described in Table 2-11.

Table 2-11 -Xverbose Log Levels

Log Level	Description
`quiet`	No logging. No messages or errors are generated.
`error`	Only error messages are logged.
`warn`	Warning messages are logged along with errors. Still a low logging level, `warn` is usually used to warn about events that could possibly lead to an error later on.
`info`	At the `info` level, not only are errors and warnings logged, but also informational messages about the current state of JRockit JVM and various JVM events. This is the default logging level if `-Xverbose` is used without arguments.
`debug`	`debug` logs messages with detailed information of JRockit JVM's behavior. Ususally, `debug` provides too much information for day to day logging, but useful for debugging.
`trace`	`trace` provides very verbose logging. This level is used by modules where even the debug level would be cluttered by the amount of information generated. Typically, `trace` is used when up to ten or one hundred pages of text per minute needs to be logged.

Other Flags and Options Affected

`-Xverbose` must be set for the following options to work:

- `-XverboseDecorations`
- `-XverboseLog`
- `-XverboseTimeStamp`

Exceptions

None

-XverboseDecorations

Use this option to set the "decorations" JRockit JVM adds to verbose printouts. Decorations are additional information—usually system-related—used to enhance the meaningfulness of verbose output; for example, the name of the module in which the message originated or number of milliseconds elapsed since the current JRockit JVM session started.

Operation

Format: `-Xverbosedecorations=<`*`decoration names`*`>`

For Example:

If you include:

`java -Xverbose:gcpause` **`-XverboseDecorations=timestamp,`**`module` *`myApp`*

at startup, the output will include these decorations:

- A human readable *timestamp*.
- The name of the *module* in which the message originated.

Note: You can also use the control-break handler `verbosity` with the argument `decorations`.

The possible decorations are listed in Table 2-12.

Table 2-12 Verbose Output Decorations

Decoration	Description
`level`	Prints the logging level for the message.
`millis`	Prints the number of milliseconds elapsed since midnight, January 1, 1970 UTC. This is same value that would be generated by `System.currentTimeMills`().
`millisstart`	Prints the number of milliseconds elapsed since JRockit JVM started.
`module`	Prints the module in which the message originated, same as the arguments to `-Xverbose`.

Table 2-12 Verbose Output Decorations

Decoration	Description
`nanos`	Prints the same value that `System.nanoTime()` would render. "nanoTime" measures time to the resolution of one-billionth of a second (a nanosecond); however, the time from which `nanoTime()` is measured (the start time) is unspecified so that the most efficient method of measurement for different operating systems can be used.
`nanosstart`	Prints the number of nanoseconds since JRockit JVM started.
`pid`	Prints the process ID.
`threadid`	Prints the thread's index. This is the same value provided by `idx` in thread dumps.
`timestamp`	Prints a human readable timestamp. This is the same value you would receive if you used the `-XverboseTimeStamp` option.

Default Value

If you use `-XverboseDecorations` without specifying a decoration, the verbose output will display the `module`, `timestamp`, and `pid` (in that order); for example:

```
D:\jrockits\R27.1.0_R27.1.0-23_1.4.2\bin>java -Xverbose:load
-Xverbosedecorations -cp L:\src\ HelloWorld
[load   ][Wed Sep 13 19:43:14 2006][00728] opened zip
D:\jrockits\R27.1.0_R27.1.0-23_1.4.2\jre\lib\rt.jar
```

Flags or Other Options Affected

This option can only be used if `-Xverbose` is also set, as that option turns on verbose logging.

Exceptions

None

-XverboseLog

This option sends messages (such as verbose output and error messages) from the Oracle JRockit JVM to the specified file instead of `stderr`.

Operation

Format: `-Xverboselog:`*`myFile.txt`*

When this command is used with a filename and extension (for example, *`myFile.txt)`*, the JVM will write any logging information to the specified file.

For Example:

`java -Xverbose:gcpause` **`-Xverboselog:verboseText.txt`** *`myApp`*

writes verbose logging information for the class *`myApp`* to a file called `verboseText.txt`

Flags or Other Options Affected

This option can only work if `-Xverbose` is set, as that option turns on verbose logging.

Exceptions

This option does not print to screen.

-XverboseTimeStamp

This option adds a timestamp to the verbose printout, which can be useful when logging events.

Operation

Format: `-XverboseTimeStamp`

You can force a timestamp to print out with other information generated by `-Xverbose` if you combine it with the command `-XverboseTimeStamp`.

For Example:

`java -Xverbose` **`-XverboseTimeStamp`** *`myApp`*

The printout generated by `-XverboseTimeStamp` will precede the information printed by `-Xverbose`, as shown here:

```
L:\src>D:\jrockits\R27.1.0_R27.1.0-13_1.4.2\bin\java -Xverbose
-XverboseTimeStamp HelloWorld
[load   ][Mon Sep 25 09:57:56 2006][00624] opened zip
D:\jrockits\R27.1.0_R27.1.0-13_1.4.2\jre\lib\rt.jar
```

Flags or Other Options Affected

This option is only effective if verbose logging is enabled either by using `-Xverbose` or by enabling it at runtime.

Exceptions

None

-Xverify

This option allows you to manually verify the correctness of the bytecode. By performing these checks once at class loading time, as opposed to repeatedly during execution, this option helps improve runtime efficiency.

Operation

Format: `-Xverify:<param>`

Combine this option with one of the parameters described in Table 2-13.

Table 2-13 -Xverify Parameters

<param>	Description
`none`	Will not verify the bytecode. Note that, while using this parameter can lessen start-up time, you will lose some of the protection provided by Java.
`remote`	Verify just those classes loaded over network.
`all`	Verify all classes/

Default Value

If you don't use this option, the default is to verify just those classes loaded over the network (`-Xverify:remote`).

Other Flags or Options Affected

None

Exceptions

None

-XX Command-line Options

This section describes Oracle JRockit JVM's `-XX` command-line options; these options are all prefaced by `-XX`. To implement some of the options, specific system requirements must be met, otherwise, the particular option will not work. We recommend that you use these options *only*

- If you have a thorough understanding of your system.
- Are aware that, if used improperly, these options can have negative effect on the stability or performance of your system.

These options are subject to change without notice.

Note: This section contains an ever-changing list of options and is continually republished as necessary to include any new `-XX` options and to indicate any deprecated `-XX` options.

-XXaggressive

`-XXaggressive` is a collection of configurations that make the JVM perform at a high speed and reach a stable state as soon as possible. To achieve this goal, the JVM uses more internal resources at startup; however, it requires less adaptive optimization once the goal is reached. We recommend that you use this option for long-running, memory-intensive applications that work alone.

Note: What this option configures is subject to change between releases.

Operation

Format: `-XXagressive:<param>`

Combine `-XXaggressive` with one of the parameters listed in Table 3-1

Table 3-1 Parameters for -XXaggresive

<param>	Description
`opt`	Schedules adaptive optimizations earlier and enables new optimizations, which are expected to be the default in future releases.
`memory`	Configures the memory system for memory-intensive workloads and sets an expectation to enable large amounts of memory resources to ensure high throughput. JRockit JVM will also use large pages, if available. **Note:** Large pages cannot be swapped in some operating systems. In these cases they are only recommended in a well-balanced system.

For Example:

```
java -XXaggressive:opt myApp
```

By specifying `-XXagressive` with the `opt` parameter, adaptive optimizations will be scheduled earlier in runtime and new optimizations will be enabled.

Default Value

If neither `opt` nor `memory` is specified, the application will run as if both were specified.

Flags or Options Affected

This option will set several things, which can be reset or changed by adding the explicit options on the command line after `-XXaggressive`.

Exceptions

None

-XXallocClearChunks

This option allows you to clear a TLA for references and values at TLA allocation time and pre-fetch the next chunk. When an integer, a reference, or anything else is declared, it has a default value of 0 or null (depending upon type). At the appropriate time, you will need to clear these references and values to free the memory on the heap so Java can use—or reuse—it. You can do either when the object is allocated or, by using this option, when you request a new TLA.

Operation

Format: `-XXallocClearChunks`

or:

`-XXallocClearChunks=<true | false>`

This is a boolean option and is generally recommended on IA64 systems; ultimately, its use depends upon the application. If you want to set the size of chunks cleared, combine this option with `-XXallocClearChunkSize`.

