Oracle® JRockit JVM

Diagnostics Guide

R27.6

June 2008

ORACLE

Oracle JRockit JVM Diagnostics Guide, R27.6

Contents

Understanding Memory Management

Understanding Threads and Locks

Migrating Applications to the Oracle JRockit JDK

Setting Up and Running the Oracle JRockit JDK

Part II. Profiling and Performance Tuning

About Profiling and Performance Tuning

Understanding the Tuning Trade-offs

First Steps for Tuning the Oracle JRockit JVM

Tuning the Memory Management System

Tuning Locks

Tuning For Low Latencies

Tuning For Better Application Throughput

Tuning For Stable Performance

Tuning For a Small Memory Footprint

Tuning For Faster JVM Startup

Part III. JRockit JDK Tools

Introduction to Diagnostics Tools

Using Oracle JRockit Mission Control Tools

Understanding Verbose Outputs

Using Thread Dumps

Running Diagnostic Commands

Oracle JRockit Time Zone Updater

Oracle JRockit Mission Control Use Cases

Part IV. Diagnostics and Troubleshooting

About Diagnostics and Troubleshooting

Diagnostics Roadmap

The Oracle JRockit JVM Starts Slowly

Long Latencies

Low Overall Throughput

The Oracle JRockit JVM's Performance Degrades Over Time

The System is Crashing

Understanding Crash Files

The Oracle JRockit JVM is Freezing

Submitting Problems to Oracle Support

Part I Understanding the Oracle JRockit JDK

CHAPTER 1

About the Oracle JRockit JDK

Note: The information in the Diagnostics Guide is only applicable to the Oracle JRockit JDK R26 and later versions.

The Oracle JRockit JDK provides tools, utilities, and a complete runtime environment for developing and running applications using the Java programming language. The JRockit JDK includes the Oracle JRockit Java Virtual Machine (JVM). The Oracle JRockit JVM is developed and optimized for Intel architectures to ensure reliability, scalability, and manageability for Java applications.

This section contains information on the following subjects:

What is the JRockit JVM?

The JRockit JVM is a high performance JVM developed to ensure reliability, scalability, manageability, and flexibility for Java applications. The JRockit JVM delivers a new level of performance for Java applications deployed on Intel 32-bit (Xeon) and 64-bit (Xeon, Itanium, and SPARC) architectures at significantly lower costs to the enterprise. Furthermore, it is the only enterprise-class JVM optimized for Intel architectures, providing seamless inter operability across multiple hardware and operating configurations. The JRockit JVM makes it possible to gain optimal performance for your Java applications when running it on either the Windows or Linux operating platforms on either 32-bit or 64-bit architectures. The JRockit JVM is especially well suited for running Oracle WebLogic Server.

For more information on JVMs in general, see the Introduction to the JVM Specification at:

`http://java.sun.com/docs/books/vmspec/2nd-edition/html/Introduction.doc.html#3057`

About the JDK

The JRockit JVM is one component of the Oracle JRockit Java development kit (JDK). In addition to the JRockit JVM, the JDK is comprised of the Java Runtime Environment (JRE), which contains the JVM and Java class libraries (as specified by the Java Platform, Standard Edition 6 API Specification), as well as a set of development tools, such as a compiler. For more information about the contents of the JRockit JDK, please refer to The Contents of a JRockit JDK Installation.

JRockit JDK Versions

The JRockit JDK numbering scheme is based upon:

- Java SE version (J2SE 1.4.2, J2SE 5.0, Java SE 6.0)
- The JRockit JVM release number (R*nn.nn.nn*)

For example, Oracle JRockit JDK **6.0 R27.6** refers to the 27.6 release of JRockit JVM used with Java SE 6.0; Oracle JRockit JDK **1.4.2 R27.6** refers to the 27.6 release of the JRockit JVM used with J2SE 1.4.2. All future versions of the JRockit JDK will follow this versioning scheme.

A full version name might look like this:

```
R27.6.0-1-85830-1.6.0_01-20070716-1248-windows-ia32
```

where `R27.6.0 is` the JRockit JVM release, `1.6.0_01` is the Java version, and `windows-ia32` is the platform on which this version runs.

Note: JRockit JDK versions that were based on J2SE releases earlier than 1.4.2 used a different numbering scheme following the Oracle WebLogic Platform versions. For this reason, the J2SE 1.3.1 version of the JRockit JDK was called 7.0.

Every JRockit JVM release comes with several Java versions. For example, JRockit JVM R27.6 comes with Java SE versions 1.4.2, 5.0, and 6. A Java version can be compatible with multiple JRockit JVM releases.

What Platforms Does the JRockit JDK Support?

The JRockit JDK is certified to be compatible with J2SE 1.3.1, 1.4.2, 5.0, and Java SE 6.0. For a complete list of platforms that the JRockit JDK supports, please refer to JRockit JDK *Supported Configurations* at:

http://e-docs.bea.com/jrockit/jrdocs/suppPlat/supp_plat.html

Compatibility Information

The JRockit JDK subscribes to an update policy that ensures compatibility from one release to the next to provide simple and complete upgrade flexibility. This policy is described in Compatibility Between Releases at:

http://e-docs.bea.com/jrockit/jrdocs/suppPlat/prodsupp.html#999010

The Contents of a JRockit JDK Installation

This section describes the various components that make up an installation of the JRockit JDK. It also identifies the folders in which the components reside.

The JRockit JDK is very similar, in the file layout, to the Sun JDK, except that it includes a new JRE with the JRockit JVM and some changes to the Java class libraries (however, all of the class libraries have the same behavior in the JRockit JDK as in the Sun JDK).

The following sections briefly describe the contents of the directories in a JRockit JDK installation:

- Development Tools (`/bin`)
- Demo (`/demo`)

- C Header Files (`/include`)
- Java Runtime Environment (JRE) (`/jre`)
- Additional Libraries (`/lib`)
- Sample (`/sample`)

Development Tools

Found in: `/bin`

Development tools and utilities help you develop, execute, debug, and document programs written in the Java programming language. The JRockit JDK includes the standard tools commonly distributed with the typical Java JDKs. While most of these are standard JDK tools and are proven to work well with Java development projects, you are free to use any other third party tools, compilers, debuggers, IDEs, and so on that might work best in your situation. The tools included with the JRockit JDK are:

- Javac compiler
- Jdb debugger
- Javadoc, which is used to create an HTML documentation site for the JVM API

For more information on these tools, please refer to Sun Microsystem's Java SE 6 Development Kit at:

`http://java.sun.com/javase/6/`

Demo

Found in: `/demo`

This directory contains various demos of how to use various libraries included in the JRockit JDK installation.

C Header Files

Found in: `/include`

Header files that support native-code programming using the Java Native Interface (JNI) and the Java Virtual Machine Tools Interface (JVMTI) and other functionality of the Java SE Platform.

Java Runtime Environment (JRE)

Found in: `/jre`

The JRockit JVM implementation of the Java runtime environment. The runtime environment includes the JRockit JVM, class libraries, and other files that support the execution of programs written in Java.

Java Virtual Machine

By definition, the JVM is the JRockit JVM, as described in this documentation set.

Standard Java SE JRE Features

In addition to JRE components specific to the JRockit JDK, the JRE also contains components found in the Sun implementation of the JRE. For a complete list of the standard Java SE JRE features, see the Sun documentation for the specific version of JRockit JDK you are running:

- JRockit JDK 6.0 R27.2 and higher:

  `http://java.sun.com/javase/6/docs/index.html`

- JRockit JDK 5.0 R25 and higher:

  `http://java.sun.com/j2se/1.5.0/docs/index.html`

- JRockit JDK 1.4.2 R26 and higher:

  `http://java.sun.com/j2se/1.4.2/docs/index.html`

Note on JRE Class Files

The JRE class files distributed with the JRockit JDK come directly from Sun, except for a small number that are tightly coupled to the JVM and are therefore overridden in the JRockit JDK. The overridden class files are in the `java.lang`, `java.io`, `java.net`, and `java.util` packages. No classes have been omitted.

Additional Libraries

Found in: `/lib`

Additional class libraries and support files required by the development tools.

Sample

Found in: `/sample`

The Sample directory contains the source files for a simple NIO-based HTTP/HTTPS Server written in Java. The server was written to demonstrate some of the functionality of the Java 2 platform. The demo is not meant to be a full tutorial, it assumes that you have some familiarity with the subject matter.

Attach API Support

Versions of the JRockit JVM running on Java 6 support the Attach API. This API is a Java extension that provides a way to attach tools written in Java to JRockit JVM and load their tool agents into it. For example, a management console might use a management agent to obtain objects in the JRockit JVM instance. If the management console has to manage an application running in a JRockit JVM instance that doesn't include the management agent, you can use this API to attach to the JRockit JVM instance and load the agent.

For more information, please see the Attach API specification at:

`http://java.sun.com/javase/6/docs/jdk/api/attach/spec/index.html`

Oracle JRockit Documentation

The Oracle JRockit JVM Diagnostics Guide is a general document applicable to the R27 release and all subsequent JRockit JDK releases.

For links to all documentation available for the latest version of the JRockit JDK, visit the Oracle JRockit documentation page at the following location:

`http://edocs.bea.com/jrockit/webdocs/index.html`

From this page, you can also access the documentation for earlier versions of the Oracle JRockit JDK.

You can find documentation for the Oracle JRockit Mission Control tools at the following location:

`http://edocs.bea.com/jrockit/tools/index.html`

JRockit JVM Command Line Options

The Oracle JRockit JVM configuration and tuning parameters are set by using specific command line options, which you can enter either along with the start-up command or include in a start-up script. These options are discussed in the Oracle JRockit *Command Line Reference*, at:

`http://edocs.bea.com/jrockit/jrdocs/refman/index.html`

JRockit JDK and JRockit Mission Control Support

You are entitled to support on the JVM and JRockit Mission Control if you have a support agreement with Oracle.

CHAPTER 2

Understanding JIT Compilation and Optimizations

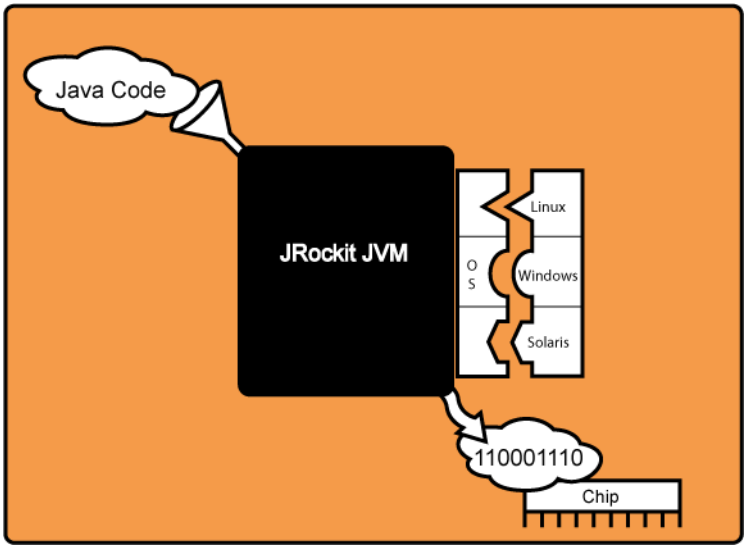
This section offers a high-level look at how the Oracle JRockit JVM generates code. It provides information on JIT compilation and how the JVM optimizes code to ensure high performance. This section contains information on the following subjects:

- More than a “Black Box”
- How the JRockit JVM Compiles Code
- An Example Illustrating Some Code Optimizations

More than a “Black Box”

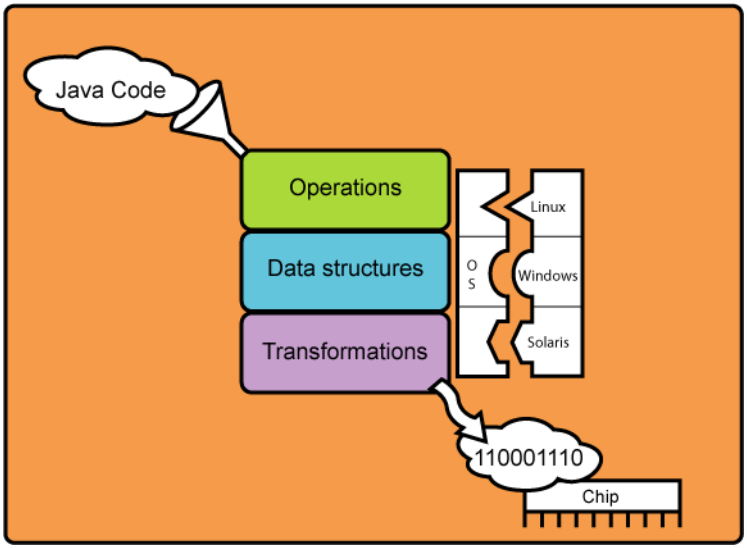
From the user’s point of view, the JRockit JVM is merely a black box that “converts” Java code to highly optimized machine code: you put Java code in one end of the JVM and out the other end comes machine code for your particular platform (see Figure 2-1).

Figure 2-1 The JRockit JVM as a Black Box



When lifting the lid of the black box you will see different actions that are taken before the code is optimized for your particular operating system. There are certain operations, data structure changes, and transformations that take place before the code leaves the JVM (see Figure 2-2).

Figure 2-2 Lifting the Black Box

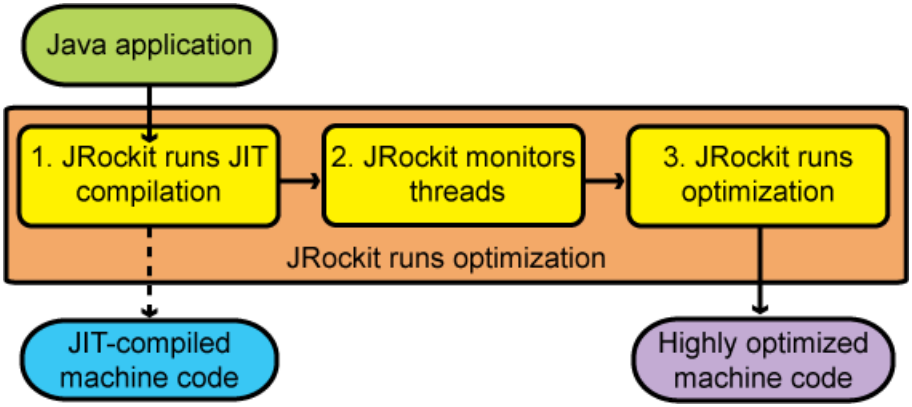


This section sheds some light on what actually happens with the Java application code when going through the JVM.

How the JRockit JVM Compiles Code

The code generator in the JRockit JVM runs in the background during the entire run of your Java application, automatically adapting the code to run its best. The code generator works in three steps, as described in Figure 2-3.

Figure 2-3 How the JRockit JVM Optimizes Code for Your Java Application



1. The JRockit JVM Runs JIT Compilation

The first step of code generation is the Just-In-Time (JIT) compilation. This compilation allows your Java application to start and run while the code that is generated is not highly optimized for the platform. Although the JIT is not actually part of the JVM standard, it is, nonetheless, an essential component of Java. In theory, the JIT comes into use whenever a Java method is called, and it compiles the bytecode of that method into native machine code, thereby compiling it "just in time" to execute.

After a method is compiled, the JRockit JVM calls that method's compiled code directly instead of trying to interpret it, which makes the running of the application fast. However, during the beginning of the run, thousands of new methods are executed, which can make the actual start of the JRockit JVM slower than other JVMs. This is due to a significant overhead for the JIT to run and compile the methods. So, if you run a JVM without a JIT, that JVM starts up quickly but usually runs slower. If you run the JRockit JVM that contains a JIT, it can start up slowly, but then runs quickly. At some point, you might find that it takes longer to start the JVM than to run an application.

Compiling all of the methods with all available optimizations at startup would negatively impact the startup time. Thus the JIT compilation does not fully optimize all methods at startup.

2. The JRockit JVM Monitors Threads

During the second phase, the JRockit JVM uses a sophisticated, low-cost, sampling-based technique to identify which functions merit optimization: a "sampler thread" wakes up at periodic intervals and checks the status of several application threads. It identifies what each thread is

executing and notes some of the execution history. This information is tracked for all the methods and when it is perceived that a method is experiencing heavy use—in other words, is “hot”—that method is earmarked for optimization. Usually, a flurry of such optimization opportunities occur in the application’s early run stages, with the rate slowing down as execution continues.

3. The JRockit JVM Runs Optimization

During the third phase, the JVM runs an optimization round of the methods that it perceives to be the most used—“hot”—methods. This optimization is run in the background and does not disturb the running application.

An Example Illustrating Some Code Optimizations

This example illustrates some ways in which the JRockit JVM optimizes Java code. The example is fairly short and simple, but it will give you a general idea of how the actual Java code can be optimized. Note that there are many ways of optimizing Java applications that are not discussed here.

In Table 2-1 you can see how the code before and after optimization. The differences might not look substantial, but note that the optimized code does not need to run down to Class B every time Class A is run.

Table 2-1 Example of before and after optimization of a class

Class A before optimization	Class A after optimization

`class A {`
`B b;`
`public void foo() {`
`  y = b.get();`
`  ...do stuff...`
`  z = b.get();`
`  sum = y + z;`
`}`
`}`
 	
`class B {`
`  int value;`
`  final int get() {`
`  return value;`
`  }`
`}`
	`class A {`
	`B b;`
	`public void foo() {`
	`  y = b.value;`
	`  ...do stuff...`
	`  sum = y + y;`
	`}`
	`}`
	
	`class B {`
	`  int value;`
	`  final int get() {`
	`  return value;`
	`  }`
	`}`

Steps Taken to Optimize Class A

When the Oracle JRockit JVM optimizes code it goes through several steps to get the best optimization possible. The example from Table 2-1 shows on how a method looks like before and after the optimization. In Table 2-2 you find an explanation of what can happen in a few optimization steps that the JVM might go through at the level of the Java application code itself. Note that several optimizations appear at the level of the assembler code, however.

Table 2-2 Different Optimization Steps

Step in Optimization	Code Transformation	Comment

Starting point	`public void foo() {`
	`  y = b.get();`
	`  ...do stuff...`
	`  z = b.get();`
	`  sum = y + z;`
	`}`	

1. Inline final method	`public void foo() {`
	`  *y = b.value;*`
	`  ...do stuff...`
	`  *z = b.value;*`
	`  sum = y + z;`
	`}`
		`// b.get() has been replaced by b.value`
		`// as latencies are reduced by accessing`
		`// b.value directly instead of through`
		`// a function call.`

2. Remove redundant loads	`public  void foo() {`
	`  y = b.value;`
	`  ...do stuff...`
	`  *z = y;*`
	`  sum = y + z;`
	`}`
		`// z = b.value has been replaced with`
		`// z = y so that latencies will be`
		`// reduced by accessing the local value`
		`// instead of b.value.`

3. Copy propagation	`public  void foo() {`
	`  y = b.value;`
	`  ...do stuff...`
	`  *y = y;*`
	`  *sum = y + y;*`
	`}`
		`// z = y has been replaced by y = y since`
		`// there is no use for an extra variable`
		`// z as the value of z and y will be`
		`// equal.`

4. Eliminate dead code	`public  void foo() {`
	`  y = b.value;`
	`  ...do stuff...`
	`  sum = y + y;`
	`}`
		`// y = y is unnecessary and can be`
		`// eliminated.`

CHAPTER 3

Understanding Memory Management

Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations. This section presents some basic memory management concepts and explains the basics about object allocation and garbage collection in the Oracle JRockit JVM. The following topics are covered:

- The Heap and the Nursery
- Object Allocation
- Garbage Collection

For information about how to use command line options to tune the memory management system, see Tuning the Memory Management System.

The Heap and the Nursery

Java objects reside in an area called *the heap*. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. When the heap becomes full, *garbage* is *collected*. During the garbage collection objects that are no longer used are cleared, thus making space for new objects.

Note that the JVM uses more memory than just the heap. For example Java methods, thread stacks and native handles are allocated in memory separate from the heap, as well as JVM internal data structures.

The heap is sometimes divided into two areas (or *generations*) called the *nursery* (or *young space*) and the *old space*. The nursery is a part of the heap reserved for allocation of new objects.

When the nursery becomes full, garbage is collected by running a special *young collection*, where all objects that have lived long enough in the nursery are *promoted* (moved) to the old space, thus freeing up the nursery for more object allocation. When the old space becomes full garbage is collected there, a process called an *old collection*.

The reasoning behind a nursery is that most objects are temporary and short lived. A young collection is designed to be swift at finding newly allocated objects that are still alive and moving them away from the nursery. Typically, a young collection frees a given amount of memory much faster than an old collection or a garbage collection of a single-generational heap (a heap without a nursery).

In R27.2.0 and later releases, a part of the nursery is reserved as a *keep area*. The keep area contains the most recently allocated objects in the nursery and is not garbage collected until the next young collection. This prevents objects from being promoted just because they were allocated right before a young collection started.

Object Allocation

During object allocation, the JRockit JVM distinguishes between *small* and *large* objects. The limit for when an object is considered large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between 2 and 128 kB. Please see the documentation for `-XXtlaSize` and `-XXlargeObjectLimit` for more information.

Small objects are allocated in *thread local areas* (*TLA*s). The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. When the TLA becomes full, the thread simply requests a new TLA. The TLAs are reserved from the nursery if such exists, otherwise they are reserved anywhere in the heap.

Large objects that don't fit inside a TLA are allocated directly on the heap. When a nursery is used, the large objects are allocated directly in old space. Allocation of large objects requires more synchronization between the Java threads, although the JRockit JVM uses a system of caches of free chunks of different sizes to reduce the need for synchronization and improve the allocation speed.

Garbage Collection

Garbage collection is the process of freeing space in the heap or the nursery for allocation of new objects. This section describes the garbage collection in the JRockit JVM.

The Mark and Sweep Model

The JRockit JVM uses the *mark and sweep* garbage collection model for performing garbage collections of the whole heap. A mark and sweep garbage collection consists of two phases, the *mark phase* and the *sweep phase*.

During the mark phase all objects that are reachable from Java threads, native handles and other root sources are *marked* as alive, as well as the objects that are reachable from these objects and so forth. This process identifies and marks all objects that are still used, and the rest can be considered garbage.

During the sweep phase the heap is traversed to find the gaps between the live objects. These gaps are recorded in a *free list* and are made available for new object allocation.

The JRockit JVM uses two improved versions of the mark and sweep model. One is *mostly concurrent mark and sweep* and the other is *parallel mark and sweep*. You can also mix the two strategies, running for example mostly concurrent mark and parallel sweep.

Mostly Concurrent Mark and Sweep

The *mostly concurrent mark and sweep strategy* (often simply called *concurrent garbage collection*) allows the Java threads to continue running during large portions of the garbage collection. The threads must however be stopped a few times for synchronization.

The mostly concurrent mark phase is divided into four parts:

- *Initial marking*, where the root set of live objects is identified. This is done while the Java threads are paused.
- *Concurrent marking*, where the references from the root set are followed in order to find and mark the rest of the live objects in the heap. This is done while the Java threads are running.
- *Precleaning*, where changes in the heap during the concurrent mark phase are identified and any additional live objects are found and marked. This is done while the Java threads are running.

- *Final marking*, where changes during the precleaning phase are identified and any additional live objects are found and marked. This is done while the Java threads are paused.

The mostly concurrent sweep phase consists of four parts:

- Sweeping of one half of the heap. This is done while the Java threads are running and are allowed to allocate objects in the part of the heap that isn't currently being swept.
- A short pause to switch halves.
- Sweeping of the other half of the heap. This is done while the Java threads are running and are allowed to allocate objects in the part of the heap that was swept first.
- A short pause for synchronization and recording statistics.

Parallel Mark and Sweep

The parallel mark and sweep strategy (also called the *parallel garbage collector*) uses all available CPUs in the system for performing the garbage collection as fast as possible. All Java threads are paused during the entire parallel garbage collection.

Generational Garbage Collection

The nursery, when it exists, is garbage collected with a special garbage collection called a *young collection*. A garbage collection strategy which uses a nursery is called a g*enerational garbage collection strategy*, or simply *generational garbage collection*.

The young collector used in the JRockit JVM identifies and promotes all live objects in the nursery that are outside the keep area to the old space. This work is done in parallel using all available CPUs. The Java threads are paused during the entire young collection.

Dynamic and Static Garbage Collection Modes

By default, the JRockit JVM uses a dynamic garbage collection mode that automatically selects a garbage collection strategy to use, aiming at optimizing the application throughput. You can also choose between two other dynamic garbage collection modes or select the garbage collection strategy statically. The following dynamic modes are available:

- `throughput`, which optimizes the garbage collector for maximum application throughput. This is the default mode.
- `pausetime`, which optimizes the garbage collector for short and even pause times.

- `deterministic`, which optimizes the garbage collector for very short and deterministic pause times. This mode is only available as a part of Oracle JRockit Real Time.

The major static strategies are:

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

For more information on how to select the best mode or strategy for your application, see Selecting and Tuning a Garbage Collector.

Compaction

Objects that are allocated next to each other will not necessarily become unreachable ("die") at the same time. This means that the heap may become fragmented after a garbage collection, so that the free spaces in the heap are many but small, making allocation of large objects hard or even impossible. Free spaces that are smaller than the minimum thread local area (TLA) size can not be used at all, and the garbage collector discards them as *dark matter* until a future garbage collection frees enough space next to them to create a space large enough for a TLA.

To reduce fragmentation, the JRockit JVM compacts a part of the heap at every garbage collection (old collection). Compaction moves objects closer together and further down in the heap, thus creating larger free areas near the top of the heap. The size and position of the compaction area as well as the compaction method is selected by advanced heuristics, depending on the garbage collection mode used.

Compaction is performed at the beginning of or during the sweep phase and while all Java threads are paused.

For information on how to tune compaction, see Tuning the Compaction of Memory.

External and Internal Compaction

The JRockit JVM uses two compaction methods called *external compaction* and *internal compaction*. External compaction moves (evacuates) the objects within the compaction area to free positions outside the compaction area and as far down in the heap as possible. Internal compaction moves the objects within the compaction area as far down in the compaction area as possible, thus moving them closer together.

The JVM selects a compaction method depending on the current garbage collection mode and the position of the compaction area. External compaction is typically used near the top of the heap, while internal compaction is used near the bottom where the density of objects is higher.

Sliding Window Schemes

The position of the compaction area changes at each garbage collection, using one or two sliding windows to determine the next position. Each sliding window moves a notch up or down in the heap at each garbage collection, until it reaches the other end of the heap or meets a sliding window that moves in the opposite direction, and starts over again. Thus the whole heap is eventually traversed by compaction over and over again.

Compaction Area Sizing

The size of the compaction area depends on the garbage collection mode used. In throughput mode the compaction area size is static, while all other modes, including the static mode, adjust the compaction area size depending on the compaction area position, aiming at keeping the compaction times equal throughout the run. The compaction time depends on the number of objects moved and the number of references to these objects. Thus the compaction area will be smaller in parts of the heap where the object density is high or where the amount of references to the objects within the area is high. Typically the object density is higher near the bottom of the heap than at the top of the heap, except at the very top where the latest allocated objects are found. Thus the compaction areas are usually smaller near the bottom of the heap than in the top half of the heap.

CHAPTER 4

Understanding Threads and Locks

A running application is usually made up of one process with its own memory space. A computer is generally running several processes at the same time. For example, a word processor application process might be running alongside a media player application process. Furthermore, a process consists of many concurrently running threads. When you run a Java application, a new JVM process is started.

Each Java process has at least one application thread. Besides the threads of the running Java application, there are also Oracle JRockit JVM internal threads that take care of garbage collection or code generation.