Default Value

By default, this mechanism is disabled. If you use this option but do not specify a boolean value, the default is `true`.

Flags or Other Options Affected

None

Exceptions

While this mechanism is, by default, disabled, it is included in `aggressive:memory` on IA64 systems.

-XXallocClearChunkSize

When used with `-XXallocClearChunkSize`, this option sets the size of the chunks to be cleared.

Operation

Format: `-XXallocClearChunks` **`-XXallocClearChunkSize=`**<*size*>`[k|K][m|M][g|G]`

Combine this option with –`XXallocClearChunks` to set the size of the chunks to be cleared

For Example:

`java -XXallocClearChunks` **`-XXallocClearChunkSize=256m`** *myApp*

Default Value

If this option is used but no value is specified, the default is 512 bytes.

Flags or Other Options Affected

See Operation for a description of how this option works with `-XXallocClearChunks`.

Exceptions

This option cannot be used unless `-XXallocClearChunks` is also used.

-XXallocPrefetch

With this option a thread-local area is split into chunks and, when a new chunk is reached, the subsequent chunk is prefetched.

Note: To fully benefit from this feature on Intel Xeon servers, it is recommended that you disable hardware prefetching in the computer's BIOS.

Operation

Format: `-XXallocPrefetch`

For Example:

```
java -XgcPrio:pausetime -XXallocPrefetch myApp
```

Default Value

If this flag is not set, allocation prefetch is disabled.

Other Flags and Options Affected

This option must be set if you want to also use `-XXallocRedoPrefetch`.

Exceptions

None

-XXallocRedoPrefetch

With this option, an additional chunk (that is, two chunks subsequent) is prefetched whenever a new chunk is used.

Note: To fully benefit from this feature on Intel Xeon servers, it is recommended that you disable hardware prefetching in the computer's BIOS.

Operation

Format: `-XXallocRedoPrefetch`

For Example:

```
java -XXallocPerfetch -XXallocRedoPrefetch myApp
```

Default Value

If this flag is not set, chunks will be fetched normally.

Other Flags and Options Affected

None

Exceptions

This option will not work unless `-XXallocPrefetch` is set.

-XXcallProfiling

This option enables the use of call profiling for code optimizations. Profiling records useful runtime statistics specific to the application and can—in many cases—increase performance because JVM can then act on those statistics.

Note: This option is supported with the JRockit JVM R27.3.0 and later version. It may become default in future versions.

Operation

Format: `-XXcallProfiling`

For Example:

```
java -XXcallProfiling myApp
```

Default Value

This option is disabled by default. You must enable it to use it.

Other Flags and Options Affected

None

Exceptions

None

-XXcompactRatio

This option sets the compaction ratio. Compaction is the garbage collector's main weapon against fragmentation. The idea is to look at a part of the heap and move all the live objects in that part together to create larger consecutive areas of free space. While the JVM is compacting the heap, all threads that want to access objects have to stand still because the JVM is moving the objects around. Consequently, only a part of the heap is compacted to reduce pause time.

In some cases, when the garbage collection time is too long it might be beneficial to reduce the compaction area to reduce the pause times. In some other cases, especially when you are allocating very large arrays, it may be necessary to increase the compaction area to reduce the fragmentation on the heap and thus make allocation faster.

Note: Since JRockit JVM now employs dynamic compaction, `-XXcompactRatio` is rarely used anymore.

Operation

Format: `-XXcompactRatio:<nn>`

Specify a ratio (`[nn]`) of the compaction rate with a percentage size of the heap.

For Example:

If

`java` **`-XXcompactRatio:10`** *myApp*

is set in a 500 MB heap, the garbage collector will compact 50 MB of the heap at each garbage/old collection.

Default Value

If you are running a static compaction, the default is approximately 6%; however, if you are not using static compaction, the default applies only at the beginning of the run.

Flags or Other Options Affected

Use of - `XXcompactRatio` will affect certain options, as described here:

- Setting `-XXcompactRatio` while running `-XgcPrio:deterministic` or `-XgcPrio:pausetime` might result in non-deterministic pause times.

- `-XXfullCompaction` is equivalent to `-XXcompactRatio:100` and they will override each other.
- You should not use this option together with `-XXthroughputCompaction` as that might reduce the throughput

Exceptions

When using `-XXcompactRatio`, be aware of the following exceptions:

- You cannot use `-XXcompactRatio` together with `-XXnoCompaction` since this will disable compaction.
- You cannot set all three of `-XXcompactRatio`, `-XXinternalCompactRatio`, and `-XXexternalCompactRatio` at the same time. Setting two of these options simultaneously *is* allowed.

-XXcompactSetLimit

This option sets the maximum number of references to objects in the compaction area.

Compaction is the process of moving live objects closer together in the Java heap to create larger free areas that can be used for allocation of large objects. The JRockit JVM compacts a small part of the Java heap at each garbage collection. The references to the objects in the compacted area are stored in a *compact-set*. When running non-deterministic garbage collection, the number of references to the compaction area will affect the compaction pause. This option can be used to limit the size of the compact-set and thus limit the compaction pause somewhat.

Operation

Format: `-XXcompactSetLimit:<size>`

Enter the preferred compaction limit following the command.

For Example:

`java` **`-XXcompactSetLimit:10000`** *`myApp`*

sets the compaction limit to 10,000 references to objects in the compaction area.

Default Values

By default the limit is a dynamic value. The default initial value depends on the garbage collector and release. Table 3-2 lists those defaults.

Table 3-2 -XXcompactSetLimit Defaults

Releases	Garbage Collection Type	Default Initial Value
R27.1 and older	A static garbage collector or `-Xgcprio:pausetime`	91490
R27.2 and later	A static garbage collector or `-Xgcprio:pausetime`	299900
All	`-Xgcprio:throughput`	7600010
All	`-Xgcprio:deterministic`	10200

Flags or Other Options Affected

None

Exceptions

When using `-XcompactSetLimit`, be aware of the following exceptions:

- When the compaction area size is increased due to failing object allocation, the compact-set limit is ignored.
- You cannot use `-XXcompactSetLimit` together with any of the following options:
  - `-XXnoCompaction`
  - `-XXfullCompaction`
  - `-XXusePointerMatrix`

-XXcompactSetLimitPerObject

WARNING: This is an advanced tuning option. You should only use it if you understand how it works and are prepared to accept the consequences of its misuse.

This option sets the maximum number of references to any single object in the compaction area. If the number of references to an object exceeds this value, the object will not be moved during the compaction.

Compaction is the process of moving live objects closer together in the Java heap to create larger free areas that can be used for allocation of large objects. JRockit JVM compacts a small part of the Java heap at each garbage collection. When an object is moved during compaction, the references to that object must be updated. Moving an object with a lot of references to it is thus more costly than moving an object with only a few references to it.

Operation

Format: `-XXcompactSetLimitPerObject:<size>`

For example:

`java` **`-XXcompactSetLimitPerObject:500`** *`myApp`*

sets the compaction limit per object to 500 references.

Default Value

The default value is 100

Flags or Other Options Affected

None

Exceptions

This option can only be used if one of the following options is also set:

- `-XgcPrio:deterministic`
- `-XgcPrio:pausetime`
- `-XXusePointerMatrix`

-XXcompressedRefs

This flag governs the use of compressed references, limiting all pointers stored on the heap to 32 bits. Compressed references use fewer Java heap resources and transport less data on the memory bus, thus improving performance. This option is also useful because it frees space on the heap that might not have been available had the references not been compressed.

Operation

Format: `-XXcompressedRefs[=[true|1|false|0]]`

For Example:

`java -XgcPrio:pausetime` **`-XXcompressedRefs=true`** *`myApp`*

enables compressed references.

or

`java -XgcPrio:pausetime` **`-XXcompressedRefs=0`** *`myApp`*

disables compressed references.

Default Value

If `-XXcompressedRefs` is not specified, compressed references are enabled on all 64-bit machines as long as the heap size is less than 4 GB. This is typically controlled using the `-Xmx` flag.

Note: In the JRockit JVM R26.4 and earlier, compressed references are disabled by default.

Flags or Other Options Affected

Other command-line options are affected by `-XXcompressedRefs` as described here:

- If you use this option with an initial heap size (`-Xms`) that is too large, execution will stop and an error message will be generated.
- If you do not specify compressed references explictly by using this option and you specify either an initial heap (`-Xms`) or a maximum heap (`-Xmx`) that is too large for compressed references, JRockit JVM will stop.

Exceptions

The following exceptions apply:

- If compressed references are not available on given hardware or operating system, a warning will be printed and execution will be stopped.
- The heap size will be limited to less than 4 GB; therefore, you can only use this option for applications that demand less than 4 GB of live data. The heap will be reduced to meet this size limitation if you specify a larger initial (`-Xms`) or maximum (`-Xmx`) heap size.

-XXdisableFatSpin

This option disables the fat lock spin code in Java, allowing threads that block trying to acquire a fat lock go to sleep directly.

Objects in Java become a lock as soon as any thread enters a synchronized block on that object. All locks are held (that is, stayed locked) until released by the locking thread. If the lock is not going to be released very fast, it can be inflated to a “fat lock.” “Spinning” occurs when a thread that wants a specific lock continuously checks that lock to see if it is still taken, spinning in a tight loop as it makes the check. Spinning against a fat lock is generally beneficial although, in some instances, it can be expensive and might affect performance. `-XXdisableFatSpin` allows you to turn off spinning against a fat lock and eliminate the potential performance hit.

Operation

Format: `-XXdisableFatSpin`

Flags or Other Options Affected

None

Exceptions

None

-XXdisableGCHeuristics

This option disables the garbage collector strategy changes. Compaction heuristics and nursery size heuristics are not affected by this option.

Operation

Format: `-XXdisableGCHeuristics`

Default Value

By default, the garbage collection heuristics are enabled.

Flags or Other Options Affected

None

Exceptions

In releases prior to R27.5, this option requires that either `-XgcPrio:throughput` or `-XgcPrio:pausetime` is being used. It does not work with `-Xgc` or `-XXsetGC`. Note that `-XgcPrio:throughput` is default.

In the JRockit JVM R27.5 and later releases this option also disables temporary strategy changes to parallel mark or sweep in the static concurrent garbage collectors.

-XXdumpFullState

Usually when the JRockit JVM crashes it saves out the state of the process (called a core dump), but the heap is removed from this state since it is huge and would take up a lot of disk space. With this option we will save all the process state including the heap. More disk space will be used, but it makes it much easier for us to use the core dump to find out what the problem was that caused the crash. This option saves a significant amount of information to disk. If you don't want to save all the information `that -XXdumpFullState` saves, use `-XXdumpSize:normal`.

This option is the default.

Operation

Format: `-XXdumpFullState`

Flags or Other Options Affected

`-XXdumpFullState` is equivalent to `-XXdumpsize:large`. For more information, see -XXdumpSize.

Exceptions

None

-XXdumpSize

This option causes a dump file to be generated and allows you to specify the relative size of that file (that is, small, medium, or large).

Operation

Format: `-XXdumpsize:<File Size>`

Use the command with one of the parameters listed in Table 3-3 to specify the relative size of the dump file.

Table 3-3 Parameters for -XXdumpsize

<File Size>	Description
`none`	does not generate a dump file.
`small`	On Windows, a small dump file is generated (on Linux a full core dump is generated). A small dump only include the thread stacks including their traces and very little else. This was the default in the JRockit JVM 8.1 with service packs 1 and 2, as well as 7.0 with service pack 3 and higher).
`normal`	Causes a normal dump to be generated on all platforms. This dump file includes all memory except the java heap. This is the default value for the JRockit JVM 1.4.2 and later.
`large`	Includes everything that is in memory, including the Java heap. This option makes `-XXdumpSize` equivalent to `-XXdumpFullState`.

Flags or Other Options Affected

None

Exceptions

None

-XXexitOnOutOfMemory

This option makes JRockit JVM exit on the first occurrence of an out of memory error. It can be used if you prefer restarting an instance of JRockit JVM rather than handling out of memory errors.

Operation

Format: `-XXexitOnOutOfMemory`

Enter this command at startup to force JRockit JVM to exit on the first occurrence of an out of memory error

Flags or Other Options Affected

None

Exceptions

None

-XXexternalCompactRatio

This option sets the number of heap parts to compact during external compaction (also called "evacuation").

Note: You can change the total number of heap parts by using the command line option `-XXheapParts`.

Operation

Format: `-XXexternalCompactRatio=`*nn*

For Example:

`java` **`-XXexternalCompactRation=12`** *myApp*

Default Value

The default number of parts to compact is 8 for `-XgcPrio:throughput`, for all other garbage collection modes it is dynamic.

Flags or Other Options Affected

Using this option together with `-XgcPrio` or `-XXthroughputCompaction` disables parts of the dynamic compaction heuristics and can affect performance.

Exceptions

When using `-XXexternalCompactRatio`, be aware of these exceptions:

- The external compact ratio set by this option is ignored when external compaction of the end of the heap is done for the purpose of shrinking the heap. It is also ignored if an increased compact ratio is required for avoiding an Out of Memory Error
- You cannot use `-XXexternalCompactRatio` together with the following options:
  - `-XXstaticCompaction`
  - `-XXnoCompaction`
- You cannot set these three options simultaneously:
  - `-XXcompactRatio`
  - `-XXinternalCompactRatio`
  - `-XXexternalCompactRatio`

Setting two of them is allowed.

-XXfullCompaction

`-XXfullCompaction` causes full compaction at all times, compacting the entire heap at each old collection. Compaction is the process of moving live objects closer together in the Java heap to create larger free areas that can be used for allocation of large objects. By default, the JRockit JVM compacts a small part of the Java heap at each garbage collection. Full compaction can increase the application throughput by minimizing the fragmentation of the heap but can also cause extremely long garbage collection pauses during the compaction.

Operation

Format: `-XXfullCompaction`

Enter this command at startup to force full compaction. This is the only way to ensure that full compaction occurs.

Flags or Other Options Affected

When using `-XXfullCompaction`, be aware of the following:

- Setting `-XXfullCompaction` is the same as setting `-XXcompactRatio:100`.
- You should not use this option together with `-XXthroughputCompaction` as that might reduce the throughput.

Exceptions

You cannot use `-XXfullCompaction` together with any of the following options:

- `-XXnoCompaction`
- `-XXcompactSetLimit`
- `-XXstaticCompaction`

-XXfullSystemGC

This option causes the garbage collector to do a full garbage collection every time `System.gc()` is called. Full garbage collection includes old space collection and the elimination of soft references. Use this option when you want the garbage collector to do maximum garbage collecting every time you explicitly invoke a garbage collection from Java.

This option is useful when the default garbage collector doesn't free enough memory; however, using it can cause longer garbage collection pauses.

Operation

Format: `-XXfullSystemGC`

When you use this option, if an old space collection is already running when `System.gc()` is called, it will first wait for it to finish and then trigger a new old space collection. `-XXfullSystemGC` frees all softly referenced objects.

Flags or Other Options Affected

None

Exceptions

You cannot use `-XXfullSystemGC` together with `-XXnoSystemGC`.

-XXgcThreads

This option specifies how many garbage collection threads the garbage collector will use. This applies both to parallel nursery and parallel old space collectors as well as the concurrent and deterministic collector.

Operation

Format: `-XXgcthreads:<# threads>`

For Example:

`java -XgcPrio:pausetime` **`-XXgcThreads:4`** *`myApp`*

sets “4” as the number of garbage collection threads the garbage collector can use during the parallel phases.

Default Value

By default, these values are based on the number of cores and hardware threads on the machine.

Flags and Other Options Affected

None

Exceptions

This option is valid only from JRockit JDK 5.0 P26.0.0 and JRockit JDK 5.0 R26.4.0.

-XXgcTrigger

This option determines how much free memory should remain on the heap when a concurrent garbage collection starts. If the heap becomes full during the concurrent garbage collection, the Java application can't allocate more memory until the garbage collection frees some heap space, which might cause the application to pause. While the trigger value will tune itself in runtime to prevent the heap from becoming too full, this automatic tuning may take too long. Instead, you can use `-XXgcTrigger` to set from the start a garbage collection trigger value more appropriate to your application.

If the heap becomes full during the concurrent mark phase, the sweep phase will revert to parallel sweep (unless `-XXnoParSweep` has been specified). If this happens frequently and the garbage collection trigger doesn't increase automatically to prevent this, use `-XXgcTrigger` to manually increase the garbage collection trigger.