This section contains basic information about threads and locks in the JRockit JVM. The following subjects are discussed:

- Understanding Threads
- Understanding Locks

For information about how to make so-called *thread dumps*, printouts of the stacks of all the threads in an application, see Using Thread Dumps. Thread dumps can be used to diagnose problems and optimize application and JVM performance.

Understanding Threads

A java application consists of one or more threads that run Java code. The entire JVM process consists of the Java threads and some JVM internal threads, for example one or more garbage collection threads, a code optimizer thread and one or more finalizer threads.

From the operating system's point of view the Java threads are just like any application threads. Scheduling of the threads is handled by the operating system, as well as thread priorities.

Within Java, the Java threads are represented by thread objects. Each thread also has a stack, used for storing runtime data. The thread stack has a specific size. If a thread tries to store more items on the stack than the stack size allows, the thread will throw a stack overflow error.

Default Stack Size for Java Threads

This section lists the default stack sizes. You can change the thread stack size with the `-Xss` command line option, for example:

```
java -Xss:512k MyApplication
```

The default stack sizes differ depending upon whether you are using IA32 and X64, as shown in Table 1:

Table 1 Default Stack Size

OS	Default Stack Size
Windows IA32	64 kB
Windows IA64	320 KB
Windows x64	128 kB
Linux IA32	128 kB
Linux IA64	1024 KB
Linux x64	256 kB
Solaris/SPARC	512 KB

Default Stack Size for JVM Internal Threads

A special "system" stack size is used for JVM internal threads; for example, the garbage collection and code generation threads. The default system stack size is 256 KB on all platforms.

Note: The `-Xss` command line option sets the stack size of both application threads and JVM internal threads.

Understanding Locks

When threads in a process share and update the same data, their activities must be synchronized to avoid errors. In Java, this is done with the `synchronized` keyword, or with `wait` and `notify`. Synchronization is achieved by the use of locks, each of which is associated with an object by the JVM. For a thread to work on an object, it must have control over the lock associated with it, it must "hold" the lock. Only one thread can hold a lock at a time. If a thread tries to take a lock that is already held by another thread, then it must wait until the lock is released. When this happens, there is so called "contention" for the lock.

There are four different kinds of locks:

- *Fat locks:* A fat lock is a lock with a history of contention (several threads trying to take the lock simultaneously), or a lock that has been waited on (for notification).
- *Thin locks:* A thin lock is a lock that does not have any contention.
- *Recursive locks:* A recursive lock is a lock that has been taken by a thread several times without having been released.
- *Lazy locks:* A lazy lock is a lock that is not released when a critical section is exited. Once a lazy lock is acquired by a thread, other threads that try to acquire the lock have to ensure that the lock is, or can be, released. Lazy locks are used by default in Oracle JRockit JVM 27.6. In older releases, lazy locks are only used if you have started the JVM with the `-XXlazyUnlocking` option.

A thin lock can be *`inflated`* to a fat lock and a fat lock can be *deflated* to a thin lock. The JRockit JVM uses a complex set of heuristics to determine when to inflate a thin lock to a fat lock and when to deflate a fat lock to a thin lock.

Spinning and Sleeping

Spinning occurs when a thread that wants a specific lock continuously checks that lock to see if it is still taken, instead of yielding CPU-time to another thread.

Alternatively, a thread that tries to take a lock that is already held waits for notification from the lock and goes into a sleeping state. The thread will then wait passively for the lock to be released.

Lock Chains

Several threads can be tied up in what is called *lock chains*. Although they appear somewhat complex, lock chains are fairly straightforward. They can be defined as follows:

- Threads A and B form a lock chain if thread A holds a lock that thread B is trying to take. If A is not trying to take a lock, then the lock chain is "open."
- If A->B is a lock chain, and B->C is a lock chain, then A->B->C is a more complete lock chain.
- If there is no additional thread waiting for a lock held by C, then A->B->C is a complete and open lock chain.

Lock Chain Types

The JRockit JVM analyzes the threads and forms complete lock chains. There are three possible kinds of lock chains: Open, Deadlocked and Blocked lock chains.

Open Chains

Open lock chains represent a straight dependency, thread A is waiting for B which is waiting for C, and so on. If you have long open lock chains, your application might be wasting time waiting for locks. You may then want to reconsider how locks are used for synchronization in your application.

Deadlock Chains

A deadlocked, or circular, lock chain consists of a chain of threads, in which the first thread in the chain is waiting for the last thread in the chain. In the simplest case, thread A is waiting for thread B, while thread B is waiting for thread A. Note that a deadlocked chain has no head. In thread dumps, the Oracle JRockit JVM selects an arbitrary thread to display as the first thread in the chain.

Deadlocks can never be resolved, and the application will be stuck waiting indefinitely.

Blocked Chains

A blocked lock chain is made up of a lock chain whose head thread is also part of another lock chain, which can be either open or deadlocked. For example, if thread A is waiting for thread B, thread B is waiting for thread A, and thread C is waiting for thread A, then thread A and B form a deadlocked lock chain, while thread C and thread A form a blocked lock chain.

CHAPTER 5

Migrating Applications to the Oracle JRockit JDK

This section describes how to migrate Java applications developed on another JDKs to the Oracle JRockit JDK to ensure optimal performance during runtime. This section contains information on the following subjects:

- About Application Migration
- Migration Support
- Migration Procedures
- Testing the Application
- Replicating Tools Supplied with the Sun JDK
- Command-line Option Compatibility Between the JRockit JVM and Sun
- Submitting Migration Tips

About Application Migration

Migrating an application to the JRockit JDK is a relatively simple process, requiring some minor environmental changes and following some simple coding guidelines. This section provides instructions and tips to successfully completing this simple process. It also describes some of the benefits and possible problems you might encounter during migration and it discusses some best J2SE coding practices for you to follow to ensure that your application runs successfully once it is running on the JRockit JDK.

Why Migrate?

The JRockit JDK is the default JDK shipped with Oracle WebLogic Server. Although there are other JDKs available on the market today that you can use to develop Java applications, Oracle recommends that you use JRockit JDK with your Oracle products.

Migration Restrictions

Migration is available for all platforms when Oracle WebLogic Server is supported with the JRockit JDK. For a list of supported platforms, please refer to:

http://edocs.bea.com/platform/suppconfigs/index.html

Migration Support

Should you experience any problems or find any bugs while attempting to migrate an application to the JRockit JDK, please send us an e-mail at **support@bea.com**. We would appreciate if you could provide as much information as possible about the problem, for example:

- Hardware
- Operating system and its version
- The program you are attempting to migrate
- Stack dumps (if any)
- A small code example that will reproduce the error
- Copies of any `*.dump` and `*.mdmp/core` process memory dump files. On Windows they are stored as `*.mdmp`, on Linux and Solaris as `core` or `core.*`

Migration Procedures

This section describes basic environmental and implementation changes necessary to migrate to JRockit JDK from a third-party JDK, such as the Sun Microsystems JDK. It includes information on the following subjects:

- Environment Changes
- Other Tips
- Tuning the JRockit JVM for Your Application

Environment Changes

Note: The changes described in this section apply primarily to Oracle WebLogic Server. If you are working with other Java applications, you will need to change the scripts and environments according to how that application is set up.

To migrate from a third-party JDK to the JRockit JDK, you need to make the following changes to the files.

- Set the `JAVA_HOME` environmental variable in `<WEBLOGIC_HOME>/common/commEnv.cmd` (or `.sh`) to the appropriate path.
- Set the `JAVA_VENDOR` environmental variable in `<WEBLOGIC_HOME>/common/commEnv.cmd` (or `.sh`) to `BEA`.
- If you are using a start-up script, remove any Sun-specific (or other JVM provider) options from the start command line (like `-XX:NewSize`). If possible, replace them with Oracle JRockit JVM-specific options; for example, `-Xns`. Other flags that might need to be changed include `MEM_ARGS` and `JAVA_VM`.

  (For more information on command-line options supported by the JRockit JVM, please refer to the Reference Manual.)
- Change `config.xml` to point the default compiler setting(s) to the `javac` compiler in the JRockit JDK.

Other Tips

For information on other coding practices that will ensure a successful migration of your application to the JRockit JDK, please refer to Developing Java Applications.

Tuning the JRockit JVM for Your Application

Once you've migrated your application to the JRockit JDK, you might want to tune the JVM for optimal performance. For example, you might want to specify a different start-up heap size or set custom garbage collection parameters. You can find detailed information on tuning the JRockit JVM in the chapters of the section Profiling and Performance Tuning.

The non-standard options, that is, options preceded with `-X`, are critical tools for tuning a JVM at startup. These options change the behavior of the JRockit JVM to better suit the needs of different Java applications.

While all JVMs use non-standard options, the option names might not be the same from JVM to JVM; for example, while the JRockit JVM will accept the non-standard option `-Xns` to set the nursery in generational concurrent and generational parallel garbage collectors, Sun's HotSpot JVM uses the non-standard option `-XX:NewSize` to set this value.

If you are migrating an application to the JRockit JDK, we recommend that you become familiar with the non-standard options available to you. For more information, please refer to the Reference Manual.

You should also be aware that, being non-standard, non-standard options are subject to change at any time.

Testing the Application

Always test your application on the JRockit JVM before putting it into production. If you develop your application on the Sun JVM (HotSpot), you *must* test your application on the JRockit JVM before you put it into production.

Why Test?

Some important reasons for testing are:

- Sometimes you might find bugs in your own program that don't occur on the Sun JVM; for example, synchronization problems.
- You might have used third party class libraries that do not follow the Java specifications and rely on Sun-specific classes or behavior.
- You might have used third-party class files that are not correct. The JRockit JVM has been known to enforce verification more rigorously than the Sun JVM.

How to Test

To test your application on the JRockit JVM:

1. Run your application against any test scripts or benchmarks that are appropriate for that application.
2. If any problems occur, handle them as you normally would for the specific application.

Replicating Tools Supplied with the Sun JDK

The following J2SE tools, normally available with the Sun JDK, are not shipped with the JRockit JDK:

- `jinfo`
- `jhat`
- `jmap`
- `jsadebugd`
- `jstack`

The JRockit JDK provides internal tools that are equivalent to or better than most of the Sun tools. Listing 5-1 lists the Sun tools and their JRockit JDK equivalents. Some of these tools require using the `jrcmd` feature. For more information, please refer to Running Diagnostic Commands

Table 5-1 JRockit JDK/Sun Tool Equivalents

Sun Tool	Shipped with JRockit JDK	JRockit JDK Equivalent
`jinfo`	No	`jrcmd <pid> print_properties` `jrcmd <pid> command_line`
`jhat`	No	JRockit Memory Leak Detector (see The JRockit Memory Leak Detector) and JRockit Runtime Analyzer (see The JRockit Runtime Analyzer).

`jmap`	No	JRockit Memory Leak Detector (see The JRockit Memory Leak Detector)
		`jrcmd <pid> print_object_summary`
		`jrcmd <pid> verbose_referents`
		`jrcmd <pid> heap_diagnostics`

`jsadebugd`	No	No equivalent tool
`jstack`	No	`jrcmd <pid> print_threads`
`jps`	Yes	`jrcmd`

Table 5-1 JRockit JDK/Sun Tool Equivalents

Sun Tool	Shipped with JRockit JDK	JRockit JDK Equivalent
`jstat`	Yes	No equivalent, `jstat` works with the JRockit JVM. The JRockit Runtime Analyzer (see The JRockit Runtime Analyzer) and the JRockit Management Console in Oracle JRockit Mission Control (see The JRockit Management Console) are however better tools for this purpose.
`jstatd`	Yes	No equivalent, `jstatd` works with the JRockit JVM.
`jconsole`	Yes	`jconsole` works with the JRockit JVM. The JRockit Management Console in JRockit Mission Control (see The JRockit Management Console) is however a better tool for monitoring the JRockit JVM.
`jrunscript`	Yes	No equivalent, `jrunscript` works with the JRockit JVM.

Command-line Option Compatibility Between the JRockit JVM and Sun

This section describes the compatibility between command-line options available when running the JRockit JVM and when running the Sun Hotspot JVM. These options correspond to each other by name only, by function only, or both by name and function.

Table 5-2 lists options that both Sun Hotspot JVM and the JRockit JVM have but have different functionality depending upon the JVM you are running.

Table 5-2 Same Name, Different Function

Option Name	Hotspot Function	JRockit JVM Function
`-Xms`	Sets the initial size of the heap	Sets the initial and minimum size of the heap. For complete description, please refer to `-Xms`.

Table 5-3 lists options that work the same or similarly on both Sun Hotspot and the JRockit JVM but have different names depending upon the JVM you are running.

Table 5-3 Different Name, Same or Similar Function

Hotspot Option Name	JRockit JVM Option Name	Function
`-XX:+AggressiveHeap`	`-XXaggressive:memory`	Configures the memory system for memory-intensive workloads and sets an expectation to enable large amounts of memory resources to ensure high throughput. The JRockit JVM will also use large pages, if available.
`-verbose:gc`	`-Xverbose:memory`	Prints out log information about the memory system
`-Xmn, -XXNewSize, -XXMaxNewSize`	`-Xns`	Sets the size of the young generation
`-XX:+UseConcMarkSweepGC`	`-Xgc:singlecon`	Sets the garbage collector to use a concurrent strategy
`-XX:+UseParallelGC`	`-Xgc:parallel`	Sets the garbage collector to use a parallel strategy

Table 5-4 lists options *only* available when using the Oracle JRockit JVM.

Table 5-4 Options Available for JRockit JVM Only

Option name	Function
`-XgcPrio`	Specifies what to prioritize: even pause times or maximum throughput
`-XpauseTarget`	Specifies a suitable pause time for the application

Submitting Migration Tips

The migration tips discussed in this section represent an evolving list. Often, a successful migration to the JRockit JDK depends as much upon the application being migrated as it does to the VMs being used. Oracle welcomes suggestions based upon your experiences with migrating applications to the Oracle JRockit JDK. Feel free to submit any migration ideas or comments to the Oracle JRockit forums at dev2dev.

CHAPTER 6

Setting Up and Running the Oracle JRockit JDK

Before using the Oracle JRockit JDK, you need to make sure that it is set up correctly. This section gives you hints on how to set up your environment for your specific platform. Once you have configured the environment correctly, you will find the diagnosing process easier. The configuration is done in the following steps:

- Installing the Oracle JRockit JDK
- Setting Up and Checking Your Linux Environment
- Setting Up and Checking Your Windows Environment
- Setting Up and Checking Your Sun Solaris Environment
- Setting the Path to the License File

Installing the Oracle JRockit JDK

The JRockit JDK is included in several Oracle products, for example Oracle JRockit Mission Control, Oracle JRockit Real Time and Oracle WebLogic.

Setting Up and Checking Your Linux Environment

The Linux operating systems exist in a large number of updates and patches. Oracle personnel is not able to test the JRockit JDK against every patch that is released. Instead we intend to test the most recent releases of some few distributions. As a general rule, you should keep your Linux environment up to date and make sure you have a release that is supported by Oracle when

running the JRockit JDK. Please see the Oracle JRockit JDK *Supported Configurations* document for a list of releases and distributions that the JRockit JDK has been successfully tested against.

The following path is the correct path for Linux installations:

```
export PATH=$HOME/jrockit-<jrockit_version>-jdk<sun_version>/bin:$PATH
```

Linux on IA32

When running the Oracle JRockit JVM on Linux IA32, it must be configured to use the glibc compiled for i686 architecture, otherwise you will see freezes and crashes with the JRockit JVM.

Check which glibc is installed by running:

```
rpm -q --queryformat '\n%{NAME} %{VERSION} %{RELEASE} %{ARCH}\n' glibc
```

Using LD_ASSUME_KERNEL

When using the JRockit JDK 1.4.2 on Linux, you should first make sure that the environment variable `LD_ASSUME_KERNEL` is *not* defined. If `LD_ASSUME_KERNEL` is defined, the JRockit JVM will use an older and slower threading implementation, which can deter—and will not improve—performance.

Running in a chroot(3) Environment

In some Linux versions the `/proc` filesystem isn't mounted when running in a `chroot(3)` environment. This may cause the JRockit JVM to become unstable or crash when running in the `chroot(3)` environment, as the JVM and some Linux libraries need to access information about the hardware platform from `/proc`.

To verify that `/proc` is mounted you can issue the shell command `getconf _NPROCESSORS_CONF` from the command line in your `chroot(3)` environment. This command should return the correct number of processors on your system, otherwise you will have to mount the `/proc` filesystem before running the JRockit JVM.

Setting Up and Checking Your Windows Environment

There are a number of environment variables that control the operation of the JRockit JVM on Windows. The following PATH environment variable needs to point to the directory of your Java installation:

```
set
PATH=%ProgramFiles%\Java\jrockit-<jrockit_version>-jdk<sun_version>\bin;%PATH%
```

Setting Up and Checking Your Sun Solaris Environment

The following path is the correct path for Solaris installations:

```
export PATH=$HOME/jrockit-<jrockit_version>-jdk<sun_version>/bin:$PATH
```

Oracle JRockit JDK is included in several products, for example Oracle JRockit Mission Control, Oracle JRockit Real Time and Oracle WebLogic. For more information, see the installation guides for your specific Oracle product.

Setting the Path to the License File

Note: Technical license checks have been removed as of this rel;ease. The following instructions apply only to versions of JRockit JDK prior to R27.6.

You can set the path to the license file by using the `-Djrockit.license.path` flag at startup. This option is useful when:

- You do not have write permissions to the JDK and therefore can't add the license path there.
- You want to start several servers with different IPs from the same Oracle JRockit JDK.

The option should point to a directory where the `license.bea` file resides and *not* directly to the license.bea file; for example:

- Correct:

  ```
  -Djrockit.license.path=C:/Program Files/Java/[JROCKIT_HOME]/jre
  ```

- Incorrect:

  ```
  -Djrockit.license.path=C:/Program
  Files/Java/[JROCKIT_HOME]/jre/license.bea
  ```

Part II Profiling and Performance Tuning

CHAPTER 7

About Profiling and Performance Tuning

Tuning the Oracle JRockit JVM to achieve optimal application performance is about the most critical aspect using this product. A poorly-tuned JVM can result in slow transactions, long latencies, system freezes, and even system crashes. This documen explores the many tuning techniques and options you can employ to see that your implementation of this JVM performs to maximum capabilities.

How to Tune: An Overview

Ideally, tuning should occur as part of the system startup, by employing various combinations of the start-up options described in the Oracle JRockit JVM *Command-Line Reference*. The Oracle JRockit JDK provides the necessary tools for monitoring your application during runtime. Properly tuned, according to the recommendations in this section, the JVM should run smoothly and provide timely results. Should runtime monitoring indicate problems along the way, you can use the recommendations in this section to guide you toward a better-tuned JVM.

What this Section Contains

This section includes information on the following subjects:

- While tuning the JRockit JVM you will often find a certain trade-off between short garbage collection pause times, high application throughput and low memory footprint. Understanding the Tuning Trade-offs describes these trade-offs and the reasons behind them.

- Each Java application has its own behavior and its own requirements. The JRockit JVM can accommodate most of them automatically, but to get the optimal performance you should tune at least some basic parameters. First Steps for Tuning the Oracle JRockit JVM gives an overview of the first steps of tuning the JRockit JVM and some best practices for tuning the JVM for a few different Oracle applications.
- A correctly tuned memory management system minimizes the overhead inflicted by garbage collection and makes object allocation fast. Tuning the Memory Management System covers the most important options available for tuning the memory management system in the JRockit JVM.
- The interaction between Java threads affects the performance of your application. Tuning Locks contains information about the JRockit JVM options for tuning how locks and contention for locks are handled.
- Do you want your application to run smoothly with minimal pauses caused by the garbage collection? If the answer is “yes”, then you want to tune for short pause times. Using the tuning techniques described in Tuning For Low Latencies to ensure that pause times are kept to a minimum and transactions execute quickly.
- Do you want to minimize the total amount of CPU time spent in garbage collection and spend more time in the application layer? If the answer is “yes”, then you want to tune for high application performance, or application *throughput*. Tuning For Better Application Throughput describes how to tune your Oracle JRockit JVM to ensure that the Java application runs as fast as possible with minimal garbage collector overhead.

CHAPTER 8

Understanding the Tuning Trade-offs

While tuning the Oracle JRockit JVM you will often find a certain trade-off between short garbage collection pause times, high application throughput and low memory footprint. This section describes these trade-offs and the reasons behind them. The following topics are covered:

- Pause Times vs. Throughput
- Performance vs. Memory Footprint

Pause Times vs. Throughput

The JRockit JVM offers a choice between short garbage collection pauses and maximum application throughput. Intuitively it looks like short garbage collection pauses would also maximize the application throughput, which may make you wonder why you have to choose between the two. This section describes the reasons behind this trade-off.

Concurrent vs. “Stop-the-World”

The trade-off between garbage collection pauses and application throughput is partly caused by the mostly concurrent garbage collection strategy that enables short garbage collection pauses. No matter how efficient a garbage collection algorithm that stops the Java threads during the entire garbage collection is, a garbage collection algorithm that allows the Java threads to continue running during parts of the garbage collection will always give you shorter individual garbage collection pauses. Unfortunately a concurrent algorithm requires more bookkeeping and some extra work, since new objects are created and references between objects change during the garbage collection. All these changes must be kept track of, which alone causes some slight

overhead, and at some point the garbage collector must handle all the changes, which causes some extra work. Simply put, the more the garbage collector can do while the Java threads are paused, the less it has to work in total.

Compaction Pauses vs. Throughput

The mark and sweep garbage collection model can cause fragmentation on the heap when small chunks of memory are freed between blocks of live objects. Compaction of the heap reduces this fragmentation. Fragmentation has a negative impact on the overall throughput as it makes object allocation more difficult and garbage collections more frequent. The JRockit JVM does partial compaction of the heap in each garbage collection, and the compaction is done while all Java threads are paused. Moving objects takes time, and compaction takes more time the more objects it moves. The trade-off is simple - reducing the amount of compaction shortens the compaction pause times but lowers the overall throughput by increasing the fragmentation.

Performance vs. Memory Footprint

A small memory footprint is desirable for applications that run on machines with limited memory resources. Unfortunately there is a certain trade-off between a small memory footprint and both application throughput and garbage collection pauses. This section describes some of the reasons for this trade-off.

Heap Size vs. Throughput

A large heap reduces the garbage collection frequency and the negative impact of fragmentation, thus improving the throughput of the application. On the other hand a large heap increases the memory footprint of the Java process.

Book Keeping vs. Pause Times

When you use a garbage collection mode that optimizes for short pauses, the Oracle JRockit JVM will have to use more advanced book keeping to keep track of changes in the heap, references to objects that are compacted etc. All this increases the memory footprint. You cannot tune the memory usage for book keeping other than by selecting a different garbage collection mode or strategy.

CHAPTER 9

First Steps for Tuning the Oracle JRockit JVM

Each Java application has its own behavior and its own requirements. The Oracle JRockit JVM can accommodate many of them automatically, but to get the optimal performance you should tune at least some basic parameters. This section gives an overview of the first steps of tuning the JRockit JVM and some best practices for tuning the JVM for a few different Oracle applications, covering the following topics:

- Step 1: Basic Tuning
- Step 2: Performance Tuning
- Step 3: Advanced Tuning
- Best Practices

For in-depth information on tuning the JRockit JVM, please see Tuning the Memory Management System and Tuning Locks.

Step 1: Basic Tuning

The first steps of tuning are:

- Tuning the Heap Size
- Tuning the Garbage Collection
- Tuning the Nursery Size
- Tuning the Pause Target

Tuning the Heap Size

The heap is the area where Java objects reside. A large heap decreases the garbage collection frequency but may take slightly longer to garbage collect. Typically a heap should be at least twice the size of the live objects in the heap, meaning that at least half of the heap should be freed at each garbage collection. For server applications you can usually set the heap as large as the available memory in your system will allow, as long as this doesn't cause paging.

Set the heap size using the following command line options:

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

For example a server application running on a machine with 2 GB RAM memory could be started with the following settings:

```
java -Xms:800m -Xmx:1000m MyServerApp
```

This starts the JVM with a heap of 800 MB and allows the heap to grow up to 1000MB.

For in-depth information on setting the heap size, see Setting the Heap Size.

Tuning the Garbage Collection

Garbage collection is the process of reclaiming space from objects that are no longer in use, so that this space can be used for allocation of new objects. Garbage collection uses system resources in one way or another. By tuning the garbage collection you can decide how and when the resources are used. The JRockit JVM offers three garbage collection modes and a number of static garbage collection strategies. These allow you to tune the garbage collection to suit your application's needs.

Select the garbage collection mode by using one of the following options:

- `-XgcPrio:throughput`, which defines that the garbage collection should be optimized for application throughput. This is the default garbage collection mode.
- `-XgcPrio:pausetime`, which defines that the garbage collection should be optimized for short garbage collection pauses.
- `-XgcPrio:deterministic`, which defines that the garbage collection should be optimized for very short and deterministic garbage collection pauses. This option is only available as part of Oracle JRockit Real Time.

For example a transaction based application which requires reasonably low latencies could be started with the following settings:

```
java -XgcPrio:pauseTime MyTransactionApp
```

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

For in-depth information on selecting a garbage collection mode or a static garbage collection strategy, see Selecting and Tuning a Garbage Collector.

Tuning the Nursery Size

Some of the garbage collection modes and strategies in the JRockit JVM use a nursery. The nursery is an area of the heap where new objects are allocated. When the nursery becomes full it is garbage collected separately in a young collection. The nursery size decides the frequency and duration of young collections. A larger nursery decreases the frequency but slightly increases the duration of each young collection.

In the JRockit JVM R27.3.0 and later the nursery size is adjusted automatically to optimize for application throughput if you use `-XgcPrio:throughput` (default) or `-Xgc:genpar`. For other garbage collection modes and static strategies or older versions of the JVM you may want to tune the nursery size manually. Typically the nursery size should be as large as possible while maintaining reasonably short young collection pauses. Depending on the application, a reasonable nursery size can be anything from a few megabytes up to about half of the heap size.

Set the nursery size by using the following command line option:

- `-Xns:<size>`

For example a transaction based application running on a machine with 2GB RAM memory could be started with the following settings:

```
java -Xms:800m -Xmx:1000m -XgcPrio:pausetime -Xns:100m MyTransactionApp
```

This starts up the JVM with a heap of 800 MB, allowing it to grow up to 1000 MB. The garbage collection is set to optimize for pause times and the nursery size is set to 100 MB. Note that the dynamic garbage collection mode may choose to run without a nursery, but whenever a nursery is used it will be 100 MB.

For in-depth information on how to tune the nursery size, see Setting the Nursery and Keep Area Size.

Tuning the Pause Target

-XgcPrio:pausetime and -XgcPrio:deterministic use a pause target for optimizing the pause times while keeping the application throughput as high as possible. A higher pause target usually allows for a higher application throughput, thus you should set the pause target as high as your application can tolerate.

Set the pause target by using the following command line option:

- `-XpauseTarget:<time>`

For example a transaction based application with transactions that normally take 100 ms and time out after 400 ms could be started with the following settings:

```
java -XgcPrio:pausetime -XpauseTarget:250 MyTransactionApp
```

This starts up the JVM with garbage collection optimized for short pauses with a pause target of 250 ms. This leaves a 50 ms margin before time-out for 100 ms transactions that are interrupted by a 250 ms garbage collection pause.

For in-depth information on tuning the pause target, see Setting a Pause Target for Pausetime Mode.

Step 2: Performance Tuning

To be able to tune your JVM for better application throughput you must first have a way of assessing the throughput. A common way of measuring the application throughput is to time the execution of a pre-defined set of test cases. Optimally the test cases should simulate several different use cases and be as close to real scenarios as possible. Also, one test run should take at least a few minutes, so that the JVM has time to warm up.