Operation

Format: `-XXgcTrigger:`*nn*

Where *nn* is the amount of free heap, as a percentage of the heap, available when a garbage collections triggered.

For example:

`java -XXgcTrigger:50` *myApp*

With this option set, JRockit JVM will trigger a garbage collection when 50% of the heap—for example, about 512 MB on a 1 GB heap—or less remains free. The current value of the garbage collection trigger will appear in the `-Xverbose:memdbg` outputs whenever the trigger changes.

Default Value

If `-XXgcTrigger` isn't specified, the system will try to automatically find a good percentage value. If `-XXgcTrigger:`*nn* is specified, it will be used instead and no automatic process is involved.

Other Flags and Options Affected

None

Exceptions

The garbage collector ignores the `-XXgcTrigger` value when it runs both parallel mark and parallel sweep, for example if you specify `-Xgc:singlepar` or `-Xgc:genpar` on the command line.

-XXheapParts

This option sets the number of heap parts to a specified, static value.

Operation

Format: `-XXheapParts=nn`

For Example:

`java` **`-XXheapParts=64`** *myApp*

Default Value

The default number of heap parts is 128. The number of heap parts can increase during runtime.

Flags or Other Options Affected

The options `-XXexternalCompactRatio` and `-XXinternalCompactRatio` use heap parts as a unit. Changing the number of heap parts will thus affect how you should set these options.

Exceptions

None

-XXhpm

This option uses hardware performance counters instead of software sampling in the HotSpot detector, which drives optimizations. The hardware performance counters gives higher accuracy for the hot spots sampling with better performance. This option is disabled b default.

Operation

Format: `-XXhpm`

Other Flags Affected

None

Exceptions

This option is available with and supported on Itanium for Red Hat 4.0 and SuSE 9.0.

-XXinitialPointerVectorSize

WARNING: This is an advanced tuning option. You should only use it if you understand how it works and are prepared to accept the consequences of its misuse.

This option sets the initial size of each "row" in the pointer matrix. The pointer matrix is a data structure used for storing references during compaction if `-XgcPrio:deterministic` or `-XgcPrio:pausetime` is used, or if `-XXusePointerMatrix` is set. Increasing the initial pointer vector size will increase the JRockit JVM's memory footprint, but may in some cases increase the application throughput.

Operation

Format: `-XXinitialPointerVectorSize:<size>`

For Example:

`java` **`-XXinitialPointerVectorSize:40`** *`myApp`*

sets the initial pointer vector size to 40 references.

Default Value

The default value is 20

Flags or Other Options Affected

None

Exceptions

This option works only when either `-XgcPrio:deterministic` or `-XgcPrio:pausetime` are used or if -XXusePointerMatrix is set.

-XXinternalCompactRatio

Sets the number of heap parts to compact during internal compaction.

Note: The total number of heap parts can be changed with the command line option `-XXheapParts`.

Operation

Format: `-XXinternalCompactRatio=nn`

For Example:

`java -XgcPrio:throughput` **`-XXinternalCompactRatio=12`** *`myApp`*

Sets the number of heap parts to compact to 12. Note that, since `-XgcPrio:throughput` is being used with this command, this overrides the default value of eight parts.

Default Value

The default number of parts to compact is 8 for `-XgcPrio:throughput`, for all other garbage collection modes it is dynamic.

Flags or Other Options Affected

Using this option together with `-XgcPrio` or `-XXthroughputCompaction` disables parts of the dynamic compaction heuristics and can affect performance.

Exceptions

When using `-XXinternalCompactRatio`, be aware of the following exceptions:

- The internal compact ratio set by this option is ignored if a temporarily increased compaction is required for avoiding an Out of Memory Error.
- You cannot use `-XXinternalCompactRatio` together with the following options:
  - `-XXstaticCompaction`
  - `-XXnoCompaction`
- You cannot set these three options simultaneously:
  - `-XXcompactRatio`
  - `-XXinternalCompactRatio`

– `-XXexternalCompactRatio`

You can set two of them at the same time, however.

-XXjra

This option enables JRA recordings. A JRA recording is a way for you to get statistics on the JRockit JVM instance you are running.

Note: This command was added beginning with JRockit JVM 1.4.2_04. Prior to that, a separate command was required for each permutation of the command, as described in Table 3-4.

Operation

Format: Format is determined by your JRockit JVM version:

- If you are running the JRockit JVM version 1.4.2_04 or later use the command `-XXjra` together with the parameters listed in the JRockit JVM 1.4.2_04 or Later in Table 3-4; for example:

  ```
  -XXjra:delay
  ```

- If you are running the JRockit JVM version 1.4.2_03 or earlier, you need to set each parameter with its own startup option (listed in JRockit JVM 1.4.2_03 or Earlier in Table 3-4).; for example:

  ```
  -XXjraDelay
  ```

Table 3-4 Command Line Startup Parameters for JRA

JRockit JVM 1.4.2_04 or Later	JRockit JVM 1.4.2_03 or Earlier	Description
`delay`	`-XXjraDelay`	Amount of time, in seconds, to wait before recording starts.
`recordingtime`	`-XXjraRecordingTime`	Duration, in seconds, for the recording. This is an optional parameter. If you don't use it, the default is 60 seconds)
`filename`	`-XXjraFilename`	The name of recording file. This is an optional parameter. If you don't use it, the default is `jrarecording.xml`.
`sampletime`	`-XXjraSampleTime`	The time, in milliseconds, between samples. Do not use this parameter unless you are familiar with how it works. This is an optional parameter.

Table 3-4 Command Line Startup Parameters for JRA

JRockit JVM 1.4.2_04 or Later	JRockit JVM 1.4.2_03 or Earlier	Description
`nativesamples`	`-XXjraNativeSamples`	Displays method samples in native code; that is, you will see the names of functions written in C-code. This is an optional parameter.
`methodtraces`	Not applicable	Enables stack traces when set to `true`, with a maximum depth of 16 (default) or the value of the tracedepth parameter. When methodtraces is set to `false` it uses the default value from the hotspotdetector (3), or if the hotspotdetector is not running (`-Xnoopt`) it uses no stack traces.
`tracedepth`	Not applicable	Allows you to set the maximum stack trace depth value in JRA recordings above the default of 16 frames. **Note:** You can also increase the maximum stack trace depth by using his parameter with the `jrarecording` control-break handler.
`heapstat=<true | false>`	Not applicable	Allows you to enable or disable the tracking of heap statistics. • `-XXjra:heapstat=true` enables heap statistic tracking • `-XXjra:heapstat=false` disables heap statistic tracking. This tracking is enabled by default but, under certain circumstances can adversely affect transaction latency. In those situations, it is strongly recommended that you disable heap statistic tracking.
`methodsampling=<true | false>`	Not applicable	When set to `true`, this command enables method sampling.
`gcsampling=<true | false>`	Not applicable	When set to `true`, this command enables the presentation of garbage collection information.

Table 3-4 Command Line Startup Parameters for JRA

JRockit JVM 1.4.2_04 or Later	JRockit JVM 1.4.2_03 or Earlier	Description
`zip=<true | false>`	Not applicable	When set to `true`, this command causes the recording file to be sipped up into a smaller file.
`hwsampling=<true | false>`	Not applicable	When set to `true` and if it is possible on the machine, this command tells the JRA to sample the hardware.
`threaddump=<true | false>`	Not applicable	When set to `true`, this command forces a thread dump at the beginning and at the end of the recording.
`threaddumpinterval=`*nn*`[ns|ms|s]`	Not applicable	Forces a thread dump at the specified interval (=*nn*). The interval can be set in nanoseconds (`ns`), milliseconds (`ms`) or seconds (`s`); for example: `threaddumpinterval=10ms` sets the thread dump interval at 10 milliseconds.
`latency=<true | false>`	Not applicable	When set to `true`, this command enables the Latency Analysis Tool (LAT).
`latencythreshold=`*nn*`[ns|ms|s]`	Not applicable	Tells the LAT to record only those events that last longer than time the specified. The interval can be set in nanoseconds (`ns`), milliseconds (`ms`) or seconds (`s`); for example: `latencythreshold=10ms` instructs the LAT to record only events that last longer10 milliseconds

Table 3-4 Command Line Startup Parameters for JRA

JRockit JVM 1.4.2_04 or Later	JRockit JVM 1.4.2_03 or Earlier	Description
`cpusamples=<true | false>`	Not applicable	When set to `true`, this command tells the JRA to sample CPU usage during the recording.
`cpusamplesinterval=`*`nn`*`[ns|ms|s]`	Not applicable	This command sets the interval for CPU sampling during the recording. The interval can be set in nanoseconds (`ns`), milliseconds (`ms`) or seconds (`s`); for example: `cpusamplesinterval=10ms` say that CPU usage will be sampled every 10 milliseconds.

For Example:

`java` **`-XXjra:delay=10,recordingtime=100,filename=jrarecording2.xml`** *`myApp`*

would result in a recording that:

- Commenced ten seconds after the JRockit JVM started (`delay=10`).
- Lasted 100 seconds (`recordingtime=100`).
- Was written to a file called `jrarecording2.xml` (`filename=jrarecording2.xml`).

To replicate this data with the JRA version released with the JRockit JVM 1.4.2_03 or older, you would need to enter the following four separate commands:

- `-XXjraDelay=10`
- `-XXjraRecordingTime=100`
- `-XXjraFilename=jrarecording2.xml`

Avoid Using Multiple Options

Do not add multiple `-XXjra` options to the command line. If you do, all `-XXjra` options except the final one will be discarded; for example, if you enter:

```
java -XXjra:filename=apa.jra -XXjra:delay=10 -XXjra:time=11 Hello
```

You get the same result as if you'd simply entered:

```
java -XXjra:time=11 Hello
```

This is because each time `-XXjra` is parsed, the default values are reset, then the sub-arguments given to this argument are parsed and set. Instead, to get the result you expect from the first example, you should enter:

```
java -XXjra:filename=apa.jra,delay=10,time=11 Hello
```

Flags or Other Options Affected

None

Exceptions

None

-XXkeepAreaRatio

This option sets the size of the keep area within the nursery as a percentage of the nursery. The keep area prevents newly allocated objects from being promoted to old space too early.

Note: This option is only available in JRockit JVM R27.3 and later releases.

Operation

Format: `-XXkeepAreaRatio:<percentage>`

For Example:

`java` **`-XXkeepAreaRatio:10`** *`myApp`*

sets the keep area size to 10% of the nursery size.

Default Value

By default the keep area is 25% of the nursery size. The keep area may not exceed 50% of the nursery size.

Flags or Other Options Affected

None

Exceptions

The keep area ratio is only valid when the garbage collector is generational.

-XXlargeObjectLimit

This option sets a size for when an object is considered large (in terms of memory management). Objects larger than the limit are considered large and will not be allocated in TLAs. These are default limits that you can change by using the `-XXlargeObjectLimit:nn`. These limits apply to JRockit JVM 1.4.2 and higher only.

Operation

Format: `-XXlargeObjectLimit:<size>[k|K][m|M][g|G]`

Combine the `-XXlargeObjectLimit` with a memory value and unit (<*value*><*unit*>).

For Example:

`java` **`-XXlargeObjectLimit:6K`** *`myApp`*

sets the large object limit to 6 kilobytes. There is no minimum or maximum large object limit.

Default Value

If no value is specified, the default is set to whichever is the lower value of the minimum TLA size and the preferred TLA size divided by 2.

Flags or Other Options Affected

When you set the minimum and/or the preferred TLA size, the large object limit as well as the minimum block size (set with `-XXminBlockSize`) may be adjusted automatically by JRockit JVM if necessary. At all times, the following relations are maintained between minimum and preferred TLA size, large object limit, and minimum block size:

```
-XXlargeObjectLimit <= -XXtlaSize:min <= -XXminBlockSize
-XXtlaSize:min <= -XXtlaSize:preferred
```

If you set two or more of the options, then you must make sure that the values you use fulfil these criteria.

It is recommended to that you primarily set the TLA size parameters for memory management tuning purposes, while you let JRockit JVM automatically adjust the large object limit and minimum block size if necessary.

Exceptions

None

-XXlargePages

This an old option that tells JRockit JVM to use large pages, if they are available, for the Java heap and other areas in the JVM. Large pages allow your application to more effectively use the translation look-aside buffer (TLB) in the processor.

Note: This is no longer the preferred option for enabling large pages. Instead, you should use -XlargePages.

-XXlazyUnlocking

When `-XXlazyUnlocking` is set, locks will not be released when a critical section is exited. Instead, once a lock is acquired, the next thread that tries to acquire such a lock will have to ensure that the lock is or can be released. It does this by determining if the initial thread still uses the lock. A shared lock will convert to a normal lock and not stay in lazy mode.

Operation

Format: `-XXlazyUnlocking`

For Example:

`java` **`-XXlazyUnlocking`** *`myApp`*

This example enables lazy unlocking in JRockit JVM R27.4 and older releases.

R27.5 Format: `-XXlazyUnlocking:enable=<true|false>`

For Example:

`java` **`-XXlazyUnlocking:enable=false`** *`myApp`*

This example disables lazy unlocking in JRockit JVM R27.5.

Default Value

In R27.5 lazy unlocking is enabled by default in Java SE 6 versions of JRockit JVM on all platforms except IA64 and with all garbage collection modes except the deterministic garbage collection mode.

Lazy unlocking is disabled by default in older releases.

Other Flags and Options Affected

None

Exceptions

This option is intended for applications with many unshared locks. Be aware that it can introduce performance penalties with applications that have many short-lived but shared locks.

-XXmaxPooledPointerVectorSize

WARNING: This is an advanced tuning option. You should only use it if you understand how it works and are prepared to accept the consequences of its misuse.

This option sets the maximum limit for pooling large pointer vectors between garbage collections. The pointer vectors are rows in the "pointer matrix," a data structure used for storing references during compaction if `-XgcPrio:deterministic` or `-XgcPrio:pausetime` is used, or if `-XXusePointerMatrix` is set. Increasing the maximum pooled pointer vector size will increase JRockit JVM's memory footprint, but can, in some cases, increase the application throughput.

Operation

Format: `-XXmaxPooledPointerVectorSize:<size>`

For Example:

`java` **`-XXmaxPooledPointerVectorSize:8000`** *myApp*

sets the maximum limit for pooling large pointer vectors to 8,000.

Default Value

The default value is 5120

Flags or Other Options Affected

None

Exceptions

This option only has effect when `-XgcPrio:deterministic` or `-Xgcprio:pausetime` is used, or if `-XXusePointerMatrix` is set.

-XXmme

This flag enables the mixed mode Java execution feature (MME). This feature is supported on 64-bit Intel Itanium Linux system. It allows user Java applications that contain 32-bit IA-32 JNI native code to run on an unmodified Intel Itanium platform.

Operation

Format: `-XXmme`

To enable mixed mode Java execution, enter this:

`java` **`-XXmme`** *`myApp`*

Default Value

If `-XXmme` is not specified, the mixed mode Java execution feature is disabled.

Flags or Other Options Affected

None

Exceptions

This feature is only available on 64-bit Intel Itanium systems. Currently, the only supported operating system is Red Hat Enterprise Linux 4 Update 4 with Intel IA-32 Execution Layer v6 or later installed.

-XXminBlockSize

`-XXminblocksize` sets the minimum block size, which is the smallest memory area that will be returned to the freelist. Consequently, this option sets the smallest available chunk of memory on the freelists.

Note: This option might not always be the best solution for setting a block size. In most instances, you will experience better results if you use -XXtlaSize.

Operation

Format: `-XXminBlockSize:<memSize>`

Where *<memSize>* is the size of the memory area that will be returned to the freelist.

To speed up garbage collection and object allocation, the JVM ignores free chunks smaller than the minimum block size. Free chunks smaller than the minimum block size cannot be used for object allocation; these free chunks are called “dark matter.” Dark matter is wasted heap memory. Increasing the minimum block size will make allocation of large objects faster and may speed up garbage collection, but may also increase the amount of dark matter. An increased amount of dark matter will increase the number of garbage collections.

Default Value

The default block size is 2 KB.