This section describes a few optional performance features that improve the performance for many applications. Once you have a way of assessing the throughput of your application you can try out the following features:

- Lazy Unlocking
- Call Profiling
- Large Pages

Lazy Unlocking

The JRockit JVM R27.3 and later offers a feature called *lazy unlocking*. This feature makes synchronized Java code run faster when the contention on the locks is low.

Try this feature on your application by adding the following option to the command line:

- `-XXlazyUnlocking`

For more information on this option, see the documentation for `-XXlazyUnlocking`.

Call Profiling

Call profiling enables the use of more advanced profiling for code optimizations and can increase the performance for many applications. This option is supported in the JRockit JVM R27.3.0 and later versions.

Try this feature on your application by adding the following option to the command line:

- `-XXcallProfiling`

For more information on this option, see the documentation for `-XXcallProfiling`.

Large Pages

The JRockit JVM can use large pages for the Java heap and other memory areas in the JVM. To use large pages, you must first configure your operating system for large pages. Then you can add the following option to the Java command line:

- `-XlargePages`

For complete instructions on how to use this option and configure your operating system for large pages, see the documentation for `-XlargePages`.

Step 3: Advanced Tuning

Some applications may benefit from further tuning. It is important that you verify the results of the tuning by monitoring and benchmarking your application. Advanced tuning of the JRockit JVM can give you improved performance and predictable behavior if done correctly, while incorrect tuning may lead to uneven performance, low performance or performance degradation over time.

This section covers the following topics:

Tuning Compaction

Compaction of objects is the process of moving objects closer to each other in the heap, thus reducing the fragmentation and making object allocation easier for the JVM. The JRockit JVM compacts a part of the heap at each garbage collection (or old collection, if the garbage collector is generational).

Compaction may in some cases lead to long garbage collection pauses. To assess the impact of compaction on garbage collection pauses you can either monitor the `-Xverbose:gcpause` outputs or create a JRA recording and look at the garbage collection pauses in the Java Runtime Analyzer (see Using Oracle JRockit Mission Control Tools for more information). Look for old collection pause times and pause parts called “compaction” and “reference updates”. The compaction pause times depend on the compaction ratio and the compact set limit.

Compaction Ratio

The compaction ratio determines how many percent of the heap will be compacted during each garbage collection (old collection). The compaction ratio is set using the following option:

- `-XXcompactRatio:<percentage>`

You can tune the compaction ratio if the garbage collection pauses are too long because of compaction. As a start, you can try lowering the compaction ratio to 1 and see if the problem persists. If it doesn’t, you should try gradually increasing the compaction ratio as long as the compaction times stay short. A good value for the compact ratio is usually between 1 and 20, sometimes even higher. If the problem persists even though you set the compaction ratio to 1, you can try changing the compact set limit.

Setting the compaction ratio too low may increase the fragmentation and the amount of “dark matter”, which is free space that is too small to be used for object allocation. You can see the amount of dark matter in JRA recordings.

Compact Set Limit

The compact set limit prevents sets a limit for how many references there can be to objects within the compaction area. If the number of references exceeds this limit, the compaction is canceled. The compact set limit is set using the following option:

- `-XXcompactSetLimit:<references>`

You can tune the compact set limit if the garbage collection pauses are too long due to compaction. As a start, you can try setting the compact set limit as low as 10.000. If the problem is solved you should try gradually increasing the compact set limit as long as the compaction times stay low. A normal value for the compact set limit is usually between 100.000 and several million, while lower values are used when the pause time limits are very low.

Setting the compact set limit too low may stop compaction from being done altogether, which you can see in the verbose logs or in a JRA recording, where all compactions are noted as "aborted". Running without any compaction at all may lead to increasing fragmentation, which will in the end force the JVM to perform a full compaction of the whole heap at once, which may take several seconds. Thus we recommend that you do not decrease the compact set limit unless you really have to.

Note: `-XXcompactSetLimit` has no effect when `-XgcPrio:deterministic` or `-XgcPrio:pausetime` is used. For these garbage collection modes you should not tune the compaction manually, but instead use the `-XpauseTarget` option to tune the garbage collection pauses.

For in-depth information on how to tune compaction, see Tuning the Compaction of Memory.

Tuning the TLA size

The thread local area (TLA) is a chunk of free space reserved on the heap or the nursery and given to a thread for its exclusive use. A thread can allocate small objects in its own TLA without synchronizing with other threads. When the TLA gets full the thread simply requests a new TLA. The objects allocated in a TLA are accessible to all Java threads and are not considered "thread local" in any way after they have been allocated.

Increasing the TLA size is beneficial for multi threaded applications where each thread allocates a lot of objects. Increasing the TLA size is also beneficial when the average size of the allocated objects is large, as this allows larger objects to be allocated in the TLAs. Increasing the TLA size too much may however cause more fragmentation and more frequent garbage collections. To assess the sizes of the objects allocated by your application you can do a JRA recording and view object allocation statistics in the Java Runtime Analyzer. See Using Oracle JRockit Mission Control Tools for more information on JRA.

The TLA size is set using the following option:

- `-XXtlaSize:min=<size>,preferred=<size>`

The “min” value is the minimum TLA size, while the “preferred” value is a preferred size. This means that TLAs will be of the “preferred” size whenever possible, but may be as small as the “min” size. Typically the preferred TLA size can be up to twice the size of the largest commonly used object size in the application. Adjusting the min size may have an effect on garbage collection performance, but is seldom necessary. A normal value for the min size is 2 KB.

For in-depth information about tuning the TLA size, see Optimizing Memory Allocation Performance.

Further Information

Further information on tuning the JRockit JVM can be found in Tuning the Memory Management System and Tuning Locks.

Best Practices

This section lists some best practices for tuning the JRockit JVM for a number of specific applications and application types.

Oracle WebLogic Server

Oracle WebLogic Server is an application server, and as such it requires high application throughput. An application server is often set up in a controlled environment on a dedicated machine. Try the following when tuning the JRockit JVM for Oracle WebLogic Server:

- Use a large heap, several gigabytes if the system allows for it.
- Set the initial/minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the default garbage collection mode, `-XgcPrio:throughput`

Oracle WebLogic SIP Server

Oracle WebLogic SIP Server is an application server specialized for the communications industry. Typically it requires fairly low latencies and is run in a controlled environment on a dedicated machine. Try the following when tuning the JRockit JVM for Oracle WebLogic SIP Server:

- Use a large heap, at least a couple of gigabytes if the system allows for it.

- Set the initial/minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the garbage collection mode optimized for pause times, `-XgcPrio:pausetime`, or the static generational concurrent garbage collector, `-Xgc:gencon`.
- Use a fairly small nursery, in the range of 50-100 MB.
- Decrease the compaction ratio or compact set limit to lower and even out the compaction pause times, see Tuning Compaction for more information.

Oracle WebLogic Event Server

Oracle WebLogic Event Server is an application server for applications based on an event-driven architecture. Typically it requires very low latencies and is run in a controlled environment on a dedicated machine. Try the following settings when tuning the JRockit JVM for Oracle WebLogic Event Server:

- Use a heap size of 1 GB to fully utilize the deterministic garbage collection mode.
- Set the initial/minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the garbage collection mode optimized for low and deterministic latencies, `-XgcPrio:deterministic`. The deterministic garbage collection mode is only available as a part of JRockit Real Time.

Oracle Workshop

Oracle Workshop consists of several Eclipse plug-ins. Eclipse requires fast response times and is typically run on a workstation together with many other applications. Try the following settings when tuning the JRockit JVM for Eclipse together with Oracle Workshop.

- Use a maximum heap size that is lower than the amount of RAM in the system and leaves space for the operating system and a varying number of other applications running simultaneously.
- Set the initial/minimum heap size (`-Xms`) lower than the maximum heap size (`-Xmx`) to allow the JVM to resize the heap when necessary.
- Use the garbage collection mode optimized for short pauses, `-XgcPrio:pausetime`, or the default garbage collection mode, `-XgcPrio:throughput`.

"Utility" Applications

Java utility applications that run for a short time and have a simple and specific purpose, for example javac, require a fast startup and often don't need a lot of memory. To tune the JRockit JVM for this kind of applications, try the following recommendations:

- Use a small heap, anything from 16 MB and up depending on the application's needs.
- Set the initial/minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the default garbage collection mode, `-XgcPrio:throughput`

"Batch" Runs

Data processing applications that process large batches of data, for example applications for XML processing or data mining, require maximum application throughput but are seldom sensitive to long latencies. To tune the Oracle JRockit JVM for this kind of applications, try the following recommendations:

- Set the heap size as large as your system can tolerate, almost as much as the amount of physical memory in the system while leaving some memory for the operating system and other applications that may be running at the same time.
- Set the initial/minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the default garbage collection mode, `-XgcPrio:throughput`.
- Increase the thread local area size. See Tuning the TLA size for more information.

CHAPTER 10

Tuning the Memory Management System

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

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

Setting the Heap and Nursery Size

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

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

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

Setting the Heap Size

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

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

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

There are two parameters for setting the heap size:

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

For example:

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

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

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

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

Setting the Heap Size on 64-bit Systems

On 64-bit systems a memory address is 64 bits long, which makes it possible to address much more memory than with a 32-bit address. On the other hand each reference requires twice as much memory. To reduce the memory usage on 64-bit systems, The JRockit JVM can use *compressed references*. Compressed references reduce the references to 32 bits, and can be used as long as the entire heap can be addressed with 32 bits. Compressed references are enabled by default

whenever applicable. Thus, on a 64.bit system, you will usually benefit from setting the maximum heap size below 4 GB as long as the amount of live data is less than 3-4 GB.

Setting the Nursery and Keep Area Size

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

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

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

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

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

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

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

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

Keep Area

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

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

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

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

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

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

Selecting and Tuning a Garbage Collector

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

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

Selecting a Dynamic Garbage Collection Mode

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

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

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

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

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

Throughput Mode

Command line option: `-XgcPrio:throughput`

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

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

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

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

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

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

Pausetime Mode

Command line option: `-XgcPrio:pausetime`

The dynamic garbage collection mode optimizing over pause times aims to keep the garbage collection pauses below a given pause target while maintaining as high throughput as possible. The JRockit JVM achieves this by choosing between a mostly concurrent garbage collection strategy that allows the Java application to continue running during large portions of the garbage collection duration, and a parallel garbage collection strategy that stops the Java application

during the entire garbage collection duration. The mostly concurrent garbage collector introduces some extra overhead in keeping track of changes during the concurrent phases, and will also cause more frequent garbage collections. This will lower the overall throughput somewhat, but keeps down the individual garbage collection pauses.

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

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

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

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

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

Setting a Pause Target for Pausetime Mode

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

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

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

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

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

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

Deterministic Mode

Command line option: `-XgcPrio:deterministic`

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

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

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

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

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

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

Special Note for WLRT Users

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

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

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

Setting a Pause Target for Deterministic Mode

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

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

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

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

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

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

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

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

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

Selecting a Static Garbage Collection Strategy

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

There are four major static garbage collection strategies available.

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

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

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

Garbage Collector Strategy Selection Workflow

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

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

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

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

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

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

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

Changing Garbage Collection Strategy During Runtime

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

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

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

Tuning the Concurrent Garbage Collection Trigger

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

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

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

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

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

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

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

Tuning the Compaction of Memory

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

Fragmentation vs. Garbage Collection Pauses

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

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

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

Adjusting Compaction

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

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

Setting the Compaction Ratio

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

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

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

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

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

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

Setting the Compact Set Limit

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

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

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

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

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

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

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

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

Turning Off Compaction

Command line option: `-XXnoCompaction`

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

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

```
java -XXnoCompaction MyApplication
```

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

Using Full Compaction

Command line option: `-XXfullCompaction`

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

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

```
java -XXfullCompaction MyApplication
```

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

Optimizing Memory Allocation Performance

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

Setting the Thread Local Area Size

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

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

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

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

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

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

Decreasing the minimum TLA size lessens the impact of fragmentation.

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

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

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

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

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

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

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

CHAPTER 11

Tuning Locks

The interaction between Java threads affects the performance of your application.There are two ways of tuning the interaction of threads.

3. By modifying the structure of your program code, for example to minimize the amount of contention between threads.
4. By using options in the Oracle JRockit JVM that affect how contention is handled when your application is running.

The Oracle JRockit JVM Diagnostics Guide does not provide any documentation on how to optimize thread management when coding Java, but this section contains information about JRockit JVM options for tuning how locks and contention for locks are handled. This section covers the following topics:

- Lock Profiling
- Disabling Spinning Against Fat Locks
- Adaptive Spinning Against Fat Locks
- Lock Deflation
- Lazy Unlocking

For more information on how the JRockit JVM handles threads and locks, see Understanding Threads and Locks.

Lock Profiling

You can enable the JRockit Runtime Analyzer to collect and analyze information about the locks and the contention that has occurred while the runtime analyzer was recording. To do this, add the following option when you start your application:

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

When lock profiling has been enabled, you can view information about Java locks on the Lock Profiling tab in the JRockit Mission Control Client.

Note: Lock profiling creates a lot (in the order of 20%) of overhead processing when your Java application runs.

There are two Ctrl-Break handlers tied to the lock profile counters. To work, both require lock profiling to be enabled with the `-Djrockit.lockprofiling` option. These are used with jrcmd.

The handler `lockprofile_print` prints the current values of the lock profile counters. The handler `lockprofile_reset` resets the current values of the lock profile counters.

For more information about Ctrl-Break handlers and using jrcmd, see Running Diagnostic Commands.

Disabling Spinning Against Fat Locks

Spinning against a fat lock is generally beneficial. However, in some instances, it can be expensive and costly in terms of performance, for example when you have locks that create long waiting periods and high contention. You can turn off spinning against a fat lock and eliminate a potential performance degradation with the following option:

```
-XXdisableFatSpin
```

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

Adaptive Spinning Against Fat Locks

You can let the JVM decide whether threads should spin against a fat lock or not (and directly go into sleeping state when failing to take it). To enable adaptive lock spinning, set the option

```
-Djrockit.useAdaptiveFatSpin=true
```

By default, adaptive spinning against fat locks is disabled. Note that whether threads failing to take a particular fat lock will go spinning or sleeping can change during runtime.

You can specify the criteria that needs to be fulfilled for threads to start spinning against a fat lock. The following options let you tune adaptive spinning.

```
-Djrockit.adaptiveFatSpinTimeStampDiff=2000000
```

This sets the maximum difference in CPU-specific ticks where spinning is beneficial.

```
-Djrockit.adaptiveFatSpinMaxSpin=1000
```

Number of spins that must fail before threads switch from spinning to sleeping.

```
-Djrockit.adaptiveFatSpinMaxSleep=1000
```

Number of sleeps that must get the lock early before threads go back to spinning.

```
-Djrockit.fatlockspins=100
```

Number of loops before JRockit JVM tries to read from the lock again in the innermost lock spin code.

Lock Deflation

If the amount of contention on a lock that has turned fat has been small, then the lock will convert back to a thin lock. This process is called lock deflation. By default, lock deflation is enabled. If you do not want fat locks to deflate, then run you application with the following option:

```
-XXdisableFatLockDeflation
```

With lock deflation disabled, a fat lock stays a fat lock even after there is no threads contending or waiting to take the lock.

You can also tune when lock deflation will be triggered. Specify, with the following option, the number of uncontended fat lock unlocks that should occur before deflation:

```
-XXfatLockDeflationThreshold=<NumberOfUnlocks>
```

Lazy Unlocking

So called “lazy” unlocking is intended for applications with many non-shared locks. Be aware that it can introduce performance penalties with applications that have many short-lived but shared locks.

When lazy unlocking is enabled, 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.

Lazy unlocking is enabled by default in the Java 6 version of the Oracle JRockit JVM R27.6 on all platforms except IA64 and for all garbage collection strategies except the deterministic garbage collector. In older releases you can enable lazy unlocking with the command line option `-XXlazyUnlocking`.

CHAPTER 12

Tuning For Low Latencies

Long latencies can make some applications behave poorly even though the overall throughput is good. For example, a transaction based system may seem to perform well as to the number of transactions executing during a specified amount of time, but still show some uneven behavior with transactions timing out now and then even on low loads. Latencies in the application or the environment in which the application is run may cause this uneven or poor performance. Latencies can be due to anything from contention in the Java code to slow network connections to a database server. Latencies may also be caused by the JVM, for example during garbage collection, depending on how the JVM is tuned. This section describes how to tune the Oracle JRockit JVM for low latencies, covering the following subjects:

Measuring Latencies

Most application developers have a way of measuring their application's performance. You can for example run a set of simulated use cases and measure the time it took to execute them, the number of a specific kind of transactions executed per minute, the average transaction time or

how many percent of the transaction times are above or below a specific threshold. When you tune for low latencies you will be most interested in measuring the amount of transaction times that are above a certain threshold. For best tuning results you should have a varied set of benchmarks that are as realistic as possible and run for a longer period of time. Twenty minutes is often a minimum, and sometimes the full effect of the tuning can be seen only after several hours.

When you have identified a situation where the long latencies occur, you can start monitoring the JRockit JVM using some of the following methods:

- Create a runtime analysis report by using the JRockit Runtime Analyzer (JRA) supplied with the product. If you are running the JRockit JVM R27.1 or later and Oracle JRockit Mission Control 2.0 or later version of JRockit Runtime Analyzer, the individual pause times for each garbage collection pause (there might be several pauses during one garbage collection) are reported. The JRA report will also show page faults occurring during garbage collection. For information on creating and analyzing a JRA report, please refer to the online help in the JRockit Mission Control Client or the Oracle JRockit Mission Control documentation.
- You can create a latency recording to monitor the occurrences of latencies in your application. For more information on creating a Latency Recording, please see the online help in the Oracle JRockit Mission Control Client or the Oracle JRockit Mission Control documentation.
- You can see garbage collection pause times in the JRockit JVM by starting the JVM with `-Xverbose:gcpause`.
- If you are using an older versions of the JRockit JVM (that is, prior to version R27.1) and an older version of JRA, use the command-line option `-Xverbose:memdbg,gcpause` to print out the garbage collection pause times. The parameter `memdbg` will also display more detailed printouts about page faults that occur during garbage collection.

Now you have the tools to see the results of your tuning.

Tune the Garbage Collection

The first step for tuning the JRockit JVM for low latencies is to select a garbage collection mode that gives you short garbage collection pauses. The best bet is one of the following two dynamic garbage collection modes or one static garbage collection strategy, described further in this section:

- Dynamic Garbage Collection Mode Optimized for Deterministic Pauses.

This is the garbage collection mode designed for very short and deterministic garbage collection pauses. It is available as a part of Oracle JRockit Real Time.

- Dynamic Garbage Collection Mode Optimized for Short Pauses.

  This is a garbage collection mode designed for short garbage collection pauses.

- Static Generational Concurrent Garbage Collection

  This static garbage collection mode provides fairly short garbage collection pauses but does not optimize for a specific pause target. Additional tuning of the nursery size and compaction may be necessary when this garbage collector is chosen.

For more information about different garbage collector options, see Selecting and Tuning a Garbage Collector.

Dynamic Garbage Collection Mode Optimized for Deterministic Pauses

Applications that require minimal latency, such as those used in the telecom and finance industries, cannot abide by the unpredictable pause times caused common garbage collection strategies. To avoid these overly-long pauses, the JRockit JVM provides "deterministic" garbage collection, a dynamic garbage collection mode that keeps the garbage collection pauses short and deterministic.

Set the deterministic garbage collector at the commandline as follows:

```
java -XgcPrio:deterministic -Xms:1g -Xmx:1g myApplication
```

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

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

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

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

```
java -XgcPrio:deterministic -Xms:1g -Xmx:1g -XpauseTarget:40ms
MyApplication
```

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

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

The deterministic garbage collector optimizes the compaction for the given pause target and does not use a nursery. Further tuning of compaction and nursery size should thus be unnecessary when the deterministic garbage collector is used.

Dynamic Garbage Collection Mode Optimized for Short Pauses

The dynamic garbage collection mode optimized for short pauses is useful for applications that don't require quite as short and deterministic pauses as the deterministic garbage collector guarantees. This garbage collection mode selects a garbage collection strategy to keep the garbage collection pauses below a given pause target (500 ms by default). Compaction will also be adjusted automatically to keep down the pause times caused by compaction.

Set the pausetime priority as follows:

`java` **`-XgcPrio:pausetime`** *`myApplication`*

If you use the pausetime priority but find that the default (500 ms) is too long, you can specify a target pause time by using the `-XpauseTarget` option, for example:

`java -XgcPrio:pausetime` **`-XpauseTarget=200ms`** *`myApplication`*

Be aware that there is a certain trade off between short pauses and application throughput. Shorter garbage collection pauses require more overhead in bookkeeping and may cause more fragmentation, which lowers the performance. If your application can tolerate pause times longer than 500 ms you can increase the pause target to increase the application's performance.

The target value is used as a pause time goal and by 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 200 ms and 5 seconds. If you don't specify a pause target, the default remains 500 ms.

The garbage collection mode for short pauses optimizes the compaction for the given pause target, so further tuning of the compaction should not be necessary. The nursery size is adjusted automatically, but for an even performance you may need to tune the nursery size manually. In

R27.3 and later releases, the nursery size is static for this garbage collection mode and will have to be tuned manually.

Static Generational Concurrent Garbage Collection

If you want to use a static garbage collector and still experience minimal pause times, use a concurrent garbage collector. Generally, using a generational garbage collector is preferable to using a single-spaced garbage collector since a generational garbage collector gives you better application throughput.

To use a generational concurrent garbage collector, enter the following at the command line:

```
java -Xgc:gencon myApplication
```

To use a single-spaced concurrent garbage collector, enter the following at the command line:

```
java -Xgc:singlecon myApplication
```

When you use a static garbage collector you may have to tune the nursery size and the compaction manually.

Tune the Heap Size

You can resize the heap by using the `-Xms` (initial and minimum heap size) and `-Xmx` (maximum heap size) command line options when you launch the JRockit JVM. Usually you can set the initial and the maximum heap size to the same value. Increasing the heap size reduces the frequency of garbage collections. A larger heap may also take slightly longer to garbage collect, but this effect is usually not considerable until the heap reaches sizes of several gigabytes.

The best approach to tuning the heap size is simply to benchmark the application with many different heap sizes. Monitor the garbage collection pauses as described in Measuring Latencies while you do this to determine the largest possible heap size for your application.

The only exception is the deterministic garbage collector. The deterministic garbage collector is verified using a heap of about 1 GB, and will work best with heaps of about this size.

To set the heap size, use the `-Xms` and the `-Xmx` options, for example

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

For more information, see Optimizing Memory Allocation Performance.

Manually Tune the Nursery Size

If you are running `-XgcPrio:pausetime` or `-Xgc:gencon` you might want to tune the nursery size manually.

The size of the nursery changes dynamically in runtime when you use `-XgcPrio:pausetime`, but setting it manually gives a more even behavior (note that when you use the dynamic garbage collector, the nursery might also be turned off completely when single-spaced garbage collection is used).

Note: In the Oracle JRockit JVM R27.3 and later versions the nursery size is static when running `-XgcPrio:pausetime`. Tuning the nursery size manually is often beneficial for both the pause times and the application throughput.

The default nursery size for `-Xgc:gencon` is static, and may thus not be optimal for all applications. You might benefit from manually setting a custom nursery size. The nursery should be as large as possible, but the nursery size must be decreased if the pause time created by a young collection (nursery garbage collection) is too long. Tune the nursery size by benchmarking your application with several different nursery sizes while monitoring the garbage collection pauses as described in Measuring Latencies.

To set the size of the nursery, use the `-Xns` option; for example:

```
java -XgcPrio:pausetime -Xns:64m myApp
```

Manually Tune Compaction

If you are using a static garbage collector, tuning compaction manually might help improve latencies. Compaction is performed during a garbage collection pause, and thus the compaction time affects the garage collection pause times. By default, the static garbage collectors use a compaction scheme that aims at keeping the compaction times fairly even, but does not put an upper bound on the compaction time.

You can limit the compaction manually by setting a static compaction area size (`-XXcompactRatio`) or by limiting the number of references that can be updated due to compaction (`-XXcopactSetLimit`). Neither action will not guarantee an upper bound on the compaction time, but will reduce the risk for long compaction times.

Be aware that if you set the compaction ratio to low, the heap slowly becomes more and more fragmented until it is impossible to find free space that is big enough for object allocation. The heap becomes full of *dark matter* (basically severe fragmentation). When this happens, a full

compaction (a compaction of the complete heap) will be done, which can result in a pause times of up to half a minute. Dark matter is reported for the heap in a JRA recording.

For complete information on limiting compaction, please refer to Adjusting Compaction.

Tune When to Trigger a Garbage Collection

The `-XXgcTrigger` 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 garbage collection frees some memory, 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 might take too long. Instead, you can use `-XXgcTrigger` option 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; for example:

```
java -XXgcTrigger myApp
```

The current value of the garbage collection trigger appears in the `-Xverbose:memdbg` outputs whenever the trigger changes.

CHAPTER 13

Tuning For Better Application Throughput

Every application has a unique behavior and has its own unique requirements on the JVM for gaining maximum application throughput. The “out of the box” behavior of the Oracle JRockit JVM gives good performance for most applications. You can however often tune the JVM further to gain some extra application throughput, which means that the application will run faster.

This chapter describes how to tune the JRockit JVM for improved application throughput. It includes information on the following subjects:

Measuring Your Application’s Throughput

In this document “application throughput” denotes the speed at which a Java application runs. If your application is a transaction based system, high throughput means that more transactions are executed during a given amount of time. You can also measure the throughput by measuring how long it takes to perform a specific task or calculation.

To measure the throughput of your application you need a benchmark. The benchmark should simulate several realistic use cases of the application and run long enough to allow the JVM to warm up and perform several garbage collections. You also need a way to measure the results, either by timing the entire run of a specific set of actions or by measuring the number of transactions that can be performed during a specific amount of time. For an optimal throughput assession, the benchmark should run on high load and not depend on any external input like database connections.

When you have a benchmark set up, you can monitor the behavior of the JVM using one of the following methods:

- Create a runtime analysis with the JRockit Runtime Analyzer (JRA) in Oracle JRockit Mission Control. In the JRA tool, you can see the frequency of the garbage collections and why garbage the collections are launched. This information provides clues for memory management tuning. For information on creating and analyzing a JRA report, please refer to the online help in the Oracle JRockit Mission Control Client or the Oracle JRockit Mission Control documentation.
- Create verbose outputs by using the command-line option `-Xverbose`; for example, `-Xverbose:memdbg,gcpause,gcreport` will show memory management data like garbage collection frequency and duration. From the JRockit JVM R27.1 and forward, setting `-Xverbose:memdbg` will also show the reason why each garbage collection was started. This will help you study the garbage collection behavior.

Now you have the tools for measuring the throughput of your Java application and can start to tune the JVM for better application throughput.

Select Garbage Collector

The first step of tuning the JRockit JVM for maximum application throughput is to select an appropriate garbage collection mode or strategy.

- Dynamic Garbage Collection Mode Optimized for Throughput

  This is the default garbage collection mode for the JRockit JVM. This mode selects the optimal garbage collection strategy for maximum application throughput.

- Static Generational Parallel Garbage Collection

  This static garbage collector is a good alternative if you do not want to use a dynamic garbage collection mode. The generational parallel garbage collector provides high throughput for applications that allocate a lot of temporary objects.

- Static Single-Spaced Parallel Garbage Collection.

  This is another alternative if you do not want to use a dynamic garbage collection mode. The single-spaced parallel garbage collector provides high throughput for applications that allocate mostly large objects.

For more information about different garbage collector options, see Selecting and Tuning a Garbage Collector.

Dynamic Garbage Collection Mode Optimized for Throughput

The default garbage collection mode in the JRockit JVM (assuming that you run in server mode, which is also default) tunes the memory management for maximum application throughput. Depending on the behavior of your application, it will select either a generational or non-generational parallel garbage collection strategy. It will also tune the nursery size, if the garbage collection strategy is generational.