Flags or Other Options Affected

None

Exceptions

If two or more of `-XXlargeObjectLimit`, `-XXtlaSize`, and `-XXminBlockSize` are set, they must have the following relationship:

```
-XXlargeObjectLimit <= -XXtlaSize <= -XXminBlockSize
```

Default Value

If no value is specified, the default is set to whichever is the lower value of the minimum TLA size and the preferred TLA size divided by 2.

Flags or Other Options Affected

The large object limit (set with `-XXlargeObjectLimit`) as well as the minimum block size may be adjusted automatically by JRockit JVM if you set the minimum and/or the preferred TLA size,

At all times, the following relations are maintained between minimum and preferred TLA size, large object limit, and minimum block size:

```
-XXlargeObjectLimit <= -XXtlaSize:min <= -XXminBlockSize
-XXtlaSize:min <= -XXtlaSize:preferred
```

If you set two or more of the options, then you must make sure that the values you use fulfil these criteria.

It is recommended to that you primarily set the TLA size parameters for memory management tuning purposes, while you let JRockit JVM automatically adjust the large object limit and minimum block size if necessary.

Exceptions

None.

-XXnoCompaction

Disables compaction during garbage collection. Compaction is the process of moving live objects closer together in the Java heap to create larger free areas that can be used for allocation of large objects. Disabling compaction may reduce garbage collection pause times, but may also lead to fragmentation in the Java heap and thus lower the application throughput or in the worst case cause an OutOfMemoryError to be thrown.

Operation

Format: `-XXnocompaction`

During every garbage collection, at least a partial compaction is done. If you prefer no compaction, assuming the application can survive without it, you must use this command at startup to disable it.

Flags or Other Options Affected

None

Exceptions

You cannot use the following options if compaction is disabled:

- `-XXcompactRatio`
- `-XXfullCompaction`
- `-XXcompactSetLimit`
- `-XXstaticCompaction`

-XXnoJITInline

This option turns off JIT inlining through JRockit JVM. “JIT inlining” inlines calls to small methods as soon as they are encountered in the code pipeline.

Note: Using this option will cause a small performance penalty.

Operation

Format: `-XXnoJITInline`

Flags and Other Options Affected

You should only use this option in combination with `-XnoOpt`. `-XXnojitinline` only disables inlining the first time a method is compiled. Unless you set `-XnoOpt` as well, methods can still be inlined when code is optimized.

Exceptions

None

-XXnoSystemGC

This option prevents a call to `System.gc()` from starting a garbage collection. If your application uses `System.gc()` and you want to the garbage collector itself to decide when to run the collection (default behavior), you should use this option. This option is useful in some debugging situations and might also enhance performance as it prevents unnecessary garbage collection from happening.

Operation

Format: `-XXnoSystemGC`

Simply enter the command in the above format at startup. This option can cause longer garbage collection pauses in some cases, but generally, it makes the application perform better.

Flags or Other Options Affected

This option will also cause `-XXprintSystemGC` to print out different information than if this option is not used.

Exceptions

You cannot use `-XXnoSystemGC` together with `-XXfullSystemGC`.

-XXoptThreads

This option tells the JVM how many threads to use for the optimizing methods. The work of optimizing methods run in the background.

Operation

Format: `-XXoptThreads:<# threads>`

Enter this option at the command line.

For Example:

`java -XgcPrio:pausetime` **`-XXoptThreads:3`** *`myApp`*

tells the garbage collector to use three threads for the optimizing methods.

Default Value

If `-XXoptThreads` is not specified, one thread is used for the optimizing methods.

Other Options or Flags Affected

None

Exceptions

This option is valild only from JRockit JVM 5.0 P26.0.0 and JRockit JVM 5.0 R26.4.0.

-XXpointerMatrixLinearSeekDistance

WARNING: This is an advanced tuning option. You should only use it if you understand how it works and are prepared to accept the consequences of its misuse.

This option sets the linear seek distance in the pointer matrix. The pointer matrix is a data structure used for storing references during compaction if `-XgcPrio:deterministic` or `-XgcPrio:pausetime` is used, or if `-XXusePointerMatrix` is set. Decreasing the linear seek distance increases JRockit JVM's memory footprint, but can, in some cases, increase the application throughput.

Operation

Format: `-XXpointerMatrixLinearSeekDistance:<distance>`

For Example:

`java -XXpointerMatrixLinearSeekDistance:5 myApp`

sets the linear seek distance to 5

Default Value

The default value is 10

Flags or Other Options Affected

None

Exceptions

This option works only when `-XgcPrio:deterministic` or `-XgcPrio:pausetime` is used, or if `-XXusePointerMatrix` is set.

-XXprintSystemGC

This option causes printing of the thread ID of the thread requesting the garbage collection. This option provides information about the frequency with which `System.gc()` is invoked. The printout is different if using `-XXnoSystemGC`. With that option, the thread ID is not printed; instead, the only information appearing indicates that `System.gc()` was called but the request was denied.

Operation

Format: `-XXprintSystemGC`

Entering this option as shown above causes the threadID to be printed out whenever `System.gc()` is invoked.

Flags or Other Options Affected

None

Exceptions

None

-XXsetGC

This option turns off the dynamic garbage collector and sets the static garbage collector of your choice. Applications with static behavior and work load might benefit from having a static garbage collector instead of a dynamic one. In such cases this option would be beneficial. This option offers more garbage collector mode selections than `-Xgc`.

Operation

Format: `-XXsetGC:[gen/single|par/con|par/con]` *myApp*

You can choose a garbage collector that is either generational or single spaced with a parallel or a concurrent mark and uses either a parallel sweep or a concurrent sweep.

- Generational Garbage Collection

  During a two-generational garbage collection, the heap is divided into two sections: an old generation and a young generation—also called the “nursery.” Objects are allocated in the nursery and when it is full, the JVM stops all Java threads and moves the live objects from the nursery, young generation, to the old generation.

- Single-spaced Garbage Collection

  The single-spaced option of garbage collection means that all objects live out their lives in a single space on the heap, regardless of their age. In other words, a single-spaced garbage collector does not have a nursery.

- Concurrent Mark/Sweep Algorithm

  The concurrent garbage collection algorithm does its marking and sweeping “concurrently” with all other processing; that is, it does not stop Java threads to do the complete garbage collection.

- Parallel Garbage Collection Mark/Sweep Algorithm

  The parallel garbage collection algorithm stops Java threads when the heap is full and uses every CPU to perform a complete mark and sweep of the entire heap. A parallel garbage collector can have longer pause times than concurrent garbage collectors, but it maximizes application throughput. Even on single CPU machines, this maximized performance makes parallel the recommended garbage collector, provided that your application can tolerate the longer pause times.

For Example:

The command:

`java` **`-XXsetGC:genparcon`** *`myApp`*

sets the garbage collection option to generational (two-spaced) with a parallel mark algorithm and a concurrent sweep algorithm.

The command:

`java` **`-XXsetGC:singleconpar`** *`myApp`*

sets the garbage collection option to single-spaced with a concurrent mark algorithm and a parallel sweep algorithm.

WARNING: Use this option **only** if you know the different effects of different garbage collection strategies. The garbage collector modes offered by `-Xgc` are sufficient for most applications.

Flags or Other Options Affected

When you specify `-XXsetGC`, the following options are affected:

- Setting `-XXsetGC` will override `-Xgc` and vice versa
- Setting `-XsetGC` will override part of the effect of `-server` and `-client`.

Exceptions

When using `-XXsetGC`, be aware of the following exceptions:

- If you set a static garbage collector, you will not be able to fully use the management API; that is, some functions of the API will not be available.
- You cannot use `-XXsetGC` together with any of the following options:
  - `-XgcPrio`
  - `-XpauseTarget`

-XXstaticCompaction

This option sets a static compact ratio and simple “sliding window” heuristics for compaction area choice.

Compaction is the process of moving live objects closer together in the Java heap to create larger free areas that can be used for allocation of large objects. JRockit JVM uses partial compaction, where only a small part of the Java heap is compacted each garbage collection. The default heuristics for choosing the compaction area size and position aim at keeping the compaction pause times even. If the application isn't sensitive to pausetimes, a static compaction area size and simple “sliding window” compaction area choice may increase performance.