Be aware that if you use the dynamic garbage collection mode optimized for throughput, the garbage collection pauses will not have any strict time limits. If your application is sensitive to long latencies, you should tune for low latencies rather than for maximum throughput, or find a middle path that gives you acceptable latencies.

The dynamic garbage collection mode optimized for throughput is the default garbage collector for the JRockit JVM. You can also turn it on explicitly like this:

```
java -XgcPrio:throughput myApplication
```

Static Single-Spaced Parallel Garbage Collection

If you want to use a static garbage collector, then you should use a parallel garbage collector in order to maximize application throughput. If the large/small object allocation ratio is high, then use a single-spaced garbage collector (`-Xgc:singlepar`). You can see the ratio between large and small object allocation if you do a JRA recording of your application.

To improve throughput by using a static garbage collector, you may also need to set other `-X` or `-XX` options to deliver that throughput.

Static Generational Parallel Garbage Collection

If you want to maximize application throughput and the large/small object allocation ratio is low, then use a generational parallel garbage collector (`-Xgc:genpar`). A generational parallel

garbage collector might be the right choice even if the large/small object allocation ratio is high when you are using a very small nursery. You can see the ratio between large and small object allocation if you do a JRA recording of your application.

To improve throughput by using a static garbage collector, you may also need to set other `-X` or `-XX` options to deliver that throughput.

Tune the Heap Size

The default heap size starts at 64 MB and can increase up to 1 GB. Most server applications need a large heap—at least larger than 1 GB—to optimize throughput. For such applications, you will need to set the heap size manually by using the `-Xms` (initial heap size) and `-Xmx` (maximum heap size) command-line options. Setting `-Xms` the same size as `-Xmx` has regularly shown to be the best configuration for improving throughput; for example:

```
java -Xms:2g -Xmx:2g myApp
```

For more information on setting the initial and maximum heap sizes, including guidelines for setting these values, please see Optimizing Memory Allocation Performance.

Manually Tune the Nursery Size

The nursery—or young generation —is the area of free chunks in the heap where objects are allocated when running a generational garbage collector (`-XgcPrio:throughput`, `-Xgc:genpar` or `-Xgc:gencon`). A nursery is valuable because most objects in a Java application die young. Collecting garbage from the young space is preferable to collecting the entire heap, as it is a less expensive process and most objects in the young space will already be dead when the garbage collection is started.

If you are using a generational garbage collector you might need to change the nursery setting to accommodate more young objects.

- `-XgcPrio:throughput` and `-Xgc:genpar` will change the nursery size dynamically in runtime. `-XgcPrio:throughput` might even turn off the nursery (that is, switch to a single generational garbage collector). In some cases manual tuning might result in a more efficient nursery size.
- `-Xgc:gencon` has a fairly low and static nursery size setting. For many applications, you may want to tune the nursery size manually when using this garbage collector.

An efficient nursery size is such that the amount of memory freed by young collections (garbage collections of the nursery) rather than old collections (garbage collections of the entire heap) is

as high as possible. To achieve this, you should set the nursery size close to the size of half of the free heap after an old collection.

To set the nursery size manually, use the `-Xns` command-line option; for example:

```
java -Xgc:gencon -Xms:2g -Xmx:2g -Xns:512m myApp
```

Manually Tune Compaction

Compaction is the process of moving chunks of allocated space toward the lower end of the heap, helping to create contiguous free memory at the other end. The JRockit JVM does partial compaction of the heap at each old collection.

The default compaction setting for static garbage collectors (`-Xgc` or `-XXsetGC`) use a dynamic compaction scheme that tries to avoid “peaks” in the compaction times. This is a compromise between keeping garbage collection pauses even and maintaining a good throughput, so it doesn't necessarily give the best possible throughput. Tuning the compaction can pay off well, depending on the application's characteristics.

There are two ways to tune the compaction for better throughput; increasing the size of the compaction area and increasing the compact set limit. Increasing the size of the compaction area will help reduce the fragmentation on the heap. Increasing the compact set limit will implicitly allow larger areas to be compacted at each garbage collection. This reduces the garbage collection frequency and makes allocation of large objects faster, thus improving the throughput.

For information on tuning these compaction options, please refer to Tuning the Compaction of Memory.

Tune the Thread-Local Area Size

Thread Local Areas (TLAs) are chunks of free memory used for object allocation. The TLAs are reserved from the heap and given to the Java threads on demand, so that the Java threads can allocate objects without having to synchronize with the other Java threads for each object allocation.

Increasing the preferred TLA size speeds up allocation of small objects when each Java thread allocates a lot of small objects, as the threads won’t have to synchronize to get a new TLA as often.

In Oracle JRockit JVM R27.3 and later releases the preferred TLA size also determines the size limit for objects allocated in the nursery. Increasing the TLA size will thus also allow larger objects to be allocated in the nursery, which is beneficial for applications that allocate a lot of

large objects. In older versions you need to set both the TLA size and the Large Object Limit to allow larger objects to be allocated in the nursery. A JRA recording will show you statistics on the sizes of large objects allocated by your application. For good performance you can try setting the preferred TLA size at least as large as the largest object allocated by your application.

For more information on how to set the TLA size, see Setting the Thread Local Area Size.

Tuning For Stable Performance

An incorrectly tuned JVM may perform well initially, but start showing lower performance or longer latencies over time or display severe performance variations. This section shows you how to tune your JVM for stable performance over time. The following topics are covered:

- Measuring the Performance Variance
- Tune the Heap Size
- Manually Tune the Nursery Size
- Tune the Garbage Collector
- Tune Compaction

Measuring the Performance Variance

To be able to measure and analyze performance variance over time you need a long-running test that continuously reports the current performance. The test scenario should be as realistic as possible and cover as many use cases as possible.

When you have identified a variance in performance you can start monitoring the Oracle JRockit JVM to see if this variance correlates to events within the JVM, for example garbage collection, fragmentation or lock deflation. The tools in Oracle JRockit Mission Control will help you do this, as well as the verbose outputs that you can enable with the `-Xverbose` command line option.

Events to look for and the proper tools for finding these are listed in Table 14-1

Table 14-1 JVM Events

Event Type	What to Look For	Tools
Heap size change	The heap increases or decreases	`-Xverbose:memdbg`, JRA, Oracle JRockit Mission Control
Nursery size change	The nursery size increases or decreases	`-Xverbose:memdbg`, JRA, Oracle JRockit Mission Control
Garbage collector strategy change	A dynamic garbage collection mode changes the garbage collection strategy	`-Xverbose:memdbg`, JRA
Increased fragmentation	The amount of dark matter increases	JRA
Full compaction	Compaction of all heap parts at once	`-Xverbose:memdbg`, JRA

Tune the Heap Size

Heap size changes in runtime may cause performance variations. You can monitor the heap size in `-Xverbose:memdbg` outputs and in JRockit Mission Control tools. A JRA recording will also tell you if the heap size has changed during the recording.

For an even performance over time, you should set the initial heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`), for example:

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

For more information on tuning the heap size, see Setting the Heap Size.

Manually Tune the Nursery Size

Nursery size changes in runtime may cause performance variations, but may also help keeping the performance high when the load changes. You can monitor the nursery size in `-Xverbose:memdbg` outputs and JRockit Mission Control tools. A JRA recording will also tell you if the nursery size has changed during the recording. If you find that performance variations in your application correlate to nursery size changes, you can set a static nursery size with the command line option `-Xns:<size>`, for example:

```
java -Xns:100m myApplication
```

For more information on tuning the nursery size, see Setting the Nursery and Keep Area Size.

Note: Overriding the dynamic nursery sizing heuristics may have a negative impact on the performance or cause performance variations in applications where the amount of live data varies during the run.

Tune the Garbage Collector

The dynamic garbage collection modes in the JRockit JVM select a garbage collection strategy based on runtime information. Changes in application behavior may cause the garbage collection strategy to change. If such changes happen often and cause a performance variations, you may want to select a static garbage collection strategy rather than a dynamic garbage collection mode. Set a static garbage collection strategy with the command line option `-Xgc:<strategy>`, for example:

```
java -Xgc:parallel myApplication
```

For more information on selecting a static garbage collection strategy, see Selecting a Static Garbage Collection Strategy.

Tune Compaction

The Oracle JRockit JVM uses the mark and sweep garbage collection model as described in The Mark and Sweep Model. This garbage collection model may cause the heap to become fragmented, which means that the free areas on the heap become many but small. The JVM performs partial compaction of the heap at each garbage collection to reduce the fragmentation. Sometimes the amount of compaction isn't enough. This leads to increasing fragmentation, which in turn leads to more and more frequent garbage collections until the heap is so fragmented that a full compaction is performed. After the full compaction the garbage collection frequency goes down, but will gradually increase as the fragmentation increases again.

This behavior will cause the performance of the Java application to vary. As the garbage collection frequency increases the performance drops. During the full compaction you may experience a prolonged garbage collection pause, which pauses the entire Java application for a while. After this the performance is high again, but starts going down as the garbage collection frequency increases again.

You can monitor the compaction ratio and garbage collection frequency in `-Xverbose:memdbg` outputs, the Management Console and JRA recordings. A JRA recording will also show you how much dark matter (severe fragmentation) there is on the heap. If you find that the garbage

collection keeps increasing until a full compaction is done, you need to increase the compaction ratio. For information on how to tune the compaction, see Tuning the Compaction of Memory.

You can also decrease the fragmentation on the heap by using a generational garbage collector. See Selecting and Tuning a Garbage Collector for information on different garbage collectors.

CHAPTER 15

Tuning For a Small Memory Footprint

If you are running on a system with limited memory resources, you may want to tune the Oracle JRockit JVM for a small memory footprint. This section describes the tuning options you have available for reducing the memory footprint of the JVM. The following topics are covered:

- Measuring the Memory Footprint
- Set the Heap Size
- Select a Garbage Collector
- Tune Compaction
- Tune Object Allocation

Measuring the Memory Footprint

The memory footprint of an application is best measured using some of the tools provided with the operating system, for example the top shell command or the Task Manager in Windows.

To determine how the memory usage of the JVM process is distributed, you can request a memory analysis by using `jrcmd` to print the JVM's memory usage. See Using jrcmd and Available Diagnostic Commands for more information.

When you have acquired information on the JVM's memory usage you can start tuning the JVM to reduce the memory footprint within the areas that use the most memory.

Set the Heap Size

The most obvious place to start tuning the memory footprint is the Java heap size. If you reduce the Java heap size by a certain amount you will reduce the memory footprint of the Java process by the same amount. You can however not reduce the Java heap size infinitely. The heap must be at least large enough for all objects that are alive at the same time. Preferably the heap should be at least twice the size of the total amount of live objects, or large enough so that the JVM spends less time garbage collecting the heap than running Java code.

The heap size is set with the command line options `-Xms` (initial heap size) and `-Xmx` (maximum heap size); for example:

```
java -Xms:100m -Xmx:100m myApplication
```

To allow the heap to grow and shrink depending on the amount of free memory in your system, set `-Xms` lower than `-Xmx`. For more information on setting the heap size, see Optimizing Memory Allocation Performance.

Select a Garbage Collector

The choice of a garbage collection mode or static strategy does not in itself affect memory footprint noticeably, but choosing the right garbage collection strategy may allow you to reduce the heap size without a major performance degradation.

If your application uses a lot of temporary objects you should consider using a generational garbage collection strategy. The use of a nursery reduces fragmentation and thus allows for a smaller heap.

The concurrent garbage collector must start garbage collections before the heap is entirely full, to allow Java threads to continue allocating objects during the garbage collection. This means that the concurrent garbage collector requires a larger heap than the parallel garbage collector, and thus your primary choice for a small memory footprint is a parallel garbage collector.

The default garbage collection mode chooses between a generational parallel garbage collection strategy and a non-generational parallel garbage collection strategy, depending on the sizes of the objects that your application allocate. This means that the default garbage collector is a good choice when you want to minimize the memory footprint.

If you want to use a static garbage collection strategy, you can specify the strategy with the `-Xgc` command line option; for example:

```
java -Xgc:genpar myApplication
```

For more information on selecting a garbage collector, see Selecting and Tuning a Garbage Collector.

Tune Compaction

Using a small heap increases the risk for fragmentation on the heap. Fragmentation can have a severe effect on application performance, both by lowering the throughput and by causing occasional long garbage collections when the garbage collector is forced to compact the entire heap at once.

If you are experiencing problems with fragmentation on the heap you can increase the compaction ratio by using the command line option `-XXcompactRatio:<percentage>`, for example:

```
java -XXcompactRatio:50 myApplication
```

If your application isn't sensitive to long latencies, you can try using full compaction. This will allow you to use a smaller heap, as all fragmentation is eliminated at each garbage collection. Enable full compaction by using the command line option `-XXfullCompaction`; for example:

```
java -XXfullCompaction myApplication
```

Compaction uses memory outside of the heap for bookkeeping. As an alternative to increasing the compaction you can use a generational garbage collector, which also reduces the fragmentation.

Tune Object Allocation

You can tune the object allocation to allow smaller chunks of free memory to be used for allocation. This reduces the negative effects of fragmentation, and allows you to run with a smaller heap. The smallest chunk of memory used for object allocation is a thread local area. Free chunks smaller than the minimum thread local area size are ignored by the garbage collector and become *dark matter* until a later garbage collection frees some adjacent memory or compacts the area to create larger free chunks. You can reduce the minimum thread local area size with the command line option `-XXtlaSize:min=<size>`, for example:

```
java -XXtlaSize:min=1k myApplication
```

In releases older than R27.2 you reduce the TLA size with the command line option `-XXtlaSize:<size>`, for example:

```
java -XXtlaSize:1k myApplication
```

For more information on how to set the thread local area size, see the documentation on `-XXtlaSize` and Optimizing Memory Allocation Performance.

Tuning For Faster JVM Startup

Small utility applications that run only for a short time may suffer a performance hit if the JVM and Java application startup time is long. The Oracle JRockit JVM is by default optimized for server use, which means that the startup times can be longer in favour of high performance as soon as the application is up and running. This section describes how to tune the JVM to decrease the startup times, covering the following topics:

- Measuring the Startup Time
- Setting the Heap Size
- Troubleshoot Your Application and the JVM

Measuring the Startup Time

The startup time of an application is the time it takes for the application to get up and running and ready to start doing what it is supposed to do. The startup time includes both the JVM startup and the Java application startup.

For information on how to measure the startup time of your application, see Timing with nanoTime() and currentTimeMillis().

Setting the Heap Size

The heap size has an impact on both the JVM startup time and the Java application startup time. The JVM reserves memory for the maximum heap size (`-Xmx`) and commits memory for the initial heap size (`-Xms`) at startup, which takes time. For large applications this is inevitable, but

you should be aware that using an oversized heap may lead to longer JVM startup times than necessary. If your application is small and runs only for a short time you may have to set a small heap size to avoid the overhead of reserving and committing more memory than the application will ever need.

On the other hand, if the initial heap is too small, the Java application startup becomes slow as the JVM is forced to perform garbage collection frequently until the heap has grown to a more reasonable size. For optimal startup performance you should set the initial heap size to the same as the maximum heap size.

Troubleshoot Your Application and the JVM

The application itself may be causing the startup to become slow. See The Oracle JRockit JVM Starts Slowly for tips on troubleshooting problems in the application and JVM.

Part III JRockit JDK Tools

CHAPTER 17

Introduction to Diagnostics Tools

Throughout the Oracle JRockit JVM Diagnostics Guide, you will be directed to use certain tools or other features of the Oracle JRockit JVM to better identify and resolve problems when running an application with the JVM. The chapters in this section provide an overview of these tools along with instructions for using them.

What this Section Contains

These tools and features include:

- Monitoring, management and analysis tools that are included in Oracle JRockit Mission Control. JRockit Mission Control contains a suite of tools that help you monitor, manage, profile, and eliminate memory leaks in your Java application without causing undue performance overhead (see Using Oracle JRockit Mission Control Tools).
- Customizable verbose logs that provide low overhead runtime information on various components of the JRockit JVM, for example memory management and code optimizations. (see Understanding Verbose Outputs).
- Thread dumps, snapshots of the state of all threads that are part of the process. These dumps reveal information about an application's thread activity, which can help you diagnose problems and better optimize application and JVM performance (see Using Thread Dumps).
- Ctrl-Break Handlers, which allow you to interrupt processing to print information about running processes or communicate directly with the JRockit JVM. You can easily send

Ctrl-Break handler commands “on the fly” to a running JVM process by using jrcmd (see Running Diagnostic Commands).

- Time Zone Updater, required for you to update installed JDK/JRE images with more recent time zone data to accommodate the U.S. 2007 daylight saving time changes (US2007DST) originating with the U.S. Energy Policy Act of 2005. (see Oracle JRockit Time Zone Updater).
- Instructions for Oracle JRockit Mission Control 1.0 users who want to create a JRockit Runtime Analyzer recording. The JRA provides a wealth of information on internals in the JRockit JVM that you will find of great interest if you are using this product as your runtime VM (see Creating a JRA Recording with JRockit Mission Control 1.0).
- Oracle JRockit Mission Control Use Cases demonstrates various ways Oracle JRockit Mission Control can be used to monitor and manage application running on the JRockit JVM. It includes use cases describing how to use:
  - The JRockit Management Console
  - The JRockit Runtime Analyzer (JRA)
  - The JRockit Memory Leak Detector (Memleak)

CHAPTER 18

Using Oracle JRockit Mission Control Tools

The suite of tools included in Oracle JRockit Mission Control are designed to monitor, manage, profile, and gain insight into problems occurring in your Java application without requiring the performance overhead normally associated with these types of tools.

This chapter serves as a generic introduction to the different versions of JRockit Mission Control. You can find more detailed information about the versions, please refer to More Information on JRockit Mission Control Versions.

This chapter contains information on these subjects:

- JRockit Mission Control Overhead
- Architectural Overview of the JRockit Mission Control Client
- The JRockit Management Console
- The JRockit Runtime Analyzer
- The JRockit Memory Leak Detector
- More Information on JRockit Mission Control Versions

JRockit Mission Control Overhead

JRockit Mission Control's low performance overhead is a result of using data collected as part of the Oracle JRockit JVM's normal adaptive dynamic optimization. This also eliminates the problem with the Heisenberg anomaly that can occur when tools using bytecode instrumentation alters the execution characteristics of the system. JRockit Mission Control functionality is always

available on-demand and the small performance overhead is only in effect while the tools are running.

Architectural Overview of the JRockit Mission Control Client

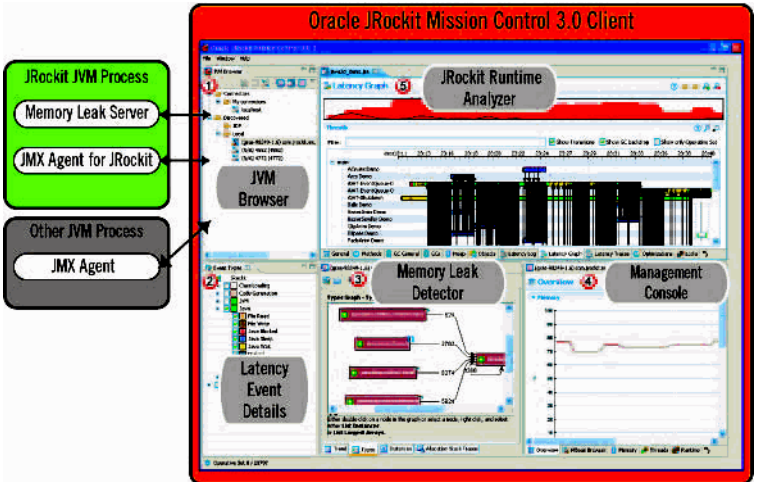
This section provides an architectural overview of all versions of JRockit Mission Control.

- JRockit Mission Control 3.0
- JRockit Mission Control 2.0
- JRockit Mission Control 1.0

JRockit Mission Control 3.0

With the Rich Client Platform (RCP) based JRockit Mission Control Client, you can launch the JRockit Memory Leak Detector, the JRockit Runtime Analyzer, and the JRockit Management Console from within the JRockit Mission Control Client. Figure 18-3 depicts how the JRockit Mission Control Client looks when all tools are loaded.

Figure 18-1 Architectural Overview of the JRockit Mission Control 3 Client



When a JRA recording is started from within the JRockit Mission Control Client, it records the status of the JRockit JVM process for the time that you have specified and creates a ZIP file

containing an XML file with the recorded data and optionally a binary file with latency data together with the corresponding data producer specification files. The ZIP file is automatically opened in the JRockit Runtime Analyzer (marked 5 in Figure 18-3) upon completion of the recording for JDK level 1.5 and later; for JDK 1.4.2 it is stored locally on the computer where the recorded JVM was running. Typical information that is recorded during a JRA recording is Java heap distribution, garbage collections, method samples, and lock profiling information (optional). New for the JRockit Mission Control 3.0 release, is that you can also record thread latency data. When viewing Latency data in the JRA Tool, the Latency Events Details become visible (marked 2 in Figure 18-3).

To view real-time behavior of your application and of the JRockit JVM, you can connect to an instance of the JVM and view real-time information through the JRockit Management Console (marked 4 in Figure 18-3). Typical data that you can view is thread usage, CPU usage, and memory usage. All graphs are configurable and you can both add your own attributes and redefine their respective labels. In the Management Console you can also create rules that trigger on certain events, for example sending an e-mail if the CPU load reaches 90%.

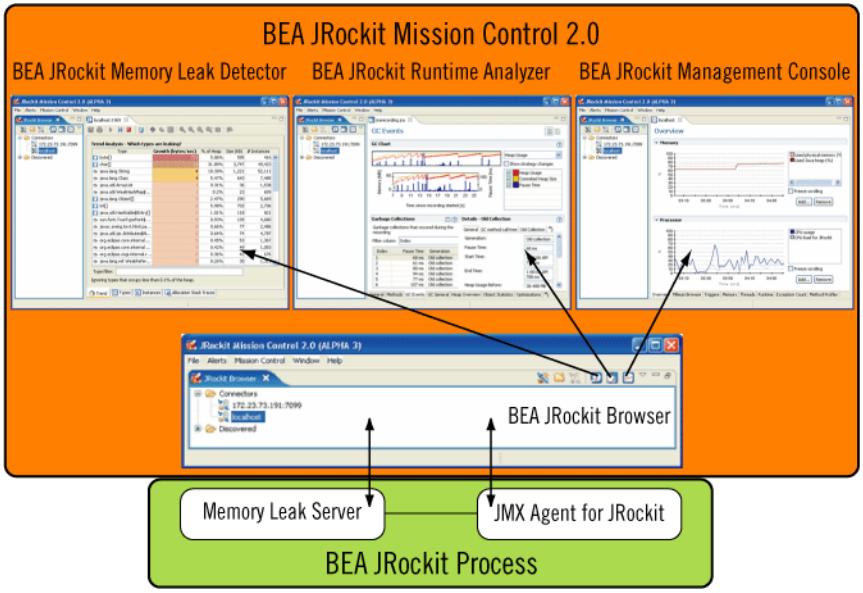
With the JMX Agent you have access to all MBeans deployed in the platform MBean server. From these MBeans, you can read attribute information, such as garbage collection durations.

To find memory leaks in your Java application, you connect the JRockit Memory Leak Detector to the running JRockit JVM process. The JRockit Memory Leak Detector connects to the JMX (RMP) Agent that instructs to start a Memory Leak server with which all further communication takes place.

JRockit Mission Control 2.0

With the new Client Platform (RCP) based JRockit Mission Control Client, you can launch the JRockit Memory Leak Detector, the JRockit Runtime Analyzer, and the JRockit Management Console from within the JRockit Mission Control Client (see Figure 18-2).

Figure 18-2 Architectural Overview of the JRockit Mission Control 2.0 Client



Through the JMX Agent, you have access to all MBeans deployed in the platform MBean server. From these MBeans, you can read attributes information, such as garbage collection duration.

When a JRA recording is started from within the JRockit Mission Control Client, it records the status of the JRockit JVM process for the time that you have specified and creates an XML file. This file is automatically opened in the JRockit Runtime Analyzer. Typical information that is recorded during a JRA recording is Java heap distribution, garbage collections, method optimizations, and method profiling information.

To find memory leaks in your Java application, you connect the JRockit Memory Leak Detector to the running JRockit JVM process. The Memory Leak Detector connects to the JMX (RMP) Agent that instructs to start a Memory Leak server with which all further communication takes place.

JRockit Mission Control 1.0

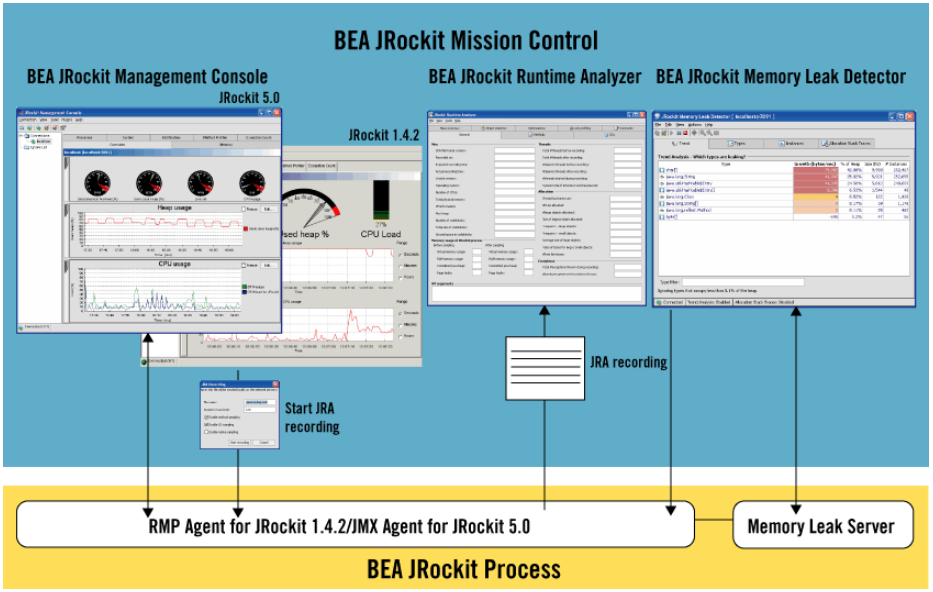
JRockit Mission Control 1.0 is available on the JRockit JDK 1.4.2 (R26.2 and later) and JRockit JDK 5.0 (R26.0 and later), see Figure 18-3. The difference between the two is the connection agent used by the JRockit Management Console and the JRockit Management Console user interface itself.

The RMP Agent (JRockit JDK 1.4.2) provides access, among other things, to live data about memory and CPU usage. With the addition of the JMX Agent (available with JRockit JDK 5.0) you will also get access to MBeans available to the platform MBean server. From these MBeans, you can read attributes information, such as garbage collection pauses.

When a JRA recording is started, for example, from the Management Console, it records the status of the JRockit JVM process for the time that you have specified. When the recording is completed, the information is saved to an XML file. This XML file can be viewed and analyzed in the JRockit Runtime Analyzer. Typical information that is recorded during a JRA recording is Java heap distribution, garbage collections, and method optimizations.

To find memory leaks in your Java application, you connect the JRockit Memory Leak Detector to the running JRockit JVM process. The Memory Leak Detector connects to the JMX (RMP) Agent that instructs to start a Memory Leak server where all further communication takes place.

Figure 18-3 Architectural Overview of JRockit Mission Control 1.0



The JRockit Browser (JRockit Mission Control 2.0 and later)

The JRockit Browser is available only with JRockit Mission Control 2.0 and later versions. This tool allows you to set up and manage all running instances of the JRockit JVM on your system. From the JRockit Browser you activate recordings, set up a tree view of different JRockit JVMs to monitor, start other JRockit Mission Control tools, etc. Each JRockit JVM instance is referred to as a Connector.

The JRockit Management Console

The JRockit Management Console is used to monitor and manage multiple (or single) JRockit JVM instances. It captures and presents live data about memory, CPU usage, and other runtime metrics. For the Management Console that is running on JRockit JDK 5.0, information from any JMX MBean deployed in the JRockit JVM internal MBean server (JMX Agent in Figure 18-3) can be displayed as well. JVM management includes dynamic control over CPU affinity, garbage collection strategy, memory pool sizes, and more.

The JRockit Runtime Analyzer

The JRockit Runtime Analyzer (JRA) is an on-demand “flight recorder” that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line by using the JRA. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, and latency analysis (JRockit Mission Control 3.0).

Latency Analysis Tool (JRockit Mission Control 3.0)

The Latency Analysis Tool is a subset of the JRockit Mission Control 3.0 version of the JRA that allows you to create JRA recordings that contain latency information for your application. Latency events occur when thread execution stops temporarily, for example when a thread waits for its turn to enter a synchronized method, or waits for data from a socket. The JRA now contains three additional tabs that all show latency data from different perspectives. These tabs are prefixed *Latency* and named: *Latency Log*, *Latency Graph*, and *Latency Traces*. Together with these three tabs and two auxiliary tabs, you can activate and deactivate event types on the latency tabs and view properties.

JRA Sample Recordings

Beginning with the JRockit JVM R27.5 and JRockit Mission Control 3.0.2, you can access three sample JRA recordings that demonstrate the features of the Latency Analysis Tool. The files are located at `JROCKIT_HOME/missioncontrol/samples/jrarecordings/`. They are:

- `pricing_server_logging_on.jra` — this recording shows an application experiencing problems with latencies because several threads try to access a `java.util.logging Logger`.
- `pricing_server_logging_off.jra` — this file contains a recording of the same application in `pricing_server_logging_on.jra`, but with logging turned off.
- `java2d_demo.jra` — this recording shows how to use the Jump-to-Source feature. It is a recording of the demo located at `JROCKIT_HOME/demo/jfc/Java2D`. The Java2D demo folder contains the source, allowing this recording to demonstrate Jump-to-Source.

  Note: Jump-to-Source is only available when running the JRockit Mission Control Client within Eclipse. To run the JRockit Mission Control Client within Eclipse, please install it from the update site. For more information see `dev2dev.bea.com/jrockit/tools.html`. You also need to set up a Java project containing the source you wish to jump to; in this case the Java2D demo.

Opening a Sample Recording

You can open a sample either from within the JRockit Mission Control Client or directly from the file system, as described below:

To open a sample recording from within the JRockit Mission Control Client

With the JRockit Mission Control Client running, do the following:

1. Open the **File** menu and select **Open File...**

   The Open File dialog box appears, showing the `JROCKIT_HOME/missioncontrol/samples/jrarecordings/` folder (the default folder).

2. Select the sample recording you want to open and click **Open**.

   The recording opens.

To open a sample recording from the file system

With the JRockit Mission Control Client running and your file system (for example, Windows Explorer) open, do the following:

1. In the file system, navigate to `JROCKIT_HOME/missioncontrol/samples/jrarecordings/` and select the recording you want to open.
2. Drag the recording from the file system directly onto the JRockit Mission Control Client

   The recording opens.

The JRockit Memory Leak Detector

The JRockit Memory Leak Detector is a tool for discovering and finding the cause for memory leaks in a Java application. The JRockit Memory Leak Detector's trend analyzer discovers slow leaks, it shows detailed heap statistics (including referring types and instances to leaking objects), allocation sites, and it provides a quick drill down to the cause of the memory leak. The Memory Leak Detector uses advanced graphical presentation techniques to make it easier to navigate and understand the sometimes complex information.

More Information on JRockit Mission Control Versions

Complete information on using JRockit Mission Control is available in the respective versions documentation. Because of the difference in deployment mechanisms between JRockit Mission Control 1.0, JRockit Mission Control 2.0, and JRockit Mission Control 3.0, each version has its own set of documentation:

- For JRockit Mission Control 1.0, please refer to the JRockit Mission Control documentation.
- For JRockit Mission Control 2.0, please refer to the online help documentation included with the JRockit Mission Control 2 GUI.
- For JRockit Mission Control 3.0, please refer to the built-in help documentation as well as eDocs. For PDF versions of the help, see the Oracle JRockit Mission Control documentation.

CHAPTER 19

Understanding Verbose Outputs

The `-Xverbose` command line option enables verbose outputs from the Oracle JRockit JVM. You can use these verbose outputs for monitoring, tuning and diagnostics. This chapter describes some of the most useful verbose modules and how to interpret the outputs from them.

Note: Outputs may differ between JVM versions and that the exact format is subject to changes at any time.

This chapter is divided into two sections:

- Memory Management Verbose Log Modules
- Other Verbose Log Modules

Memory Management Verbose Log Modules

Many of the verbose modules available in the JRockit JVM are dedicated to memory management and garbage collection.Table 19-1 lists the verbose modules described in this section.

Table 19-1 Memory Management Verbose Modules

Module	Description	Uses
`memory`	Basic garbage collection information	Tuning and monitoring
`nursery`	Nursery details	Tuning and monitoring

Table 19-1 Memory Management Verbose Modules

Module	Description	Uses
`memdbg`	Memory management details	Tuning, monitoring and diagnostics
`compaction`	Compaction details	Tuning, monitoring and diagnostics
`gcpause`	Garbage collection pause times	Tuning, monitoring and diagnostics
`gcreport`	Garbage collection summary	Tuning and monitoring
`refobj`	Reference object information (R27.5)	Monitoring and diagnostics
`referents`	Reference object information	Monitoring and diagnostics

Most verbose modules are available in several log levels. This section covers only the default (`info`) log level, which is the most useful for general tuning and monitoring purposes.

Verbose Memory Module

The `-Xverbose:memory` (or `-Xverbose:gc`) module provides basic information on garbage collection events. The overhead for this module is very low, which means that you can enable it even in a production environment.

Initial Verbose Memory Outputs

At JVM startup the `memory` log module outputs some basic numbers on the memory management system configuration and a guide to how to read the garbage collection printouts.

Listing 19-1 shows an example of the initial output from the `memory` log module. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-1 Initial Verbose Memory Output

```
1: [memory ] GC mode: Garbage collection optimized for throughput, initial
strategy: Generational Parallel Mark & Sweep

2: [memory ] heap size: 307200K, maximal heap size: 307200K, nursery size:
153600K

3: [memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms
```

```
4: [memory ] <s/start> - start time of collection (seconds since jvm start)
5: [memory ] <end>     - end time of collection (seconds since jvm start)
6: [memory ] <before>  - memory used by objects before collection (KB)
7: [memory ] <after>   - memory used by objects after collection (KB)
8: [memory ] <heap>    - size of heap after collection (KB)
9: [memory ] <pause>   - total sum of pauses during collection (milliseconds)
10: [memory ]             run with -Xverbose:gcpause to see individual pauses
```

Line 1 describes the garbage collection mode used in this run, as well as the initial garbage collection strategy.

Line 2 shows you the initial and maximum heap size, as well as the initial nursery size.

Lines 3-10 describe the format of the garbage collection outputs.

Verbose Memory Garbage Collection Outputs

The `memory` log module prints out a line for each garbage collection.

Listing 19-2 shows a snippet of verbose outputs from the `memory` log module. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-2 Verbose Memory Garbage Collection Outputs

```
1: [memory ] 83.976: parallel nursery GC 307200K->191044K (307200K), 35.584 ms
2: [memory ] 84.772-85.766: GC 288200K->73425K (307200K), 994.090 ms
3: [memory ] 87.036: parallel nursery GC 233783K->119655K (307200K), 57.716 ms
4: [memory ] 87.974: parallel nursery GC 233655K->119527K (307200K), 35.876 ms
```

Each garbage collection output starts with a timestamp, which is the time in seconds since the JVM started. At the end of each line you can also see the total heap size within parenthesis, and the total garbage collection duration for each garbage collection. Note that the garbage collection duration may consist of both pauses and concurrent garbage collection phases.

Lines 1, 3 and 4 are outputs from young collections. The young collection on line 1 reduced the amount of occupied heap from 307200 KB to 191044 KB.

Line 2 is an output from an old collection, which reduced the amount of occupied heap from 288200 KB to 73425 KB.

Verbose Memory Page Faults Warning

Page faults cause memory access to become slow, and page faults during garbage collection may cause long garbage collection times. Because of this, the verbose `memory` log module prints out a warning whenever the number of page faults during the garbage collection was more than 5% of the number of pages in the heap.

Listing 19-3 shows an example of a page fault warning. This example is from the JRockit JVM R27.4.

Listing 19-3 Verbose Memory Page Faults Warning

```
[memory ] Warning: Your computer has generated 9435 page faults during the last
garbage collection.

[memory ] If you find this swapping problematic, please consider running JRockit
with a smaller heap.
```

Verbose Nursery Log Module

The `-Xverbose:nursery` log module provides details on young collections and nursery sizing. Some of this information is useful for monitoring and tuning the JRockit JVM. The overhead is very low, which means that you can enable this module even in a production environment.

The `nursery` log module is available in the JRockit JVM R27.2 and later releases.

Verbose Nursery Young Collection Output

Listing 19-4 shows an example of an output from the `nursery` log module during a young collection. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-4 Verbose Nursery Young Collection Output

```
1: [nursery] KeepAreaStart: 0x015FFFF0 KeepAreaEnd: 0x01AFFFF0

2: [nursery] Young collection 86 started. This YC is running while the OC is in
phase: not running.

3: [nursery] Setting mmNurseryMarker[0] to 0x015FFFF0

4: [nursery] Setting mmNurseryMarker[1] to 0x01AFFFF0
```

Lines 1, 3 and 4 contain information related to the keep area size and position. This information is only interesting for advanced diagnostics.

Line 2 shows you the sequence number of the young collection. It also informs you that the old collection is not running while this young collection is running. Other possible old collection phases are marking, precleaning and sweeping, which are the concurrent phases of the concurrent old collection.

Verbose Nursery Size Adjustment Output

Some garbage collection modes and strategies will adjust the nursery size in runtime for optimal performance. The nursery log module shows some information on the nursery sizing calculations and nursery size changes.

Listing 19-5 shows an example of nursery sizing outputs from an old collection. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-5 Verbose Nursery Size Adjustment Output

```
1: [nursery] Optimal nursery size: 157286400, free heap: 314572800

2: [nursery] Nursery size increased from 0kb to 153600kb. Parts: 4023
```

Line 1 shows you that the heuristics have found that an optimal nursery size would be 157286400 bytes. The memory management system may however not be able to create a nursery of exactly this size.

Line 2 states that the nursery size is being increased from 0 (no nursery) to 153600 KB, and that the nursery consists of 4023 chunks.

Verbose Memdbg Log Module

The `-Xverbose:memdbg` log module provides details on garbage collection and memory management. Enabling this log module also enables some other log modules, for example the nursery log module. This section only describes the outputs labeled `memdbg`, all other verbose outputs have been removed from the examples.

The overhead of the verbose `memdbg` outputs is low, and thus it can be used in production environments.

Initial Verbose Memdbg Output

Listing 19-6 shows an example of the initial output from the `memdbg` verbose module. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-6 Initial Verbose Memdbg Output

```
1: [memdbg ] Memory layout after heap allocation:
2: [memdbg ] ' ' - free, '-' - OS reserved range, 'r' - reserved,  'x' - committed,
3: [memdbg ] '+' - committed/read, 'e' - committed/executable
4: [memdbg ] 'J' - java heap, 'j' - java heap data structures
5: [memdbg ] 00 0.00Gb
-eeJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
6: [memdbg ] 10 0.25Gb JJJJJJJJJJJJJ
7: [memdbg ] 20 0.50Gb
8: [memdbg ] 30 0.75Gb
9: [memdbg ] 40 1.00Gb
10: [memdbg ] 50 1.25Gb
11: [memdbg ] 60 1.50Gb
12: [memdbg ] 70 1.75Gb      e                 e    -              e e e   ejj
```

```
13: [memdbg ] 80 2.00Gb
----------------------------------------------------------------

14: [memdbg ] 90 2.25Gb
----------------------------------------------------------------

15: [memdbg ] a0 2.50Gb
----------------------------------------------------------------

16: [memdbg ] b0 2.75Gb
----------------------------------------------------------------

17: [memdbg ] c0 3.00Gb
----------------------------------------------------------------

18: [memdbg ] d0 3.25Gb
----------------------------------------------------------------

19: [memdbg ] e0 3.50Gb
----------------------------------------------------------------

20: [memdbg ] f0 3.75Gb
----------------------------------------------------------------

21: [memdbg ] Minimum TLA size is 2048 bytes

22: [memdbg ] Preferred TLA size is 38912 bytes

23: [memdbg ] Large object limit is 2048 bytes

24: [memdbg ] Minimal blocksize on the freelist is 2048 bytes

25: [memdbg ] Initial and maximum number of gc threads: 2, of which 2 parallel
threads, 1 concurrent threads, and 2 yc threads

26: [memdbg ] Prefetch distance in balanced tree: 4

27: [memdbg ] Using prefetch linesize: 32 bytes  chunks: 512 bytes pf_dist: 64
bytes
```

Lines 1-20 describe the memory lay-out in the machine after the Java heap has been allocated. This information can be used for diagnosing problems related to heap positioning.

Lines 21-24 show you the minimum and preferred Thread Local Area sizes, the large object limit and the minimum block size on the freelist. These values may depend on the heap size and the garbage collector, as well as the JRockit JVM version.

Line 25 informs you of the number of garbage collection threads.

Line 26 and 27 contain information on prefetching, which is mostly useful for advanced diagnosing and tuning.

Verbose Memdbg Parallel Old Collection Output

The `memdbg` module adds a lot of useful information to the garbage collection outputs. For most tuning and diagnosing, this information is essential. The outputs differ between garbage collection types.

Listing 19-7 shows an example of a verbose memdbg output from a single generational parallel old collection. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-7 Verbose Memdbg Parallel Old Collection Output

```
1: [memdbg ] GC reason: System.gc() called, cause: System.gc()
2: [memdbg ] Stopping of javathreads took 0.213 ms
3: [memdbg ] old collection 1 started
4: [memdbg ] Alloc Queue size before GC: 0, tlas: 0, oldest: 0
5: [memdbg ] Compacting 8 heap parts at index 120 (type external) (exceptional
false)
6: [memdbg ] Starting parallel marking phase.
7: [memdbg ] Hard handles: Processed 1946 handles during normal processing.
8: [memdbg ] Weak handles: Processed 104 handles during normal processing.
9: [memdbg ] total mark time: 7.440 ms
10: [memdbg ] ending marking phase
11: [memdbg ] starting parallel sweeping phase
12: [memdbg ] total sweep time: 3.844 ms
13: [memdbg ] ending sweeping phase
14: [memdbg ] Alloc Queue size after GC: 0, tlas: 0, oldest: 0
15: [memdbg ] Page faults before GC: 6618, page faults after GC: 7938, pagesin
heap: 76800
16: [memdbg ] Restarting of javathreads took 0.023 ms
```

Line 1 displays the reason for the garbage collection. In this example the garbage collection was triggered by a `System.gc()` call.

Line 2 displays the time it took to stop all Java threads for garbage collection, in this case 0.213 ms. This information is useful for latency diagnosing.

Line 3 shows the sequence number of the garbage collection.

Line 4 contains information on the object allocation queue status at the start of the garbage collection. The allocation queue contains all pending object allocation requests that have not yet been satisfied. In this example the allocation queue is empty, which is normal when the garbage collection is started by `System.gc()`.

Line 5 shows you the compaction planned for this garbage collection. In this example 8 heap parts will be compacted starting at heap part 120. The compaction type is external, which means that objects are moved out of the compaction area, and the compaction is not exceptional, which means that it may be aborted or interrupted if the compaction heuristics decide that it will take too long to perform the compaction.

Line 6 marks the start of the mark phase.

Lines 7 and 8 show information on weak and hard handles. This is mostly useful for advanced diagnostics and monitoring.

Line 9 shows you the total time for the mark phase. This time can include both pauses and concurrent phases, although in this case the entire mark phase is done while the Java threads are paused.

Line 10 marks the end of the mark phase.

Line 11 marks the start of the sweep phase.

Line 12 shows you the total time for the sweep phase. This time can include both pauses and concurrent phases, although in this case the entire mark phase is done while the Java threads are paused.

Line 13 marks the end of the sweep phase.

Line 14 displays information on the status of the object allocation queue after the garbage collection. You can compare this information to the information on line 4 to get an idea of how well object allocation is faring. Many objects in the allocation queue at the end of a garbage collection may be an indication that object allocation is difficult, for example due to heavy fragmentation.

Line 15 shows statistics on page faults before and after the garbage collection. Page faults during the garbage collection may slow down the garbage collection severely.

Line 16 displays the time it took to restart all Java threads after garbage collection. This information is useful for latency diagnosing.

Verbose Memdbg Concurrent Old Collection Output

The memdbg module adds a lot of useful information to the garbage collection outputs. For most tuning and diagnosing, this information is essential. The outputs differ between garbage collection types.

Listing 19-8 shows an example of a verbose memdbg output from a single generational mostly concurrent old collection. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-8 Verbose Memdbg Concurrent Old Collection Output

```
1: [memdbg ] GC reason: GC trigger reached, cause: Heap too full
2: [memdbg ] Stopping of javathreads took 0.425 ms
3: [memdbg ] old collection 5 started
4: [memdbg ] Alloc Queue size before GC: 0, tlas: 0, oldest: 0
5: [memdbg ] Compacting 16 heap parts at index 112 (type internal) (exceptional
false)
6: [memdbg ] Starting initial marking phase (OC1).
7: [memdbg ] Restarting of javathreads took 31.908 ms
8: [memdbg ] Starting concurrent marking phase (OC2).
9: [memdbg ] Hard handles: Processed 4251 handles during concurrent processing.
10: [memdbg ] Starting precleaning phase (OC3).
11: [memdbg ] Weak handles: Processed 146 handles during concurrent processing.
12: [memdbg ] Stopping of javathreads took 0.219 ms
13: [memdbg ] Starting final marking phase (OC4).
14: [memdbg ] Hard handles: Processed 8 handles during remaining processing.
15: [memdbg ] Weak handles: Processed 40 handles during remaining processing.
16: [memdbg ] total concurrent mark time: 512.526 ms
```

```
17: [memdbg ] ending marking phase

18: [memdbg ] starting concurrent sweeping phase

19: [memdbg ] total concurrent sweep time: 54.623 ms

20: [memdbg ] ending sweeping phase

21: [memdbg ] Alloc Queue size after GC: 0, tlas: 0, oldest: 0

22: [memdbg ] gc-trigger is 13.200 %

23: [memdbg ] Page faults before GC: 89592, page faults after GC: 92886, pages
in heap: 76800

24: [memdbg ] Restarting of javathreads took 0.030 ms
```

Line 1 displays the reason for the garbage collection. In this example the garbage collection was started because the heap occupancy reached the limit for when a concurrent garbage collection should be started.

Line 2 displays the time it took to stop all Java threads for garbage collection, in this case 0.425 ms. Similar information is displayed on line 12. This information is useful for latency diagnosing.

Line 3 shows the sequence number of the garbage collection.

Line 4 contains information on the object allocation queue status at the start of the garbage collection. The allocation queue contains all pending object allocation requests that have not yet been satisfied. In this example the allocation queue is empty.

Line 5 shows you the compaction planned for this garbage collection. In this example 16 heap parts will be compacted starting at heap part 112. The compaction type is internal, which means that objects are moved within the compaction area, and the compaction is not exceptional, which means that it may be aborted or interrupted if the compaction heuristics decide that it will take too long to perform the compaction.

Line 6 marks the start of the initial mark phase.

Line 7 displays the time it took to start the Java threads for the concurrent marking phase. Similar information is displayed on line 24.

Line 8 marks the start of the concurrent mark phase.

Lines 9, 11, 14 and 15 show information on weak and hard handles. This is mostly useful for advanced diagnostics and monitoring.

Line 10 marks the start of the concurrent precleaning phase.

Line 13 marks the start of the final marking phase.

Line 16 shows you the total time for the mark phase. This time includes both pauses and concurrent phases.

Line 17 marks the end of the mark phase.

Line 18 marks the start of the sweep phase.

Line 19 shows you the total time for the sweep phase. This time includes both pauses and concurrent phases.

Line 20 marks the end of the sweep phase.

Line 21 displays information on the status of the object allocation queue after the garbage collection. You can compare this information to the information on line 4 to get an idea of how well object allocation is faring. Many objects in the allocation queue at the end of a garbage collection may be an indication that object allocation is difficult, for example due to heavy fragmentation.

Line 22 displays the value of the GC trigger. The next concurrent old collection will start when less than this percentage of the heap is free.

Line 23 shows statistics on page faults before and after the garbage collection. Page faults during the garbage collection may slow down the garbage collection severely.

Verbose Memdbg Young Collection Output

Listing 19-9 shows an example of a verbose memdbg output from a young collection. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-9 Verbose Memdbg Young Collection Output

```
1: [memdbg ] GC reason: TLA allocation failed, cause: Get TLA From Nursery
2: [memdbg ] Stopping of javathreads took 0.198 ms
3: [memdbg ] Hard handles: Processed 4259 handles during normal processing.
4: [memdbg ] Weak handles: Processed 144 handles during normal processing.
5: [memdbg ] nursery GC 86: promoted 48984 objects (1718K) in 39.310 ms
```

```
6: [memdbg ] Page faults before GC: 89240, page faults after GC: 89754, pages in
heap: 76800

7: [memdbg ] Nursery size after YC: 20971520

8: [memdbg ] Restarting of javathreads took 0.030 ms
```

Line 1 displays the reason for the garbage collection. In this example the garbage collection was started because allocation of a thread local area failed. This is the normal reason for starting a young collection.

Line 2 displays the time it took to stop all Java threads for garbage collection, in this case 0.198 ms. This information is useful for latency diagnosing.

Lines 3 and 4 show information on weak and hard handles. This is mostly useful for advanced diagnostics and monitoring.

Line 5 shows details on the young collection. 48984 objects of a total size of 1718 KB were promoted during this young collection. The young collection took 39.310 ms.

Line 6 shows statistics on page faults before and after the garbage collection. Page faults during the garbage collection may slow down the garbage collection severely.

Line 7 shows the nursery size after the young collection.

Line 8 displays the time it took to start all Java threads after the garbage collection, in this case 0.030 ms.

Aborted Compactions

Verbose `memdbg` outputs tell you if compaction is aborted due to too many pointers to the compaction area. When that happens, the following output is printed:

```
[memdbg ] Pointerset limit hit, compaction aborted.
```

Parallel Sweep in Concurrent Old Collections

If the heap becomes full during the mark phase of a concurrent old collection, the garbage collector will by default override the concurrent sweep phase and use parallel sweep instead. When that happens, the following output is printed:

```
[memdbg ] The heap became full during concurrent mark. Running parallel
sweep.
```

Verbose Compaction Log Module

The JRockit JVM performs partial compaction of the heap at each old collection. Compaction reduces fragmentation in the heap, which makes object allocation faster. The verbose `compaction` log module displays details on compaction. The overhead of this log module is low, which means that it can be enabled in production environments.

Listing 19-10 shows an example of a verbose output from the `compaction` log module. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-10 Verbose Compaction Old Collection Output

```
1: [compact] OC 4: 8 parts (max 128), index 8. Type internal, (exceptional false)

2: [compact] Area start: 0x01EC0000, end: 0x03180000

3: [compact] Updated 14130 references in 0 nonmoved and 11700 moved objects

4: [compact] Average compact time ratio: 0.442280

5: [compact] Compaction pause: 2.548 (target 50.000), update ref pause: 10.025
(target 50.000)

6: [compact] Updated 59048 refs (limit: 84386).

7: [compact] Compaction ended at index 15, object end address 0x03179A00.

8: [compact] Summary: 4;8;15;8;1;0;2.548;50.000;10.025;50.000;59048;84386

9: [compact] Adjusting compactsetlimit: 102698

10: [compact] Pause per 1000 refs, current: 0.169920, average: 0.486862. Target
pause: 50.000
```

Line 1 shows a summary of the upcoming compaction. In this example the sequence number of the old collection is 4. 8 heap parts out of a total of 128 heap parts will be compacted, starting at index 8. The compaction type is internal, which means that objects will be moved within the compaction area. The compaction is not exceptional, which means that it can be aborted or interrupted if it takes too long.

Line 2 shows the physical addresses of the start and the end of the compaction area. This information is useful for advanced diagnostics.

Line 3 displays statistics on the number of references updated within the compaction area due to moved objects. In this example 14130 references were updated in 11700 moved objects.

Line 4 informs you that the average ratio between time spent in moving objects and updating references to moved objects is 0.44.

Line 5 shows detailed information on the two components of the pause time caused by compaction, as well as the current pause targets for these components. In this example, moving objects took only 2.548 ms and updating references to moved objects took 10.025 ms, while the target was 50 ms for each of the components.

Line 6 shows the total number of references updated, both those within the compaction area and those outside the compaction area that pointed at objects within the compaction area. This information is useful for monitoring and tuning.

Line 7 shows the end index and physical address of the compaction. This information is useful for diagnostics.

Line 8 contains a machine readable summary of some of the statistics from this compaction. This line is useful for collecting statistics.

Line 9 informs you that the compaction heuristics have adjusted the compact set limit. The compact set limit determines the amount of compaction that can be done at each old collection.

Line 10 shows statistics on the average time for updating 1000 references. This value is used as a base for the compact set limit calculation.

Aborted Compaction Verbose Output

Whenever a compaction is aborted, the verbose compaction log module will display information on the reason for the aborted compaction. In the following example the pointer matrix for a garbage collection thread reached its top limit, which means that there were too many pointers to objects within the selected compaction area.

```
[compact] Pointermatrix for thread '(GC Worker Thread 1)' failed to extend
beyond 25817 elements.
```

Verbose Gcpause Log Module

The `-Xverbose:gcpause` log module displays information on individual garbage collection pauses. For monitoring, tuning and diagnosing latencies this information is essential. The overhead of the log module is low, which means that it can be used in production environments.

Verbose Gcpause Parallel Old Collection Output

A parallel old collection pauses all Java threads during the entire garbage collection. The output from `-Xverbose:gcpause` during a parallel old collection is thus fairly simple, as seen in Listing 19-11. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-11 Verbose Gcpause Parallel Old Collection Output

```
1: [gcpause] Threads waited for memory 147.424 ms starting at 6.398 s
2: [gcpause] old collection phase 1-0 pause time: 146.861682 ms, (start time:
6.398 s)
3: [gcpause] (pause includes compaction: 10.169 ms (external), update ref: 14.390
ms)
```

Line 1 tells you that Java threads waited 147.424 ms for memory due to the old collection. This value can be different than the old collection pause time.

Line 2 shows the pause time for "phase 1-0" of the old collection. Phase 1 is the first phase, while phase 0 is the default phase while the garbage collection isn't running. This means that the timing started in phase 1 and ended after the garbage collector was finished, and thus includes the entire garbage collection.

Line 3 shows some details on how much of the pause time consisted of compaction and reference updates due to compaction.

Verbose Gcpause Concurrent Old Collection Output

A mostly concurrent (or "concurrent") old collection consists of several concurrent garbage collection phases with short pauses in between.