Operation

Format: `-XXstaticCompaction`

Compaction occurs in a “sliding window” scheme, meaning that only part of the heap is compacted during each garbage collection. How much of the heap that is compacted each time can be specified using `-XXcompactRatio`. By default about 6%, or 8/128, of the heap is compacted each time. This means that in the first garbage collection parts 1-8 are compacted, in the second, parts 9-16 are compacted, and so on. After compacting parts 120-128 the “window” starts over at part 1-8. The compaction type for static mode is external (evacuation).

For Example:

- `java` **`-XXstaticCompaction`** *`myApp`*

  Specifies static compaction.

- `java` **`-XXstaticCompaction`** `-XXcompactRatio:10` *`myApp`*

  Specifies static compaction and that 10% of the Java heap should be compacted each garbage collection.

Flags or Other Options Affected

You can use `-XXcompactRatio` to specify the size of the compaction area.

Exceptions

When using `-XstaticCompaction`, be aware of the following exceptions:

- The compaction area may temporarily increase if object allocation is failing due to fragmentation on the Java heap.

- You cannot use `-XXstaticCompaction` together with the following options:
  - `-XXnoCompaction`
  - `-XXfullCompaction`
  - `-XXthroughputCompaction`
  - `-XXinternalCompactRatio`
  - `-XXexternalCompactRatio`

-XXthroughputCompaction

This option adjusts the compaction ratio dynamically, based upon the ratio of live data in the heap. This option can improve application throughput for applications with a high allocation rate but low ratio of live data.

Operation

Format: `-XXthroughputCompaction`

Enter this option at the command line.

Other Flags or Options Affected

When using `-XXthroughputCompaction`, be aware of the following:

- This option should not be used together with, `-XXfullCompaction`, `-XXcompactRatio`, `-XXinternalCompactRatio`, or `-XXexternalCompactRatio` as these options limit the compaction ratio in other ways.
- If used together with `-XgcPrio:pausetime`, `-Xgcprio:deterministic`, or `-XpauseTarget` the pausetimes may become too long and non-deterministic.

Exceptions

When using `-XXthroughputCompaction`, be aware of these exceptions:

- This option is valild only on JRockit JVM 5.0 P26.0.0 and later.
- You cannot use `-XXthroughputCompaction` together with the following options:
  - `-XXnoCompaction`
  - `-XXstaticCompaction`

-XXtlaSize

Sets the thread-local area size.

To increase performance JRockit JVM uses thread-local areas (TLA) for object allocation. This option can be used to tune the size of the thread-local areas, which may affect performance.

Operation

Format: `-XXtlaSize:<param>=<size>[k|K][m|M][g|G]`

Where <*param*> is one of the parameters listed in Table 3-5.

Table 3-5 -XXtlaSize Parameters

Parameter	Description
`min=<size>`	Sets the minimum size of a TLA.
`preferred=<size>`	Sets the preferred size of a TLA. The system will try to get TLAs of this size if possible, but will accept TLAs down to the minimum size, if that's what's available. Occasionally, a TLA might get larger than the preferred size, too. The preferred size must not be lower than the minimum size.
`fixed=<size>`	Sets TLAs to a fixed size. This is equivalent to setting both `min` and `preferred` to the same value, and mimics the behavior in older versions of the JRockit JVM. You cannot combine `fixed` with any of `min` or `preferred`.

There are no upper or lower limits to `-XXtlaSize`. The default value is 2KB.

Use this option with caution, as changing the thread-local area size can have severe impact on the performance.

Specify <*size*> in bytes, using the normal `k`,`M`,`G` suffixes.

For example:

```
-XXtlasize:min=2k,preferred=16k
```

sets the default for large heaps.

```
-XXtlasize:min=8k,preferred=512k
```

set a TLA size suitable for heaps of several GB.

Note: The old style of setting TLA size (that is, `-XXtlasize=256k`) is still supported but has been deprecated. If you use the old style, JRockit JVM will interpret the option as if the `fixed` parameter was used; for example, `-XXtlasize=256k` would be interpreted as `-XXtlasize:fixed=256k`.

Default Value

The default value for the minimum size is 2 kB. The minimum value cannot be lower than the large object limit. If the large object limit is explicitly set higher than the minimum TLA size (with `-XXlargeObjectLimit`), then the minimum TLA size will be raised to match the large object limit.

The default value for the preferred size depends on the heap size or the nursery size and the garbage collector selected at startup.Table 3-6 lists the default sizes for the different configurations.

Table 3-6 Default Preferred TLA Sizes

Releases	Garbage Collectors	Default Preferred TLA Size
R26.4 and older	All	2 kB
R27.1 - R27.2	All	2 kB - 16 kB depending on the heap size.
R27.3 and later	`-XgcPrio:deterministic`, `-Xgc:singlecon`	16 kB
R27.3 and later	`-XgcPrio:pausetime`, `-Xgc:gencon`	16 kB - 256 kB depending on the nursery size
R27.3 and later	`-XgcPrio:throughput`, `-Xgc:genpar`	16 kB - 64 kB depending on the heap size
R27.3 and later	`-Xgc:parallel`	16 kB - 256 kB depending on the heap size
R27.3 and later	Nursery size set with `-Xns`	16 kB - 256 kB depending on the nursery size

Flags or Other Options Affected

When you set the minimum and/or the preferred TLA size, the large object limit (set with `-XXlargeObjectLimit`) and the minimum block size (set with `-XXminBlockSize`) may be adjusted automatically by JRockit JVM if necessary. At all times, the following relations are maintained between minimum and preferred TLA size, large object limit, and minimum block size:

```
-XXlargeObjectLimit <= -XXtlaSize:min <= -XXminBlockSize
-XXtlaSize:min <= -XXtlaSize:preferred
```

If you set two or more of the options, then you must make sure that the values you use fulfil these criteria. By default, the large object limit will be set to whichever is the lower value of the minimum TLA size and the preferred TLA size divided by 2. The default minimum block size is 2k.

It is recommended to that you primarily set the TLA size parameters for memory management tuning purposes, while you let JRockit JVM automatically adjust the large object limit and minimum block size if necessary.

Exceptions

None.

-XXtsf

This option enables the trace scheduler framework (TSF) on 64-bit Intel Itanium systems.

Note: This option replaces the `-Djrockit.codegen.tracesched=<true|false> argument` available in previous releases of the JRockit JVM.

Operation

Format: `-XXtsf=<true|false>`

To enable the trace scheduler framework, enter:

`java` **`-Xtsf=true`** *myApp*

The following line will be logged when TSF is enabled:

```
[INFO ] Trace scheduling is enabled.
```

Default Value

If `-XXtsf=` is not specified or if it is set to `false`, the trace scheduler framework feature is disabled.

Flags or Other Options Affected

None

Exceptions

This feature is only available on 64-bit Intel Itanium systems.

-XXusePointerMatrix

WARNING: This is an advanced tuning option. You should only use it if you understand how it works and are prepared to accept the consequences of its misuse.

This option indicates that the pointer matrix should be used instead of the pointerset. The pointer matrix is default when running `-Xgcprio:deterministic` or `-Xgcprio:pausetime`.

Operation

Format: `-XXusePointerMatrix`

For Example:

`java` **`-XXusePointerMatrix`** *`myApp`*

Flags or Other Options Affected

You should not use this option together with `-XgcPrio:throughput`, as it alters or disables many of the throughput-specific compaction heuristics.

Exceptions

You cannot use this option together with `-XXcompactSetLimit`.

-XX:MaximumNurseryPercentage

This option lets you set an upper nursery size limit in that will be relative to the free heap space available after the latest old collection. Do this by specifying the limit as a percentage value of the available free heap size.

Operation

Format: `-XX:MaximumNurseryPercentage=<value> [1-95]`

If you try to set the upper nursery size limit to a value lower than 1 or higher than 95, you will get an error message.

For Example:

`java` **`-XX:MaximumNurseryPercentage=80`** *`myApp`*

Default Value

The default value is 95.

Exceptions

You cannot use this option with a single-spaced (non-generational) garbage collector. The option can be used both with a static generational garbage collector or with dynamic garbage collection.

-XX:(+|-)UseNewHashFunction

This option enables a new, faster hash function for HashMap that was introduced in Java 5.0 Update 8 and is part of the JRockit JVM as of R27.1.0. This hash function can improve performance through improved hash spread, but changes the order in which elements are stored in the HashMap. For compatibility reasons, JRockit JVM 5.0 uses the old hash function by default unless started with `-XXaggressive`.