Listing 19-12 shows an example of an output from a mostly concurrent old collection. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-12 Verbose Gcpause Concurrent Old Collection Output

```
1: [gcpause] old collection phase 1 pause time: 19.184561 ms, (start time: 24.027
s)
```

```
2: [gcpause] old collection phase 4-5 pause time: 25.100956 ms, (start time:
24.720 s)

3: [gcpause] (pause includes yc: 24.456 ms, compaction: 0.230 ms (external),
update ref: 0.003 ms)

4: [gcpause] old collection phase 5 pause time: 0.253384 ms, (start time: 24.769
s)

5: [gcpause] old collection phase 5-0 pause time: 1.751061 ms, (start time: 1.391
s)
```

Line 1 shows the pause time for phase 1 of the old collection. Phase 1 is the initial marking phase.

Line 2 shows the pause time for the pause that starts with phase 4 and ends at the beginning of phase 5 of the old collection. This phase is the final marking phase and compaction.

Line 3 shows some details of the phase 4-5 pause. The young collection performed in phase 4 took 24.456 ms, the compaction took 0.23 ms and reference updates after compaction took 0.003 ms. This information is useful for tuning and diagnosing compaction pause times.

Line 4 shows the pause time for the pause in the middle of phase 5. Phase 5 is the concurrent sweep phase. The heap is swept in two parts, and the short pause is for switching parts to sweep.

Line 5 shows the pause time for the pause at the end of phase 5, where final statistics are collected and the garbage collection is wrapped up.

Verbose Gcpause Young Collection Output

A young collection consists of a single pause. Thus the verbose `gcpause` output for a young collection is very simple, as seen in this example:

```
[gcpause] nursery collection pause time: 37.832462 ms
```

Verbose Gcreport Log Module

The `-Xverbose:gcreport` log module prints out a summary of garbage collection activity at the end of the run.

Listing 19-13 shows an example of a verbose gcreport output. This example is from the JRockit JVM R27.4. Line numbers have been added.

Listing 19-13 Verbose GCreport Output

```
1: [memory ] Memory usage report
2: [memory ]
3: [memory ] young collections
4: [memory ]     number of collections = 5647
5: [memory ]     total promoted =        58920467 (size 2144906056)
6: [memory ]     max promoted =          219349 (size 10249784)
7: [memory ]     total GC time =         37.543 s
8: [memory ]     mean GC time =          6.648 ms
9: [memory ]     maximum GC Pauses =      59.602 , 71.426, 75.759 ms
10: [memory ]
11: [memory ] old collections
12: [memory ]     number of collections = 34
13: [memory ]     total promoted =        776698 (size 28208080)
14: [memory ]     max promoted =          72997 (size 2655216)
15: [memory ]     total GC time =         13.970 s (pause 4.583 s)
16: [memory ]     mean GC time =          410.872 ms (pause 134.790 ms)
17: [memory ]     maximum GC Pauses =      147.064 , 172.153, 209.094 ms
18: [memory ]
19: [memory ]     number of concurrent mark phases  = 21
20: [memory ]     number of parallel mark phases  = 13
21: [memory ]     number of concurrent sweep phases  = 18
22: [memory ]     number of parallel sweep phases  = 16
```

Lines 3-9 display information on the young collections during this run.

Line 4 shows the total number of young collections.

Line 5 shows the total number of objects promoted and their total size in bytes.

Line 6 shows the largest number of objects promoted during a single young collection and their total size in bytes.

Line 7 shows the total time spent in young collections.

Line 8 shows the average time spent in a single young collection.

Line 9 shows the three longest pause times caused by young collection.

Lines 11-17 show statistics on old collections.

Line 12 shows the total number of old collections.

Line 13 shows the number of objects promoted during old collections and their total size in bytes.

Line 14 shows the largest number of objects promoted during a single old collection and their total size in bytes.

Line 15 shows the total time spent in old collections and the sum of all garbage collection pauses caused by old collections.

Line 16 shows the average time spent in a single old collection and the average sum of pauses during a single old collection.

Line 17 shows the three longest old collection pauses.

Line 19 displays the number of concurrent mark phases. In this example 21 of the old collections used concurrent mark.

Line 20 shows the number of parallel mark phases. In this example 13 of the old collections used parallel mark.

Line 21 shows the number of concurrent sweep phases. In this example 18 of the old collections used concurrent sweep.

Line 22 shows the number of parallel sweep phases. In this example 16 of the old collections used parallel sweep.

Verbose Refobj and Referents Log Modules

The `-Xverbose:referents` log module was introduced in the JRockit JVM R27.2, while `-Xverbose:refobj` was introduced in the JRockit JVM R27.5 and will replace the verbose `referents` module.

The verbose `referents` module introduces some overhead and is not suitable for production environments. The verbose `refobj info` level output in R27.2 is much cheaper and can be used

in production environments. The debug level output of `refobjs` corresponds to the old verbose `referents` information on `info` level and should not be used in production.

Verbose Refobj Output on Info Level

The `-Xverbose:refobj` log module shows a summary of the number of reference objects and how they are handled at each garbage collection.

Listing 19-14 shows an example of a verbose `refobj` output. This example is from the JRockit JVM R27.5. Line numbers have been added.

Listing 19-14 Verbose Refobj Output on Info Level

```
1: [refobj ] SoftRef: Reach:   2 Act: 0 PrevAct: 11 Null: 50
2: [refobj ] WeakRef: Reach: 183 Act: 0 PrevAct:  0 Null: 10
3: [refobj ] Phantom: Reach:   0 Act: 0 PrevAct:  0 Null:  0
4: [refobj ] ObjMoni: Reach:   2 Act: 0 PrevAct:  0 Null:  0
5: [refobj ] Finaliz: Reach:  17 Act: 0 PrevAct:  0 Null:  0
6: [refobj ] WeakHnd: Reach: 306 Act: 0 PrevAct:  0 Null:  0
7: [refobj ] SoftRef: @Mark:  62 @Preclean:   1 @FinalMark:   0
8: [refobj ] WeakRef: @Mark: 178 @Preclean:  15 @FinalMark:   0
9: [refobj ] Phantom: @Mark:   0 @Preclean:   0 @FinalMark:   0
10: [refobj ] ObjMoni: @Mark:   2 @Preclean:   0 @FinalMark:   0
11: [refobj ] Finaliz: @Mark:   0 @Preclean:  17 @FinalMark:   0
12: [refobj ] WeakHnd: @Mark:   0 @Preclean: 105 @FinalMark: 201
13: [refobj ] SoftRef: SoftAliveOnly: 0 SoftAliveAndReach: 0
```

Lines 1-6 show the number of occurrences of each reference object type, finalizers, weak handles and object monitors. The references objects are categorized by status, as follows:

- **Reachable:** Reference objects with reachable referents. A referent that is reachable on a harder level is considered reachable; for example a referent of a soft reference is reachable if it also is hard reachable.

- **Activated:** Activated references are such that the referent is no longer reachable on any harder level than this reference, which means that the reference can be cleared or put on a reference queue.
- **Previously Activated:** References that have been activated at a previous garbage collection but have not yet been cleared are previously activated.
- **Null:** Null references are such that the reference in the reference object is null, but the reference object itself still exists.

Lines 7-12 show statistics on in which garbage collection phases the reference objects were handled.

Line 13 shows how many soft references are soft alive only and how many are also hard reachable.

Verbose Referents Output and Verbose Refobj on Debug Level

The old verbose `referents` log module and the `debug` level of the `refobj` log module displays detailed information on reference objects and referents, as seen in Listing 19-15. This example is from the JRockit JVM R27.2. Note that the output may look different in later releases.

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).

Additional information is that the header/footer of the report informs what type of collection took place. In the header you can see the time since the last old collection and the amount of free at that time. If any soft references are present you will also find information on which softly reachable referents are collected based on when they where last looked up through `get()`.

Note: These verbose outputs have a significant negative impact on the performance.

Listing 19-15 Verbose Referents Output

```
--- Verbose reference objects statistics - heap collection -------
63.7 MB free memory (of 64.0 MB) after last heap GC, finished 0.177 s ago.
Soft references: 0 (0 only soft reachable, 2 cleared this GC)
    java/lang/ref/SoftReference: 0 (0, 2)
         0 (0, 1) java/lang/StringCoding$CharsetSD
         0 (0, 1) [Ljava/lang/String;
    Softly reachable referents not used for at least 0.000 s cleared.
Weak references: 10 (0 cleared this GC)
    java/lang/ref/WeakReference: 9 (0)
```

```
         9 (0)    java/lang/Class
    java/lang/ThreadLocal$ThreadLocalMap$Entry: 1 (0)
         1 (0)    java/lang/ThreadLocal
Weak object handles: 63 (0 cleared this GC)
        35 (0)    sun/misc/Launcher$AppClassLoader
        28 (0)    java/lang/String
Final handles: 10 (0 pending finalization, 0 became pending this GC)
         4 (0, 0) java/util/jar/JarFile
         3 (0, 0) java/util/zip/Inflater
         2 (0, 0) java/io/FileOutputStream
         1 (0, 0) java/io/FileInputStream
Phantom references: 2 (0 only phantom reachable, 0 became phantom reachable this
   GC)
    jrockit/vm/ObjectMonitor: 2 (0, 0)
         1 (0, 0) java/lang/Object
         1 (0, 0) java/io/PrintStream
--- End of reference objects statistics - heap collection --------
```

In this example the garbage collection has cleared two soft references referring to objects or arrays of the types java.lang.StringCoding and java.lang.String, respectively.

There are 10 weak references, 63 weak object handles, 10 final handles and 2 phantom references.

Other Verbose Log Modules

Among the various log modules available in the JRockit JVM, `opt` and `exceptions` are two of the most used and most useful.

Verbose Opt Log Module

The `-Xverbose:opt` log module displays information on code optimizations done by the optimizing compiler.

Listing 19-16 shows an example of a verbose opt output. This example is from the JRockit JVM R27.5. Line numbers have been added.

Listing 19-16 Verbose Opt Output

```
1: [opt    ] #1 5 (0x1c) o0 java/util/Random.acquireSeedLock()V
2: [opt    ] #1 5 (0x1c) o0 @0x13AA0000-0x13AA003F  2.43 ms (2.43 ms)
```

```
3: [opt    ] #2 5 (0x1c) o0 java/util/Random.next(I)I
4: [opt    ] #2 5 (0x1c) o0 @0x13AA0370-0x13AA03FF  2.66 ms (20.19 ms)
```

Lines 1 and 3 show the names of two methods that are optimized.

Lines 2 and 4 show the addresses of the methods and the time it took to optimize them.

You can use the verbose `opt` information to diagnose and monitor the optimizations.

Verbose Exceptions Log Module

The `-Xverbose:exceptions` log module prints each Java exception that is thrown in the application. You can use this information to monitor and troubleshoot exceptions in your application.

Listing 19-17 shows some example outputs from the verbose exceptions log module. This example is from the Oracle JRockit JVM R27.5. Each line displays the name of the exception thrown as well as the exception message, if such is available.

Listing 19-17 Verbose Exceptions Output

```
[excepti][00004] java/lang/NullPointerException
[excepti][00004] java/lang/NullPointerException: null array passed into
arraycopy
[excepti][00004] java/lang/ArrayIndexOutOfBoundsException
[excepti][00004] java/lang/ArrayIndexOutOfBoundsException
[excepti][00004] java/lang/NullPointerException: null array passed into
arraycopy
```

`-Xverbose:exceptions=debug` prints out the same information but also provides stack traces for each exception.

CHAPTER 20

Using Thread Dumps

This chapter describes how to get and use Oracle JRockit JVM thread dumps. For basic background information about threads and thread synchronization, see Understanding Threads and Locks.

A thread dump is a snapshot of the state of all threads that are part of the process. The state of each thread is presented with a so called stack trace, which shows the contents of a thread's stack. Some of the threads belong to the Java application you are running, while others are JVM internal threads.

A thread dump reveals information about an application's thread activity that can help you diagnose problems and better optimize application and JVM performance; for example, thread dumps automatically show the occurrence of a deadlock. Deadlocks bring some or all of an application to a complete halt.

The following subjects are discussed in this chapter:

- Creating Thread Dumps
- Reading Thread Dumps
- Thread Status in Thread Dumps
- Troubleshooting with Thread Dumps

Creating Thread Dumps

To create a thread dump from a process, do either of the following:

- Press `Ctrl-Break` while the process is running (or by sending `SIGQUIT` to the process on Linux).
- Enter the following at the command line at startup:
  `bin\jrcmd.exe <pid> print_threads`

The thread dump appears at the command line.

Note: For more information about jrcmd and Ctrl-Break handlers, see Running Diagnostic Commands.

Reading Thread Dumps

This section describes the typical contents of a thread dump by going through an example thread dump from the beginning to end. First, an example thread dump, broken up into its components is presented (see Listing 20-1, Listing 20-2, Listing 20-3, Listing 20-4 and Listing 20-5). First, information about the main thread is printed, then all the JVM internal threads, followed by all other Java application threads (if there are any). Finally, information about lock chains are printed.

The example thread dump is taken from a program that creates three threads that are quickly forced into a deadlock. The application threads Thread-0, Thread-1, and Thread-2 correspond to three different classes in the Java code.

The Beginning of The Thread Dump

The thread dump starts with the date and time of the dump, and the version number of the JRockit JVM used (see Listing 20-1).

Listing 20-1 The initial information of a thread dump

```
===== FULL THREAD DUMP ===============
Wed Feb 21 13:46:45 2007
BEA JRockit(R) R27.1.0-109-73164-1.5.0_08-20061129-1428-windows-ia32
```

Stack Trace for Main Application Thread

Listing 20-2 shows the stack trace of the main application thread. There is a thread information line, followed by information about locks and a trace of the thread's stack at the moment of the thread dump.

Listing 20-2 The main thread in the thread dump

```
"Main Thread" id=1 idx=0x2 tid=48652 prio=5 alive, in native, waiting
-- Waiting for notification on: util/repro/Thread1@0x01226528[fat lock]
at jrockit/vm/Threads.waitForSignal(J)Z(Native Method)
at java/lang/Object.wait(J)V(Native Method)
at java/lang/Thread.join(Thread.java:1095)
^-- Lock released while waiting: util/repro/Thread1@0x01226528[fat lock]
at java/lang/Thread.join(Thread.java:1148)
at util/repro/DeadLockExample.main(DeadLockExample.java:23)
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace
```

After the name and other identification information, the different status messages of the main thread are printed. The main thread in Listing 20-2 is a running thread (`alive`), it is either executing JVM internal code or user-defined JNI code (`in native`), and it is currently waiting for an object to be released (`waiting`). If a thread is waiting on a notification on a lock (by calling `Object.wait()`), this is indicated at the top of the stack trace as `Waiting for notification on`.

Locks and Lock Chains

For each thread, the JRockit JVM prints the following information:

- If the thread is trying to take a lock (to enter a synchronized block), but the lock is already held by another thread, this is indicated at the top of the stack trace, as "Blocked trying to get lock".
- If the thread is waiting on a notification on a lock (by calling `Object.wait()`), this is indicated at the top of the stack trace as "Waiting for notification".

- If the thread has taken any locks, this is shown in the stack trace. After a line in the stack trace describing a function call is a list of the locks taken by the thread in that function. This is described as `^-- Holding lock` (where the `^--` serves as a reminder that the lock is taken in the function written above the line with the lock).

The semantics for waiting (for notification) on an object in Java is somewhat complex. First, to enter a synchronized block, you must take the lock for the object, and then you call `wait()` on that object. In the wait method, the lock is released before the thread actually goes to sleep waiting for a notification. When it receives a notification, wait re-takes the lock before returning. So, if a thread has taken a lock, and is waiting (for notification) on that lock, the line in the stack trace that describes when the lock was taken is not shown as "Holding lock," but as "Lock released while waiting."

All locks are described as `Classname@0xLockID[LockType]`; for example:

```
java/lang/Object@0x105BDCC0[thin lock]
```

`Classname@0xLockID` describe the object to which the lock belongs. The classname is an exact description, the fully qualified classname of the object. `LockID`, on the other hand, is a temporary ID which is only valid for a single thread dump. That is, you can trust that if a thread A holds a lock `java/lang/Object@0x105BDCC0`, and a thread B is waiting for a lock `java/lang/Object@0x105BDCC0`, in a single thread dump, then it is the same lock. If you do any subsequent thread dumps however, `LockID` is not comparable and, even if a thread holds the same lock, it might have a different `LockID` and, conversely, the same `LockID` does not guarantee that it holds the same lock. `LockType` describes the JVM internal type of the lock (fat, thin, recursive, or lazy). The status of active locks (monitors) is also shown in stack traces.

Presentation of Locks Out of Order

The lines with the lock information might not always be correct, due to compiler optimizations. This means two things:

- If a thread, in the same function, takes lock A first and then lock B, the order in which they are printed is unspecified.
- If a thread, in method `foo()` calls method `bar()`, and takes a lock A in `bar()`, the lock might be printed as being taken in `foo()`.

Normally, this should not be a problem. The order of the lock lines should never move much from their correct position. Also, lock lines will never be missing—you can be assured that all locks taken by a thread are shown in the thread dump.

JVM Internal Threads

Listing 20-3 shows the traces of JVM internal threads. The threads have been marked as daemon threads, as can be seen by their `daemon` state indicators. Daemon threads are either JVM internal threads (as in this case) or threads marked as daemon threads by `java.lang.Thread.setDaemon()`.

Listing 20-3 The first and last thread in a list of JVM internal Threads

```
"(Signal Handler)" id=2 idx=0x4 tid=48668 prio=5 alive, in native, daemon
[...]
"(Sensor Event Thread)" id=10 idx=0x1c tid=48404 prio=5 alive, in native,
daemon
```

As you can see, lock information and stack traces are not printed for the JVM internal threads in Listing 20-3. This is the default setting.

If you want to see stack traces for the JVM internal threads, then use the parameter `nativestack=true` when you send the `print_threads` handler. At the command line, write the following:

```
bin\jrcmd.exe <pid> print_threads nativestack=true
```

Other Java Application Threads

Normally, you will primarily be interested in the threads of the Java application you are running (including the main thread). All Java application threads except the main thread are presented near the end of the thread dump. Listing 20-4 shows the stack traces of three different application threads.

Listing 20-4 Additional application threads

```
"Thread-0" id=11 idx=0x1e tid=48408 prio=5 alive, in native, blocked
-- Blocked trying to get lock: java/lang/Object@0x01226300[fat lock]
at jrockit/vm/Threads.waitForSignal(J)Z(Native Method)
at
```

```
jrockit/vm/Locks.fatLockBlockOrSpin(ILjrockit/vm/ObjectMonitor;II)V(Unknow
n Source)
at
jrockit/vm/Locks.lockFat(Ljava/lang/Object;ILjrockit/vm/ObjectMonitor;Z)Lj
ava/lang/Object;(Unknown Source)
at
jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Ob
ject;(Unknown Source)
at
jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknow
n Source)
at util/repro/Thread1.run(DeadLockExample.java:34)
^-- Holding lock: java/lang/Object@0x012262F0[thin lock]
^-- Holding lock: java/lang/Object@0x012262F8[thin lock]
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace


"Thread-1" id=12 idx=0x20 tid=48412 prio=5 alive, in native, blocked
-- Blocked trying to get lock: java/lang/Object@0x012262F8[thin lock]
at jrockit/vm/Threads.sleep(I)V(Native Method)
at jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I(Unknown
Source)
at
jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Ob
ject;(Unknown Source)
at
jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknow
n Source)
at util/repro/Thread2.run(DeadLockExample.java:48)
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace


"Thread-2" id=13 idx=0x22 tid=48416 prio=5 alive, in native, blocked
-- Blocked trying to get lock: java/lang/Object@0x012262F8[thin lock]
at jrockit/vm/Threads.sleep(I)V(Native Method)
at jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I(Unknown
Source)
```

```
at
jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Ob
ject;(Unknown Source)
at
jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknow
n Source)
at util/repro/Thread3.run(DeadLockExample.java:65)
^-- Holding lock: java/lang/Object@0x01226300[fat lock]
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace
```

All three threads are in a blocked state (indicated by `blocked`), which means that they are all trying to enter synchronized blocks. Thread-0 is trying to take `Object@0x01226300[fat lock]` but this is held by Thread-2. Both Thread-2 and Thread-1 are trying to take `Object@0x012262F8[thin lock]` but this lock is held by Thread-0. This means that Thread-0 and Thread-2 form a deadlock, while Thread-1 is blocked.

Lock Chains

One prominent feature of the JRockit JVM is that it automatically detects deadlocked, blocked and open lock chains among the running threads. The analysis in Listing 20-5 presents the all the lock chains created by the threads T1, T2, T3, T4 and T5. This information can be used to tune and troubleshoot your Java code.

Listing 20-5 Deadlocked and blocked lock chains

```
Circular (deadlocked) lock chains
=================================
Chain 6:
"Dead T1" id=16 idx=0x48 tid=3648 waiting for java/lang/Object@0x01225018
held by:
"Dead T3" id=18 idx=0x50 tid=900 waiting for java/lang/Object@0x01225010
held by:
"Dead T2" id=17 idx=0x4c tid=3272 waiting for java/lang/Object@0x01225008
held by:
"Dead T1" id=16 idx=0x48 tid=3648
```

```
Blocked lock chains
===================
Chain 7:
"Blocked T2" id=20 idx=0x58 tid=3612 waiting for
java/lang/Object@0x01225310 held by:
"Blocked T1" id=19 idx=0x54 tid=2500 waiting for
java/lang/Object@0x01224B60 held by:
"Open T3" id=13 idx=0x3c tid=1124 in chain 1

Open lock chains
================
Chain 1:
"Open T5" id=15 idx=0x44 tid=4048 waiting for java/lang/Object@0x01224B68
held by:
"Open T4" id=14 idx=0x40 tid=3380 waiting for java/lang/Object@0x01224B60
held by:
"Open T3" id=13 idx=0x3c tid=1124 waiting for java/lang/Object@0x01224B58
held by:
"Open T2" id=12 idx=0x38 tid=3564 waiting for java/lang/Object@0x01224B50
held by:
"Open T1" id=11 idx=0x34 tid=2876 (active)
```

Thread Status in Thread Dumps

This section describes the different statuses or states a thread can show in a thread dump. There are three types of states:

- Life States
- Run States
- Special States

Life States

Table 20-1 describes the life states a thread can show in a thread dump.

Table 20-1 Thread Life States

State	Description
`alive`	This is a normal, running thread. Virtually all threads in the thread dump will be alive.
`not started`	The thread has been requested to start running by `java.lang.Thread.start()`, but the actual OS process has not yet started, or executed far enough to pass control to the JRockit JVM. It is extremely unlikely to see this value. A `java.lang.Thread` object that is created, but not has had `start()` executed, will not show up in the thread dump.
`terminated`	This thread has finished its `run()` method and has also notified any threads joining on it, but it is still kept in the JVM internal thread structure for running threads. It is extremely unlikely to see this value. A thread that has been terminated for a time longer than a few milliseconds will not show up in the thread dump.

Run States

Table 20-2 describes the run states a thread can show in a thread dump.

Table 20-2 Thread Run States

State	Description
`blocked`	This thread has tried to enter a synchronized block, but the lock was taken by another thread. This thread is blocked until the lock gets released.
`blocked (on thin lock)`	This is the same state as `blocked`, but with the additional information that the lock in question is a thin lock.
`waiting`	This thread has called `Object.wait()` on an object. The thread will remain there until some other thread sends a notification on that object.
`sleeping`	This thread has called `java.lang.Thread.sleep()`.
`parked`	This thread has called `java.util.concurrent.locks.LockSupport.park()`.
`suspended`	The thread's execution has been suspended by `java.lang.Thread.suspend()` or a JVMTI/JVMPI agent call

Special States

Table 20-3 describes the special states a thread can show in a thread dump. Note that all these states are not mutually exclusive.

Table 20-3 Special Thread States

State	Description
`interrupted`	The user has called `java.lang.Thread.interrupt()` on this thread.
`daemon`	This is either an JVM internal thread or a thread that has been marked as a daemon thread by `java.lang.Thread.setDaemon()`.
`in native`	This thread is executing native code. This could either mean user-supplied JNI code, or JVM internal code.
`in suspend critical mode`	This thread is executing JVM internal code, and has marked itself as being `suspend critical`, meaning that for a short moment, it will block a garbage collection from taking place.
`native_blocked`	This thread is executing JVM internal code, and have tried to take an JVM internal lock. The thread is blocked, since that lock is held by another thread.
`native_waiting`	This thread is executing JVM internal code, and is waiting for notification from another thread on an JVM internal lock.

Troubleshooting with Thread Dumps

This section contains information on about how to use thread dumps for troubleshooting and diagnostics.

To use thread dumps for troubleshooting, beyond detecting deadlocks, you need to take several thread dumps from the same process. However, if you want to do long time analysis of behavior you will likely be more helped by combining occasional thread dumps with other diagnostics tools, such as the JRockit Runtime Analyzer, which is part of Oracle JRockit Mission Control (see Using Oracle JRockit Mission Control Tools for more information).

Detecting Deadlocks

The Oracle JRockit JVM automatically analyzes the thread dump information and detects whether there exists any circular (deadlocked) or blocked lock chains in it.

Detecting Processing Bottlenecks

For detecting more than deadlocks in your threads, you have to make several consecutive thread dumps. This lets you detect the occurrence of contention, where multiple threads are trying to get the same lock. Contention might create long open lock chains that, while not deadlocked, will degrade performance.

If you discover (in a set of consecutive thread dumps) that one or more threads in your application is temporarily stuck waiting for a lock to be released, then you might have reason to look over the code of your Java application to see if the synchronization (serialization) is necessary or if the threads can be organized differently.

Viewing The Runtime Profile of an Application

By making several consecutive thread dumps, you might quickly get an overview of which parts of your Java application that are most heavily used. However, you should consult the **Threads** tab in JRockit Management Console for more detailed information about the workload on the different parts of your application.

CHAPTER 21

Running Diagnostic Commands

Use diagnostic commands to communicate with a running Oracle JRockit JVM process. These commands tell the JRockit JVM to for example print a heap report or a garbage collection activity report, or to turn on or off a specific verbose module. This chapter describes how to run diagnostic commands and lists the available commands. The following sections are included:

Diagnostic Commands Overview

Diagnostic commands help you communicate with a running JRockit JVM process. With these commands you can for example ask for a heap report or enable or disable a verbose module.

You can send diagnostic commands to a running JVM process in several ways:

- By using `jrcmd`, a command line tool that sends the commands to a given JRockit JVM process.
- By pressing Ctrl-Break, whereupon the JVM will look for a `ctrlhandler.act` file and execute the commands listed therein.

- By using the JRockit Management Console in Oracle JRockit Mission Control to send diagnostic commands to a running JRockit JVM process.

You can enable or disable any diagnostic command using the system property `-Djrockit.ctrlbreak.enable<name>=<true|false>`, where `name` is the name of the diagnostic command. The following two handlers are disallowed by default and need to be turned on:

- `run_class`
- `force_crash`.

For example:

```
-Djrockit.ctrlbreak.enablerun_class=true
```

Using jrcmd

`jrcmd` is a command line tool included with the JRockit JDK you can use to send diagnostic commands to a running JVM process. This section provides a brief overview of `jrcmd`. It includes information on the following subjects:

- How jrcmd Communicates with the JRockit JVM
- How to Use jrcmd
- jrcmd Examples
- Known Limitations of jrcmd

How jrcmd Communicates with the JRockit JVM

`jrcmd` uses the JRIPC library, a small C library, to communicate with a running JRockit JVM process. JRIPC has the following basic functionality

- It discovers which JRockit JVM processes are running on the machine
- It sends diagnostic commands to a JRockit JVM process
- It reads performance counters exposed by the JRockit JVM

How to Use jrcmd

To use `jrcmd`, simply enter it at the command line, with the appropriate parameters:

```
jrcmd <jrockit pid> [<command> [<arguments>]] [-l] [-f file] [-p] -h]
```

where:

- [<*command*> [<*arguments*>]] is any diagnostic command and its associated arguments; for example, `version`, `print_properties`, `command_line`, and so on.
- `-l` displays the counters exposed by this process
- `-f` reads and execute commands from the file
- `-p` lists JRockit JVM processes on the local machine
- `-h` shows help

If the PID is 0, commands will be sent to all JRockit JVM processes. If no options are given, default is `-p`.

jrcmd Examples

Here are some examples of using jrcmd for:

- Listing JRockit JVM Processes
- Sending a Command to a Process
- Sending Several Commands

Listing JRockit JVM Processes

Do the following to list all JRockit JVM processes running on the machine:

1. Run `jrcmd` or `jrcmd -p` to list the running JRockit JVMs; for example:

   ```
   > jrcmd -P
   10064 Sleeper
           -Xverbose:memory -Xmx30m
   >
   ```

   You will see the PID of the process (`10064`) and the program it is currently running (`Sleeper`) as well as the parameters used to start the JVM (`-Xverbose:memory -Xmx30m`).