Note: This flag is supported as of JRockit JVM 5.0 R27.1. It is not available in JRockit JVM 1.4.2.

Operation

Format: `-X:[+|-]useNewHashFunction`

This option uses Sun's implementation format; that is, you must place the colon (:) between the `-XX` and the option name followed by a the necessary operator to indicate enabling (+) or disabling (-) the new hash function.

For Example:

```
-XX:+UseNewHashFunction
```

Explicitly enables the new hash function.

```
-XX:-UseNewHashFunction
```

Explicitly disables the new hash function.

Default Value

The new hash function is disabled by default in the JRockit JVM 5.0.

Flags or Other Options Affected

`-XXaggressive` enables use of the new hash function unless it is explictly disabled using `-XX:-UseNewHashFunction`.

Exceptions

None

-XX:(+|-)UseThreadPriorities

This option enables you to control the priority of Java threads using `java.lang.Thread.setPriority()` and related APIs. If this feature is disabled, using these APIs has no effect.

WARNING: This feature is experimental and not supported by Oracle at this time. Improper use can cause serious performance issues.

Operation

Format: `-XX:[+|-]UseThreadPriorities`

This option uses Sun's implementation format; that is, you must place the colon (`:`) between the `-XX` and the option name followed by a the necessary operator to indicate enabling (`+`) or disabling (`-`) the use of `java.lang.Thread.setPriority()` and related APIs.

For example:

`-XX:+UseThreadPriorities`

Explicitly enables use of the APIs.

`-XX:-UseThreadPriorities`

Explicitly disables use of the APIs.

Default Value

`-XX:-UseThreadPriorities`. Thread priorities are disabled by default.

Flags or Other Options Affected

None

Exceptions

Availability of this option is determined by the platform you are running on.

- Windows: Available in JRockit JVM R26.0.0 and later.
- Linux: Available in Oracle JRockit JVM R26.4.0 and later. You must have root privileges to use thread priorities on most Linux versions.
- Solaris: Not available.

CHAPTER 4

Oracle JRockit JVM System Properties

The `System` class maintains a set of properties—key/value pairs—that define traits or attributes of the current working environment. When the Java application first starts up, the system properties are initialized with information about the runtime environment, including information about the current user, the current version of the Java runtime, and even the product vendor's bug report URL.

This chapter describes the key system properties available with the Oracle JRockit JVM:

- java.vendor
- java.vendor.url
- java.vendor.url.bug
- java.version
- java.runtime.version
- java.vm.name
- java.vm.vendor
- java.vm.vendor.url
- java.vm.version
- java.vm.specification.version
- java.vm.specification.vendor

- java.vm.specification.name
- os.name
- os.arch
- os.version

As described above, system properties are part of the `System` class as defined by J2SE 1.4.2 and 5.0. You can obtain these properties in a Java class simply by calling the `getProperty()` method. Listing 4-1 shows how the `getProperty()` method is used to get system properties.

Listing 4-1 Obtaining System Properties

```
String os_name        =   System.getProperty("os.name");
String os_arch        =   System.getProperty("os.arch");
String java_home      =   System.getProperty("java.home");
String java_vm_name   =   System.getProperty("java.vm.name");
```

java.vendor

This property identifies the J2SE JDK/JRE product vendor. This includes:

- Sun: “Sun Microsystems Inc.”
- JRockit: “Oracle”

For example:

```
Oracle
```

java.vendor.url

This property identifies the J2SE JDK/JRE product vendor URL; for example:

- Sun: “`http://java.sun.com/`”
- JRockit: “`http://www.bea.com/`”

java.vendor.url.bug

This property identifies the J2SE JDK/JRE product vendor bug report URL. This includes:

- Sun: "`http://java.sun.com/cgi-bin/bugreport.cgi`"
- JRockit: "`http://support.bea.com`"

java.version

This property identifies the J2SE JDK/JRE product version; that is, which version of the JDK or JSE you are running. The version number also appears on the first line of the output and is common to Sun JRE and Oracle JRockit JRE. The information generated by this property appears in this format:

```
<jdk_major_version>.<jdk_minor_version>.<jdk_micro_version>[_<jdk_update_versi
on>][-<milestone>]
```

The value appears in **bold** in the example version output below:

```
java version "1.5.0_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_03-b07)
BEA JRockit(R) (build dra-45238-20050523-2021-win-ia32, R25.2.0-28)
```

For more information on `java.version`, please refer to:

http://java.sun.com/j2se/versioning_naming.html

java.runtime.version

This property identifies the J2SE JDK/JRE product version and build identifier. The value also appears on the second line of the `java -version` output in the following format:

```
<jdk_major_version>.<jdk_minor_version>.<jdk_micro_version>[_<jdk_update_version>][-<milestone>]-<build_number>
```

The value appears in **bold** in the example version output below:

```
java version “1.5.0_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_03-b07)
BEA JRockit(R) (build dra-45238-20050523-2021-win-ia32, R25.2.0-28)
```

This value is common to both the Sun JRE and JRockit JRE.

For more information, please refer to:

http://java.sun.com/j2se/versioning_naming.html

java.vm.name

This property identifies the JVM implementation. It appears on the third line of `java -version` output. Depending upon the JVM you are using, the VM name will appear as either:

- Sun: "`Java HotSpot(TM) Client VM`" "`Java HotSpot(TM) Server VM`"

or

- JRockit: "JRockit`(R)`"

The value appears in **bold** in the example version output below:

```
java version "1.5.0_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_03-b07)
BEA JRockit(R) (build dra-45238-20050523-2021-win-ia32, R25.2.0-28
```

java.vm.vendor

This property identifies the JVM implementation vendor; for example:

- Sun: “Sun Microsystems Inc.”
- JRockit: “Oracle”

java.vm.vendor.url

This property identifies the JVM implementation vendor URL; for example:

- Sun: "`http://java.sun.com/`"
- JRockit: "`http://www.bea.com/`"

java.vm.version

This property identifies the JVM implementation version. The version ID appear on the third line of `java -version` output. This ID is the main method to distinguish between JRockit JVM versions. Below are examples from a few different releases of the JRockit JVM

- JRockit JVM R24.5.0: `ari-49095-20050826-1856-win-ia32`
- JRockit JVM 5.0 SP2: `dra-45238-20050523-2021-win-ia32`
- JRockit JVM R26.4.0: `R26.4.0-63-63688-1.5.0_06-20060626-2259-win-ia32`
- JRockit JVM R27.3.1: `R27.3.1-1-85830-1.6.0_01-20070716-1248-windows-ia32`

The value appears in **bold** in the example version output below:

```
java version "1.5.0_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_03-b07)
BEA JRockit(R) (build dra-45238-20050523-2021-win-ia32, R25.2.0-28)
```

java.vm.specification.version

This property identifies the version of the Java Virtual Machine specification upon which your JRockit JVM instance is based.

For example:

```
1.0
```

java.vm.specification.vendor

This property identifies the vendor of the Java Virtual Machine specification upon which your JRockit JVM instance is based.

For example:

```
Sun Microsystems Inc
```

java.vm.specification.name

This property identifies the name of the Java Virtual Machine specification upon which your JRockit JVM instance is based.

For example:

```
Java Virtual Machine Specifications
```

os.name

This property identifies the operating system. For the JRockit JVM, this includes:

- Windows versions
- Linux versions
- Solaris

For example:

```
Windows XP
```

For additional information on O/S support, please refer to Oracle JRockit JDK *Supported Configurations* at

`http://edocs.bea.com/jrockit/jrdocs/suppPlat/supp_plat.html`

os.arch

This property identifies the operating system architecture. For Oracle JRockit JVM, this includes:

- x86 on IA32
- amd64 on AMD64
- ia64 on Itanium
- sparcv9 on Solaris SPARC

For example:

```
x86
```

For additional information on supported architectures, please refer to Oracle JRockit JDK *Supported Configurations* at

`http://edocs.bea.com/jrockit/jrdocs/suppPlat/supp_plat.html`

os.version

This property identifies the operating system version. For example:

```
5.1
```

For additional information on O/S support, please refer to Oracle JRockit JDK *Supported Configurations* at

`http://edocs.bea.com/jrockit/jrdocs/suppPlat/supp_plat.html`