Sending a Command to a Process

To send a command to the process you identified in Listing JRockit JVM Processes, do the following:

1. Find the PID from Listing JRockit JVM Processes (`10064`)

2. Enter `jrcmd` with that PID and the `version` command; for example:

   ```
   > jrcmd 10064 version
   ```

   This command sends the `version` command to the JRockit JVM. The response will be:

   ```
   Oracle WebLogic JRockit(R) Virtual Machine build 9.9.9-1.5.0-Jun 9
   2004-13:52:53-<internal>, Native Threads, GC strategy: parallel
   ```

Sending Several Commands

You can create a file (just like the `ctrlhandler.act` file) with several commands and execute all of them. Use this procedure:

1. Create a file called commands.txt with the following contents:
   - `version`
   - `timestamp`
2. Execute the file with `jrcmd`; for example:

   ```
   > jrcmd 10064 -f commands.txt
   ```

   The system will respond:

```
Oracle WebLogic JRockit(R) Virtual Machine build 9.9.9-1.5.0-Jun  9
2004-13:52:53-<internal>, Native Threads, GC strategy: parallel

==== Timestamp ====  uptime: 0 days, 00:05:04 time: Fri Jun 11 14:28:31 2004
```

3. Use the PID `0` to send the commands to all running JRockit JVM processes.

Known Limitations of jrcmd

When using jrcmd, be aware of these limitations:

- In order to issue diagnostic commands to a process on Linux or Solaris, you need to run jrcmd as the same user as the one running the Java process.
- When using jrcmd on Windows, you need to run the Java process and jrcmd from the same Windows station. If you run the Java process as a Windows service, and run jrcmd on your desktop, it will not work, since they are running in two separate environments.
- When an JRockit JVM is started as root and then changed to a less privileged user, jrcmd will not be able to communicate properly with the process thereafter due to security restrictions.

  - The following things can be done:
    Root can list the running processes.
    The less privileged user can send commands to the process.
  - The following things cannot be done:
    Root cannot send commands to the process; any commands will be treated as a Ctrl-Break signal and print a thread dump instead.
    The less privileged user cannot list the running JRockit JVM process, but if they know the process ID (PID), they can send commands to the process using `jrcmd <pid> <command>`.

- If the default Windows temporary directory (java.io.temp) is on a FAT file system, jrcmd will not be able to discover local processes. For security reasons, local monitoring and management is only supported if your default Windows temporary directory is on a file system that supports setting permissions on files and directories (for example, on an NTFS file system). It is not supported on a FAT file system that provides insufficient access controls.

Ctrl-Break Handler

Another way you can run diagnostic commands is by pressing Ctrl-Break. When you press Ctrl-Break, the JRockit JVM will search for a file named `ctrlhandler.act` (see Listing 21-1) in your current working directory. If it doesn't find the file there, it will look in the directory containing the JVM. If it does not find this file there either, it will revert to displaying the normal thread dump. If it finds the file, it will read the file searching for command entries, each of which invoke the corresponding diagnostic command.

Listing 21-1 ctrlhandler.act File

```
set_filename filename=c:\output.txt append=true
print_class_summary
print_object_summary increaseonly=true
print_threads
print_threads nativestack=true
print_utf8pool
jrarecording filename=c:\myjra.xml time=120 nativesamples=true
verbosity set=memory,memdbg,codegen,opt,sampling filename="c:\output"
timestamp

stop
# ctrl-break-handler will stop reading the file after it finds
# the stop key-word
```

```
#
# version - print JRockit version
#
# print_threads - the normal thread dump that lists all the currently
# running threads and there state
#
# print_class_summary - prints a tree of the currently loaded classes
#
# print_utf8pool - print the internal utf8 pool
#
# print_object_summary - display info about how many objects of each
# type that are live currently and how much size
# they use. Also displays points to information
#
# jvmpi_datadump
#
# jvmpi_datareset
#
# jrarecording - starts a jrarecording
#
# verbosity - changes the verbosity level , not functional in ariane142_04
#
# start_management_server - starts a management server
# kill_management_server - shuts the management server down
#   (the managementserver.jar has to be in the bootclasspath for
#   these command to work)
#
#
```

In the `ctrlhandler.act` file, each command entry starts with a Ctrl-Break Handler name followed by the arguments to be passed to the Ctrl-Break Handler. The arguments should be on the property form (that is, name=value; for example, `set_filename filename=c:\output.txt append=true`). String, integer or boolean values are acceptable property types.

You can disable Ctrl-Break functionality by setting this command:

`-Djrockit.dontusectrlbreakfile=true`.

Available Diagnostic Commands

Table 21-1 lists the currently available diagnostic commands.

Table 21-1 Existing Ctrl-Break Handlers

Command	Description
`set_filename filename=<file> [append=true]`	Set the file which all commands following this command will use for printing. You can have several `set_filename` commands in a file. It takes two arguments: filename and an optional append to specify if you want to append to the file or overwrite it. The default file is `stderr`, and to overwrite the file.
`timestamp`	Prints a timestamp.
`version`	Prints the JRockit JVM version
`print_threads [nativestack=true] [jvmmonitors=true]`	Prints a normal thread dump. • `nativestack=true` will print C-level stacktraces as well as Java traces. • `jvmmonitors=true` will also print the JRockit JVM's internal native locks (those that are registered): status and wait queue, and with `-XXnativeLockProfiling=true` their profile stats (acquired/contended/tryfailed).
`verbosity [args=<components>] [filename=<file>]`	Change the verbosity level normally specified with -Xverbose. This handler does not work in R25.
`command_line`	Prints the command line used to start the JRockit JVM.
`print_object_summary`	See the JRockit Memory Leak Detector User Guide (for JRockit Mission Control 1.0) or the Memory Leak Detector built-in help (for Oracle JRockit Mission Control 2.0 and later).
`print_class_summary`	Print all loaded classes.
`print_utf8pool`	Print all UTF8 strings.
`print_memusage`	Print all memory the OS says the JRockit JVM process is holding onto, as well as what each subsystem thinks it is holding onto.

Table 21-1 Existing Ctrl-Break Handlers

Command	Description
`oom_diagnostics`	**Note:** This command applies only to versions of JRockit JVM R26.3 and earlier. Cause an `OutOfMemoryDiagnostics` to be printed. If both `set_filename` and `-Djrockit.oomdiagnostics.filename` is set, the latter takes precedence. This command is deprecated in the JRockit JVM R26.4. Use `heap_diagnostics` instead.
`heap_diagnostics`	Cause a heap diagnostic to be printed. Output ends up on Ctrl-Break Handler output stream and does not take the property `-Djrockit.oomdiagnostics.filename` into consideration. This command applies only to versions of JRockit JVM R26.4 and later.
`heapreport`	Prints out a report on the JVM's native memory allocation on the C-Heap. This is only supported if you are running with `HEAP_TRACE` defined.
`gcreport`	Prints out a comprehensive summary of garbage collection activity so far during the run. In order to be able to dynamically print out the same information as `-XgcReport` would provide at the end of an application run, make sure to have the option flag `-XgcReport` in your start-up configuration, otherwise the correct measurements won't be performed.
`jrarecording [filename=<file>] [time=<time>] [nativesamples=true]`	Starts a JRA recording. For more information, please refer to Creating a JRA Recording with JRockit Mission Control 1.0
`run_optfile [filename=<file>]`	See Creating and Using an Optfile.
`start_management_server`	Starts the management server. (Actually the listening socket that in turn starts servers whenever a connection is established). `managementnserver.jar` has to be in the boot classpath for this command to work.

Table 21-1 Existing Ctrl-Break Handlers

Command	Description
`kill_management_server`	Stops the management server. (Actually shuts down the listening socket.) The only reason it isn't named `stop_management_server` is that stop is a reserved keyword that stops parsing of the act file. The managementserver.jar has to be in the boot classpath for this command to work.
`lockprofile_print`	Will print the current values of the lock profile counters. Enable lock profiling with `-Djrockit.lockprofiling`.
`lockprofile_reset`	Will reset the current values of the lock profile counters. Enable lock profiling with `-Djrockit.lockprofiling`.
`print_exceptions [stacktraces=all/true/false] [exceptions=all/true/false]`	Enable/disable printing of exceptions (see `-Xverbose`). To turn exception printing off completely you need to set `exceptions = false` even if it was turned on by `stacktraces = true`.
`force_crash`	Forces the Oracle JRockit JVM to crash/dump.
`run_class [class=<classname>] [daemon=<true|false>]`	Runs any class implementing the Runnable interface. Must be enabled with: `-Djrockit.ctrlbreak.enablerun_class=true`. Note that the class name must use slashes (`/`) to separate package names; for example: `jrcmd <pid> run_class class=java/lang/Thread`
`memprof [sampleRate=<seconds>] [trendSize=<size>] [forceThreshold=<bytes>] [verboseResultStats=<true|false>] [skipSymbols=<symbolexcludelist>]`	Turns on memory profiling in the running application. Memory profiling can be very helpful for diagnosing such problems as memory leaks.

Getting Help

To get help about the available commands, execute the special command `help`. This will print all available commands.

- `help <handlername>` prints help for the specified command.
- `help help` will print help for `help`.
- `help all` will print the help for all commands.

CHAPTER 22

Oracle JRockit Time Zone Updater

The Oracle JRockit Time Zone Updater (TZUpdater) allows you to update installed JDK/JRE images with more recent time zone data to accommodate the U.S. 2007 daylight saving time changes (US2007DST) originating with the U.S. Energy Policy Act of 2005.

Oracle recommends using the latest Oracle JRockit JDK release as the preferred vehicle for delivering both time zone data updates and other product improvements, such as security fixes. If you are unable to use the latest JRockit JDK/JRE update release, this tool provides a route of updating time zone data while leaving other system configuration and dependencies unchanged.

This section contains information on the following subjects:

Downloading the TZUpdater

Download the TZUpdater from:

http://commerce.bea.com/products/weblogicjrockit/tzupdater/accept_terms_tzupdater.jsp

Introduction to the TZUpdater

To upgrade a specific Java installation, you need to include the full path to the Java executable of that installation. If `tzupdater.jar` is run by just running `java -jar...`, or by double-clicking the `tzupdater.jar` file, this will invoke Sun's Java on many systems, which will result in an error message being displayed. The section Example of the Default way of Using TZUpdater explains the typical use of TZUpdater.

A single JDK/JRE image is modified per execution. For administering of multiple JDK/JRE instances, see System-wide Usage.

Prior to running the TZUpdater, you need to stop any running instances of the specific JDK/JRE that you will operate upon.

The TZUpdater modifies and updates the JVM it is run with, thus it is important to run the tool as a command-line application, see Command-line Options Described.

System Requirements to Run the TZUpdater

The TZUpdater supports Oracle's JDK/JRE releases 1.4 or later on all supported platforms.

Using the TZUpdater

The command-line interface is the following:

```
JAVA_HOME/bin/java -jar tzupdater.jar options
```

Command-line Options Described

If no command-line option is specified, the usage message is displayed. To perform time zone data update, either the `-u` or `-f` option must be specified, see Table 22-1 for a list of all available command line options.

Table 22-1 List of available options.

Command	Option Name	Description
`-h`	help	Prints the usage to `stdout` and exit. Other options are ignored if specified.
`-V`	version	Shows the tool version number and the `tzdata` version numbers of the JRE and the archive embedded in the jar file and exit.
`-u`	update	Updates the time zone data. If this option is specified with `-h`, `-t`, or `-V` option, the command displays the usage to `stdout` and exit.
`-f`	force	Force update the tzdata even if the version of the tzdata archive is older then the JRE's tzdata version. This option doesn't require the `-u` option to perform the update.
`-v`	verbose	Displays detailed messages to `stdout`.
`-bc`	backward compatible	Keeps backward compatibility with the 3-letter time zone IDs of JDK 1.1. Any time zone IDs that conflict with the JDK 1.1 time zone IDs will be removed from the installed time zone data. See Known Issues for details. This option must be specified with the `-u`, `-f`, or `-t` option.
`-t`	test	Runs verification tests only and exit. The -f option is ignored if specified. If the -bc option is specified, any test cases for time zone IDs that conflict with the JDK 1.1 time zone IDs will be ignored.

Example of the Default way of Using TZUpdater

Below is an example of the default way of using the TZUpdater to upgrade time zone data on a Java installation at, for example, `/opt/bea/jrockit90_150_06`.

1. Test the current version of the timezone data of the JRE:

   ```
   > /opt/bea/jrockit90_150_06/bin/java -jar tzupdater.jar -V
   ```

   ```
   tzupdater version: 1.0.0-b03
   JRE time zone data version: tzdata2005n
   Embedded time zone data version: tzdata2007a
   ```

2. Update the timezone data:

   ```
   > /opt/bea/jrockit90_150_06/bin/java -jar tzupdater.jar -u
   ```

3. Verify that the version is updated:

   ```
   > /opt/bea/jrockit90_150_06/bin/java -jar tzupdater.jar -V
   ```

   ```
   tzupdater version: 1.0.0-b03
   JRE time zone data version: tzdata2007a
   Embedded time zone data version: tzdata2007a
   ```

4. Run the built in tests to test the new timezone data. If nothing is printed the tests has succeeded.

   ```
   > /opt/bea/jrockit90_150_06/bin/java -jar tzupdater.jar -t
   ```

Error Handling

The TZUpdater tries to restore the original state when it has encountered an unexpected error, such as lack of disk space. Such errors will generate a `TzRuntimeException`.

System-wide Usage

Stop any running instances of the JDK/JRE that you will operate on before running the TZUpdater for that JDK/JRE.

It is possible for systems to accrete multiple copies of JDK/JRE images, so you might need to apply the tool individually to each JDK/JRE image. Microsoft Windows users can use the desktop search utility to find each image. To locate multiple installed copies of the JDK/JRE on a UNIX derivative system, follow these steps:

1. Find locally installed JDK/JRE instances for UNIX derived systems:

   ```
   /usr/bin/find DIRPATH -fstype nfs -prune -o -fstype  autofs  -prune
   -o -name java -print -exec {} -version ;
   ```

   where `DIRPATH` is a directory path to search for installed Java SE instance, for example, `/usr`.

2. Automate updating of locally installed instances:

   ```
   /usr/bin/find DIRPATH -fstype nfs -prune -o -fstype  autofs  -prune
   -o -name java -print -exec {} -jar /ABSOLUTEPATH/tzupdater.jar -u ;
   ```

   where `DIRPATH` is a directory path to search for installed Java SE instance, for example, `/usr`. Replace `ABSOLUTEPATH` with the full pathname to the directory where `tzupdater.jar` is expanded.\

Determining Your TZUpdater Version

Use the command `tzinfo` to see which version of TZUpdater you're using. Enter the command from your JRockit JDK installation directory's `bin\` directory. The system will respond with complete version information; for example, if you entered:

```
\jrockits\R27.3.0_R27.3.0-45_1.5.0\bin\tzinfo
```

```
The system would respond:
```

```
java version "1.5.0_10"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_10-b03)
BEA JRockit(R) (build R27.3.0-45-79733-1.5.0_10-20070330-1521-windows-ia32,
compiled mode)
```

```
Time zone data version: tzdata2006k
```

Removing TZUpdater Changes

Stop any running instances of the JDK/JRE that you will operate on before running the TZUpdater for that JDK/JRE.

The modifications the current TZUpdater results in can only be manually removed by following these steps:

1. Locate the `zi` directory under the modified `JAVAHOME/jre/lib` directory. This is the newer data.
2. Locate a `zi.tzdata*` directory in the same `JAVAHOME/jre/lib` directory. This is the replaced, older data.
3. Obtain the currently installed timezone data version from the command `java -jar tzupdater.jar -V`.
4. Rename the current `zi` directory to something like "`zi.tzdata2007a`" or whatever the version command in step 3 gave. Make sure that this renaming does not conflict with the older data directory.
5. Rename the older data directory to `zi`.
6. Validate the change in the currently active timezone data by executing `java -jar tzupdater.jar -V`.
7. Restart your applications on the updated JDK/JRE instance.

Known Issues

The tool has a few restrictions due to the TimeZone API and implementation constraints.

Table 22-2 Known issue in the TZUpdater.

Issue	Explanation of the known issue
Time zone display names	This tool will not update time zone display names of time zones that are completely new or have display name related changes. An example is that Europe/Podgorica is a new time zone ID in `tzdata2006n` due to the Serbia/Montenegro split. Another example is that America/Indiana/Knox changed from Eastern Time to Central Time on April 2, 2006, which change appeared since `tzdata2006a`. The time zone display names for America/Indiana/Knox were changed in Java version 1.5.0_08.
JDK 1.1 time zone ID compatibility	In tzdata2005r, the Olson time zone database added EST, MST, and HST with no DST rules (i.e., standard time only). These IDs conflict with the same IDs in JDK 1.1. When users need the JDK 1.1 compatible time zone IDs rather than the complete set of the Olson time zone IDs, these incompatible IDs need to be removed. See also Sun's bug 6466476.
`TimeZone.getAvailableIds(int)` and `TimeZone.getRawOffset()` limitation and JCK	These TimeZone methods do not take a time stamp based on the API design assumption that a time zone's GMT offset never changes. Therefore, it is not possible to keep their return values consistent all the time. There is JCK test case TimeZone2014 that the return values of these methods are consistent. This test case fails if and only if there are any time zones of which GMT offsets will change in future time. A workaround fix was added to 1.5.0_4 and 1.6.0, which involved a time zone data file format change. Therefore, two versions of time zone data archives need to be provided with the tool in case that a future GMT offset change is involved in any time zones. See Sun's bug 5055567 for details. **Note:** The JCK test case fails without the 5055567 fix until the actual GMT offsets transition occurs.
Software Package Management errors	The current TZUpdater works outside of the native operating environment software package management infrastructure. Once you have used TZUpdater to install newer time zone data files, commands such as Solaris `pkgchk` will report errors concerning the files altered by TZUpdater. These are files under the `jre/lib/zi` directory.

CHAPTER 23

Oracle JRockit Mission Control Use Cases

This chapter demonstrates various ways Oracle JRockit Mission Control can be used to monitor and manage application running on the Oracle JRockit JVM. It includes the following use cases:

- Analyzing System Behavior with the JRockit Management Console
- Analyzing System Problems with the JRockit Runtime Analyzer
- Detecting a Memory Leak

Analyzing System Behavior with the JRockit Management Console

Marcus wants to monitor his application, DemoLeak which he's running on an instance of the JRockit JVM, to ensure that he has tuned it to provide the best possible performance. To do this, he will run the JRockit Management Console concurrent with the application run. The Management Console will provide realtime information about memory, CPU usage, and other runtime metrics. The Management Console is a multi-tabbed interface, each tab allowing him to monitor and/or manage an aspect of a running application. Which tabs his version of the Management Console uses depends on which Java plug-ins he has installed with the console. When fully-implemented, the console will include eight tabs and one menu, which map to seven plug-ins.

Getting Started

To get started, Marcus launches the JRockit Mission Control Client from the command prompt, by entering:

```
jrockit\bin\jrmc
```

While the JRockit Mission Control Client is starting up, he launches the DemoLeak application. At the command prompt, he enters:

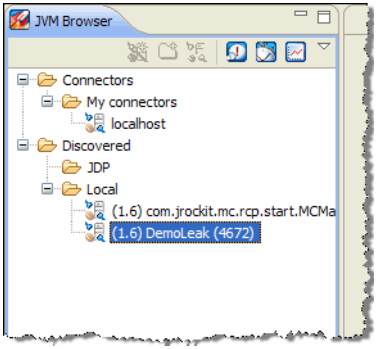
```
jrockit\bin\java DemoLeak
```

Next, he starts the Management Console with a local connection.

To launch the Management Console, Marcus does the following:

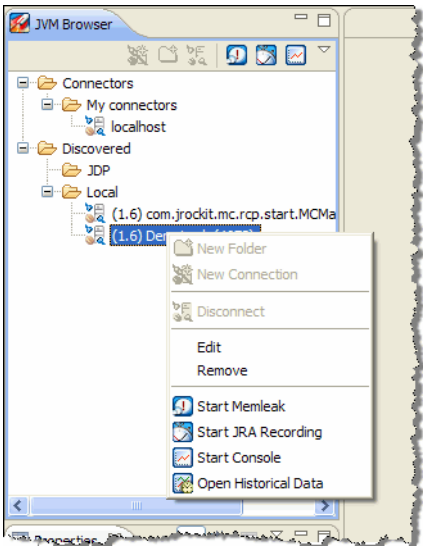
1. In the JRockit Browser, he locates the JRockit JVM instance to which he wants to connect. In this case, it will be the one running the DemoLeak class under **Discovered/Local**.

Figure 23-1 Locating the Appropriate JRockit JVM Instance



2. He right-clicks the mouse to open a context menu for the connection.

Figure 23-2 Context Menu for Selected JRockit JVM Instance



3. He selects **Start Console**.

   After a few moments, the Management Console appears in the right panel of the JRockit Mission Control Client. Note that in Marcus's implementation of the JRockit Mission Control Client, he can see the following tabs:

   – Overview
   – MBean Browser
   – Memory
   – Threads
   – Runtime
   – Triggers
   – Exception Count
   – Method Profiler

Analyzing Memory Usage

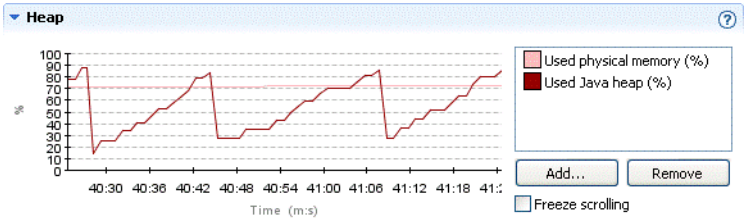
One way to spot problems with application performance to is to see how it uses memory during runtime. To analyze how his application is using the memory available to it, Marcus will use the

Memory Tab. This tab focuses on heap usage, memory usage, and garbage collection schemes. The information provided on this tab can greatly assist Marcus in determining whether he has configured the JRockit JVM to provide optimal application performance.

To analyze memory usage, Marcus does the following:

1. First, he examines the Heap graph, which shows the used Java heap growing until it reaches 80% to 90% of the available heap before a garbage collection is triggered. At that, point the graph falls back, indicating that new heap space is available. As the graph in Figure shows, this cycle repeats itself throughout the run.

Figure 23-3 Heap Graph



2. Next, he takes a look at the Memory Statistics and Garbage Collector panels, which show additional information about memory usage and the garbage collector, respectively. If necessary, Marcus can change some of the values from this tab; for example he could change the allocated Java heap size or the garbage collection strategy if he felt that those originally selected weren't allowing the application to run optimally.

Plotting Garbage Collection Times

Next, he decides to see the duration for each garbage collection. Overly long garbage collection times are a common cause of poor application performance. To see the duration of the garbage collections, Marcus can plot this information on the Heap graph.

The graphs shown in the various tabs are all preconfigured with a few useful default attributes, but any numerical attribute from any MBean can be added. In addition to the standard MBeans in J2SE 5.0 and the JRockit JVM specific MBeans, JRockit Mission Control itself provides so called synthetic MBeans that derives attributes from multiple other attributes. One such attribute is the garbage collection times

Note: The attribute for garbage collection durations is called PauseTimes even though the Java application isn't necessarily paused during the whole garbage collection. When a concurrent garbage collector is in use, the garbage collector runs concurrently with the Java application for the most part of the garbage collection duration. The misleading

naming of the attribute is a known issue and will be fixed in upcoming releases. The correct name of the attribute would be Duration.

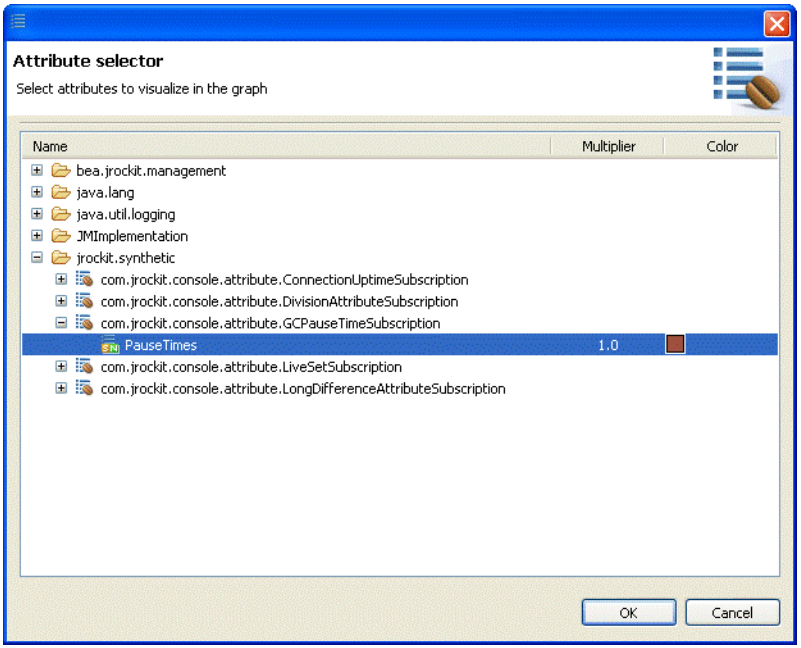
To do this, Marcus does the following:

1. In the Memory tab, he clicks **Add…** (to the right of the Heap graph).

   The Attribute Selector appears.

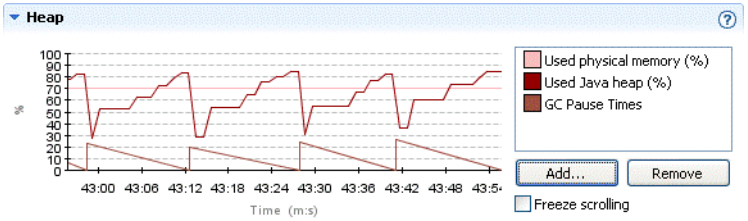
2. He drills down to the PauseTimes attribute, as shown in Figure 23-4, select it and clicks **OK**.

Figure 23-4 Selecting an Attribute to Add to the Heap Graph



The new attribute should now be shown in the Heap graph. This synthetic attribute is a somewhat special in that it only shows values just before and after a garbage collection, causing the triangular-shaped plot, as shown in Figure 23-5. The value is shown in milliseconds.

Figure 23-5 Heap Graph with Pause Time Plot Added



Setting an Alert Trigger

In his search for bottlenecks in the system, Marcus looks at the CPU load graph and notices that the CPU load for the JVM sometimes hits the roof. Marcus would like to know how often this happens for a longer period of time. Instead of staying and watching the CPU graph continuously he sets an alert trigger to alert him whenever the CPU load generated by the VM is high for a longer period of time.

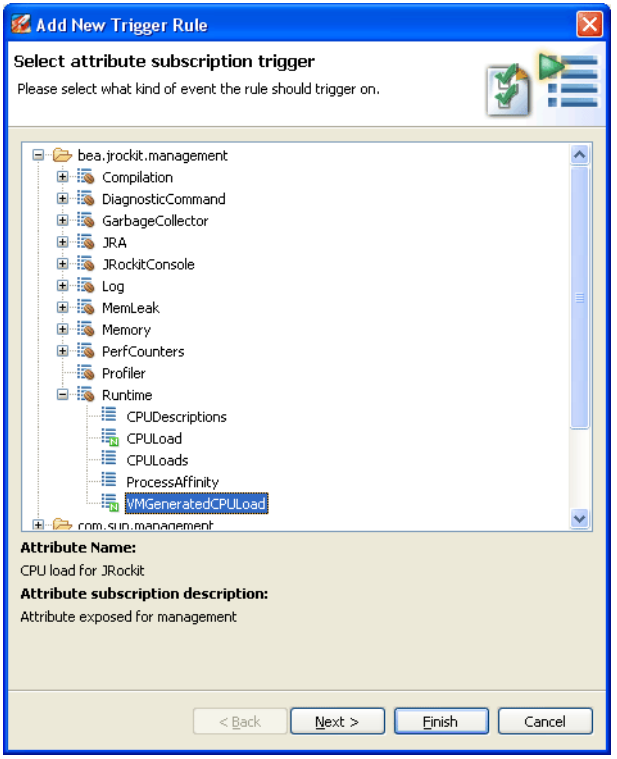
To set the alert, Marcus does the following:

1. Marcus goes to the Triggers tab and clicks **Add…** (under Trigger Rules).

   The Add New Trigger Rule wizard appears.

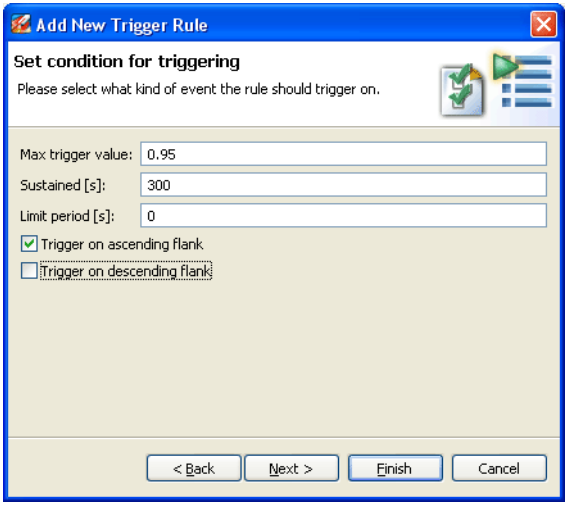
2. He drills down and selects the VMGeneratedCPULoad as shown in Figure 23-6. He then clicks **Next >**.

Figure 23-6 Selecting an Attribute to Trigger On



3. He enters the conditions as shown in Figure 23-7.

Figure 23-7 Setting Conditions for Triggering

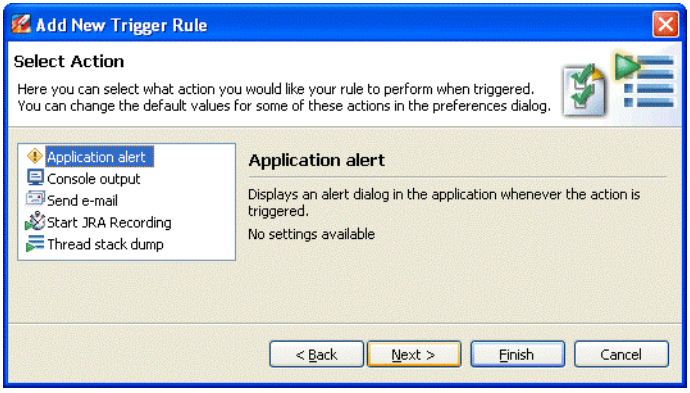


Since the CPU load value ranges from 0-1, Marcus sets the **Max trigger value** to 0.95. Marcus wants to be alerted when the CPU usage is high for at least five minutes and sets **Sustained [s]** to 300. He sets the **Limit period [s]** to 10 to prevent triggers less than 10 seconds apart.

4. He clicks **Next >**.

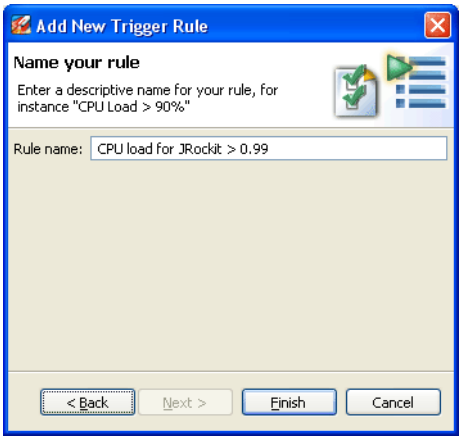
5. The Add New Trigger Rule: Select Action dialog box appears (Figure 23-8). Marcus selects **Application alert** and clicks **Next >**.

Figure 23-8 Selecting the Trigger Action



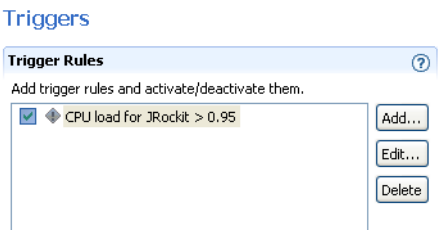
6. Marcus skips the optional constraints of when to arm the trigger by clicking **Next >**.
7. Marcus names the rule as shown in Figure 23-9 (`CPU load for JRockit > 0.95`) and clicks **Finish**.

Figure 23-9 Naming the Rule



8. Marcus then activates the new rule by checking the box next to the rule name.

Figure 23-10 Trigger Selected



9. To verify that he is getting useful data, Marcus then returns to the Memory tab and checks the CPU activity. He notices that the Trigger Alerts dialog box doesn't appear, so he edits the rule by going back to the Triggers tab, selecting the rule and lowering the **Max trigger value** to 0.90 or so under **Trigger Condition**.
10. Since Marcus doesn't want the Triggers Alert dialog box to appear every time an event is triggered, he will uncheck **Show dialog on alerts** to prevent the this from happening (he can display the dialog from the Window menu whenever he wants).

Profiling Methods Online by Using the Console

Next, Marcus wants to see how many times and for how long some specific methods have run, a process called method profiling. JRockit Mission Control has two tools for profiling methods:

- To run create a runtime analysis with the JRockit Runtime Analyzer (JRA), which we will demonstrate in Analyzing System Problems with the JRockit Runtime Analyzer. While this is best way to find out which methods are most likely affecting performance, it is also the more complex tool to run.
- By using the Method Profiler tab in the Management Console. This tool provides efficient and detailed method profiling while requiring a minimal amount of overhead and system intrusion. It also allows you to profile an application for which you are already collecting and viewing other information on the console.

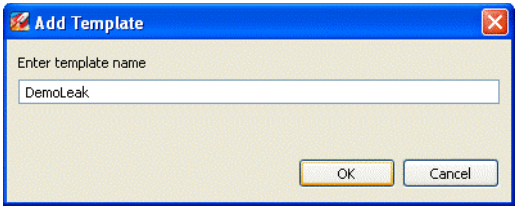
To profile methods by using the Method Profiler tab, Marcus does the following

1. First, he needs to create a method profiling template by going to the Method Profiler tab and click **Add…** in the Templates panel.

   The Add Template dialog box appears.

2. He enters a name for the new template in the Add Template dialog box, as shown in Figure 23-11.

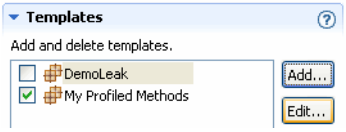
Figure 23-11 Template Name Added



3. He then clicks **OK**.

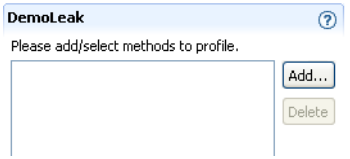
   The dialog box closes and the new template is added to the list (Figure 23-12).

Figure 23-12 DemoLeak Template Added



4. He enables the new DemoLeak template by checking the box in front of the name, then selects the template and clicks **Add…** in the DemoLeak panel (Figure 23-13).

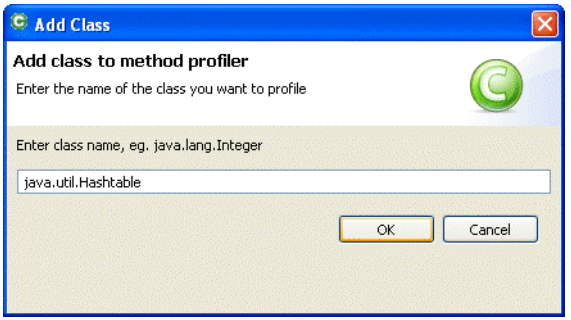
Figure 23-13 DemoLeak Panel



Add class to method profiler dialog box appears.

5. In the Add class to method profiler dialog box, he enters `java.util.Hashtable`, as shown in Figure 23-14

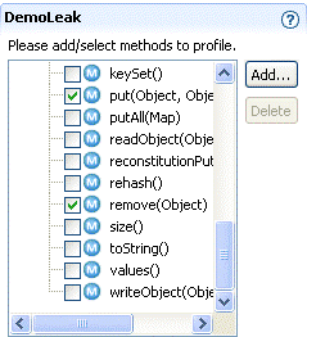
Figure 23-14 Adding java.util.Hashtable Class



6. He clicks **OK**.

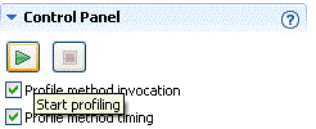
7. In the DemoLeak panel, he expands the `java.util.Hashtable` class, scrolls down and checks the boxes in front of the `put(Object, Object)` and `remove(Object)` methods, as shown inn Figure 23-15.

Figure 23-15 java.util.Hashtable Methods Selected



8. He then starts profiling by clicking the green play button in the Control Panel (Figure 23-16).

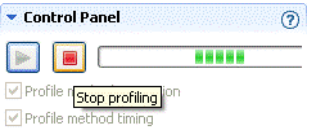
Figure 23-16 Start Profiling Button



9. To interpret the results, he examines the Profiling Information panel. He notes how the number of invocations of `Hashtable.put(Object, Object)` grows slightly faster than the number of invocations of `Hashtable.remove(Object)`.

10. Marcus then stops profiling by clicking the red stop button in the Control Panel (Figure 23-17).

Figure 23-17 Stop Profiling Button



Analyzing System Problems with the JRockit Runtime Analyzer

Fiona is not happy with how the application DemoLeak is performing. She is particularly concerned about the way her application performs the longer it runs. For example, while the application works fine early in its run, after a while, it starts reporting the wrong results and

throwing exceptions where it shouldn't. She also notices that eventually, it hangs at roughly the same time every time she runs it. To assess what the problem is, Fiona decides to create a runtime analysis by use the JRockit Runtime Analyzer (JRA).

The JRA is an on-demand "flight recorder" that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line, using the JRA Tool. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, and, in JRockit Mission Control 3.0, latency analysis.

Getting Started

To start the diagnostics process, Fiona does the following:

1. She starts the JRockit Mission Control Client from the command line by typing:

   ```
   jrockit\bin\jrmc
   ```

2. While the JRockit Mission Control Client starts, Fiona starts the DemoLeak by entering:

   ```
   jrockit\bin\java DemoLeak
   ```

Creating the Recording

Next, Fiona creates a JRA recording from a local connection. To do so, Fiona does the following:

1. Launches the JRockit Mission Control Client and locates in the JRockit Browser the JRockit JVM instance to which she wants to connect. This should be the one running the DemoLeak class under Discovered/Local.
2. Right-clicks the mouse to display a context menu for the selected connection.
3. Selects **Start JRA Recording** to launch the Start JRA Recording wizard.
4. Selects the connection to the JRockit JVM instance on which she wants to start the recording.
5. Selects filename and directory and types a descriptive name for the recording in the **Local filename** field. Note that the JRA recording file is created in the current directory of the JRockit JVM process, unless Fiona specifies a different path. If an old file already exists, it will be overwritten by the new recording.
6. Enters the desired length of the recording (in seconds) in **Recording time**.

   Note: If Fiona sets a recording length that is too short, for example, less than 30 seconds, she will probably not get enough sample data for the recording to be meaningful.

7. Selects the sampling options, as described in Table 23-1:

Table 23-1 Selected Sampling Options

Sampling Option	Description
Record samples of methods	Records samples of methods
Use gc sampling	Records garbage collection events
Use native sampling	Records samples of native code
Compress recording	Compresses recording to a zip file
Selected JRockits	Shows the JRockit JVM instance from which she will create her recording

8. Clicks **Finish**.

   The JRA recording progress window appears. When the recording is finished, it loads in the JRA. Fiona will now look at the JRA Recording.

Looking at the Recording

Next, Fiona will use the JRockit Mission Control Client to view the JRA recording. First, she opens the General tab by doing the following:

1. In the JRockit Mission Control Client, clicking **File > Open file > Open JRA Recording**.
2. Locating and selecting the recorded file and clicking **Open**.
3. Clicking **OK**.

   The JRA General tab for that recorded file now opens, allowing Fiona to view the data in the recording. The General tab contains information on the JVM, your system and your application. It is divided into the panels described in Table 23-2:

Table 23-2 General Tab Sections

Data Field	Description
General Information	Contains all general information about the JVM, operating system, recording time, and so on.
Memory Usage	Contains information on how the JRockit JVM is using the memory.

Table 23-2 General Tab Sections

Data Field	Description
VM Arguments	Lists all startup options that were used.
Allocation	Contains information on how your application allocates memory on the Java heap.
Threads	Contains information on thread usage.
Exceptions	Contains exceptions related information.

By looking at this tab, Fiona is able to verify which version of the JVM she was running. She can also see that large object were allocated at a rate, or "frequency", of 22.153 MB per second while small objects were allocated at a significantly faster rate of 261.983 MB per second.

Examining the Methods Tab

Next, Fiona will look at the Methods tab. The Method tab lists the top hot methods with their predecessors and successors during the recording. The Methods tab is divided into the following panels described in Table 23-3:

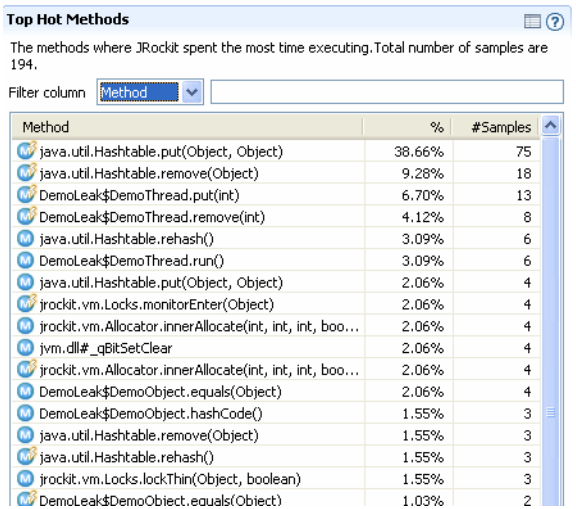
Table 23-3 Methods Tab Panels

Field	Description
Top Hot Methods	A listing of the top hot methods. A hot method is defined as the methods where the JVM spends most of its time during application execution. Being "hot" might indicate that a specific method is causing system problems.
Predecessors	A listing of all methods called prior to calling the method Fiona selected in the Top Hot Methods list. This information can be helpful in determining if some aspect of a certain method is complicit is poor system performance. If Fiona selects too many methods, no information will appear in this section.
Successors	A listing of all methods called after the calling method that Fiona selected in the Top Hot Methods list. This information can be helpful in determining if some aspect of a certain method is complicit is poor system performance. If Fiona selects too many methods, no information will appear in this section.

Examining the Top Hot Methods

The method sampling in the JRockit JVM is based on CPU sampling. The Top Hot Methods section lists all methods sampled during the recording and sorts them with the most sampled method s first, as shown in Figure 23-18.

Figure 23-18 Top Hot Methods



Note: If Fiona enabled native sampling during the recording, she would see symbols with a pound sign, such as `jvm.dll#_qBitSetClear`. These denote functions in native libraries such as the JVM itself or various operating system libraries.

By looking at the list of top hot methods, Fiona sees that the three hottest methods are:

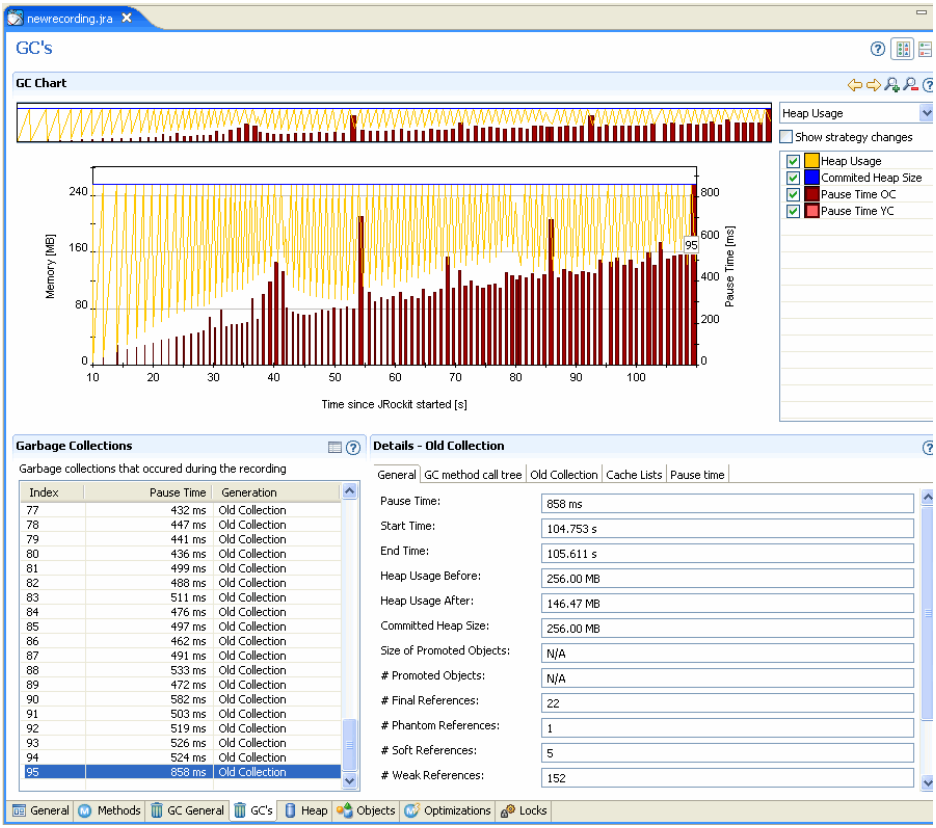
- `java.util.Hashtable.put(Object)`
- `java.util.Hashtable.remove(Object,Object)`
- `DemoLeak$DemoThread.put(int)`

Starting with this information, Fiona has a good idea of where to start looking for possible areas of concern. Fiona knows that the hottest methods are those that are sampled most often. In some situations, the number of samplings in the hottest methods will dwarf those of the less-hot methods. Hot methods are a good indicator of performance problems, especially memory leaks, because the high amount of sampling affects how much time the JVM has been executing the specific method.

Examine Garbage Collection Events

Next Fiona examines the GC's tab (Figure 23-19) to better understand system behavior and garbage collection performance during runtime.

Figure 23-19 GC's Tab



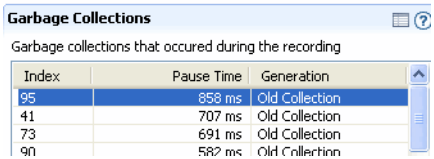
This tab is divided into the six panels described in Table 23-4

Table 23-4 GC Events Tab Panels

Panel	Description
GC Events Overview timeline	This timeline shows the entire recording based on when the recording is initially started. Fiona uses this information to refocus the Heap Usage graph.
Heap Usage graph	This graph shows heap usage compared to pause times and how that varies during the recording. When Fiona selects a specific area in the GC Events Overview, she only sees that section of the recording. She can change the graph content in the Heap Usage drop-down list (marked 6 in Figure 23-19) to get a graphical view of the references and finalizers after each old collection.
Garbage Collections events	This list shows all garbage collection events that occurred during the recording. When she clicks a specific event, Fiona will see a flag in the Heap Usage graph for that particular event.
Details	This panel contains all the details about the specific garbage collection round. When Fiona selects a garbage collection in the Garbage Collection list, the tabs in the Details section change depending on whether or not she selected an old collection or a young collection.
Chart Configuration	This panel lets Fiona change the appearance on the active chart.
Heap Usage	Fiona uses this list to toggle the view on the Heap Usage chart to view References and finalizers. It shows different types of reference counts after each collection.

Looking at the data in the Garbage Collections panel (Figure 23-20), Fiona sees that the three longest garbage collection pause times are indexed 95 (856 ms), 41 (707 ms), and 73 (691 ms).

Figure 23-20 Garbage Collection Panel



This data implied that, as processing continued on her application, garbage collections were taking longer. Fiona now has additional evidence to help diagnose what might be causing her

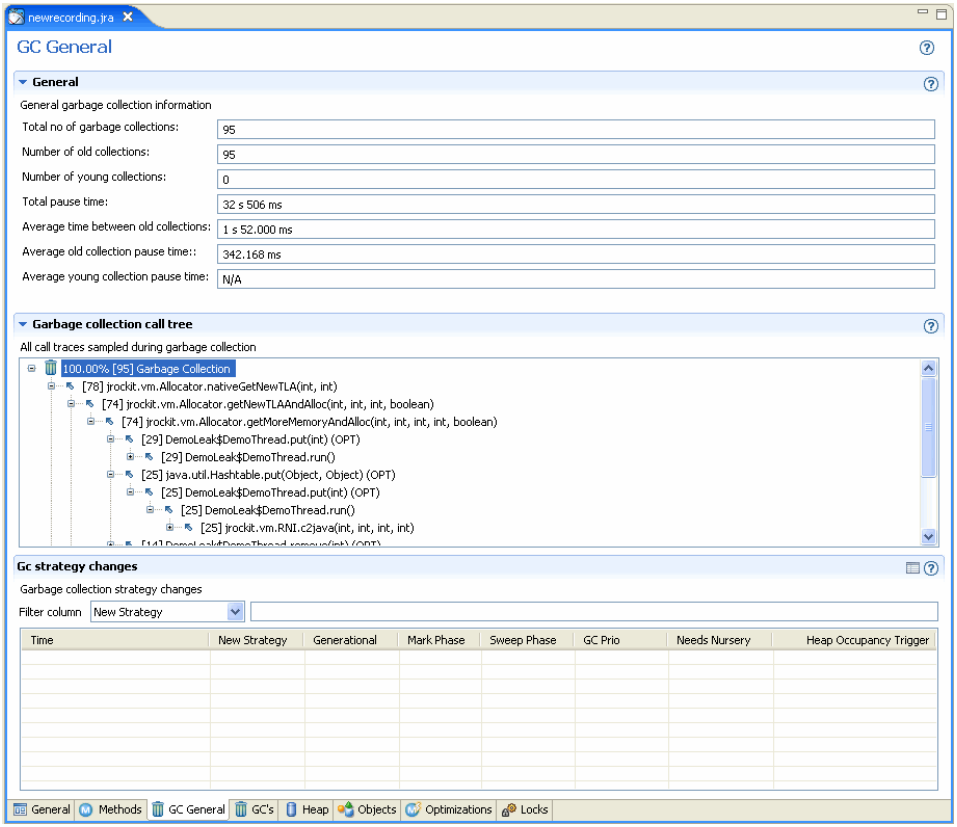
application's performance to deteriorate. She sees that garbage collection times are increasing, particularly later during runtime, and that the garbage collections are freeing less space on the heap. She can, with some confidence, predict that she is experiencing a memory leak.

Note: In this example, a memory leak is revealed to Fiona fairly quickly and she finds it because the evidence is obviously pointing in that direction. In most cases, a memory leak will reveal itself much more slowly and probably wouldn't be obvious on the GC's tab. Instead, a user would get better results by using the JRockit Memory Leak Detector, as described in Detecting a Memory Leak.

Examine the GC General Tab

Fiona can gain more insight into how garbage collection activity might be indicating a memory leak by looking at the GC General tab (Figure 23-21).

Figure 23-21 GC General Tab



This tab is divided into three panels that provide information about the garbage collection at a glance. This tab is divided into the panels described in Table 23-5:

Table 23-5 GC General Tab

Panel	Description
General	This section shows overall statistics about the garbage collections during the entire JRA recording.

Table 23-5 GC General Tab

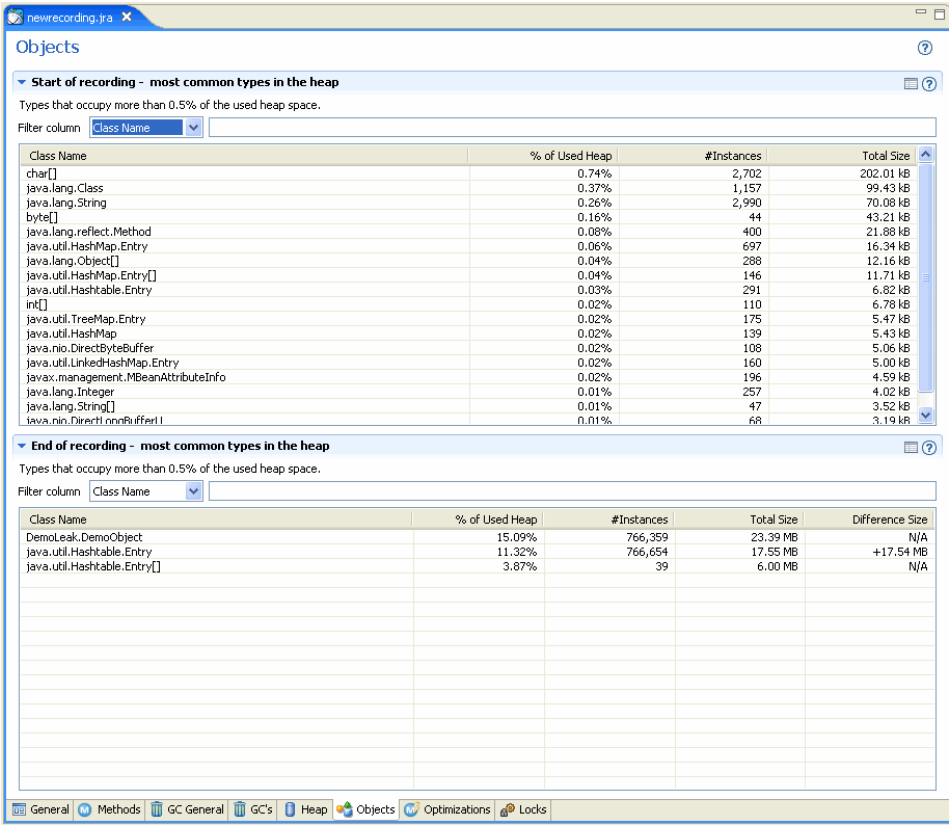
Panel	Description
Garbage Collection Call Tree	This section is a collection of all call traces that were sampled for all garbage collections for the JRA recording.
GC Strategy Changes	This section lists when a garbage collection strategy change took place and how it changed.

Fiona expands the stack tree down to user code and sees that many allocations are from the `hashtable` type, which indicates that this type is allocation intense. Reducing the allocation of this type would probably reduce the pressure on the memory management system.

Compare Object Statistics

Next, Fiona decides that it would be helpful to compare object statistics collected at the beginning of the recording to those collected after the recording. At the beginning and at the end of a recording session, snapshots are taken of the most common types and classes of object types that occupy the Java heap; that is, the types of which the total number of instances occupy the most memory. The results are shown on the Objects tab (Figure 23-22).

Figure 23-22 Objects Tab



The Object Statistics tab is divided into the panels described in Table 23-6.

Table 23-6 Object Statistics Tab

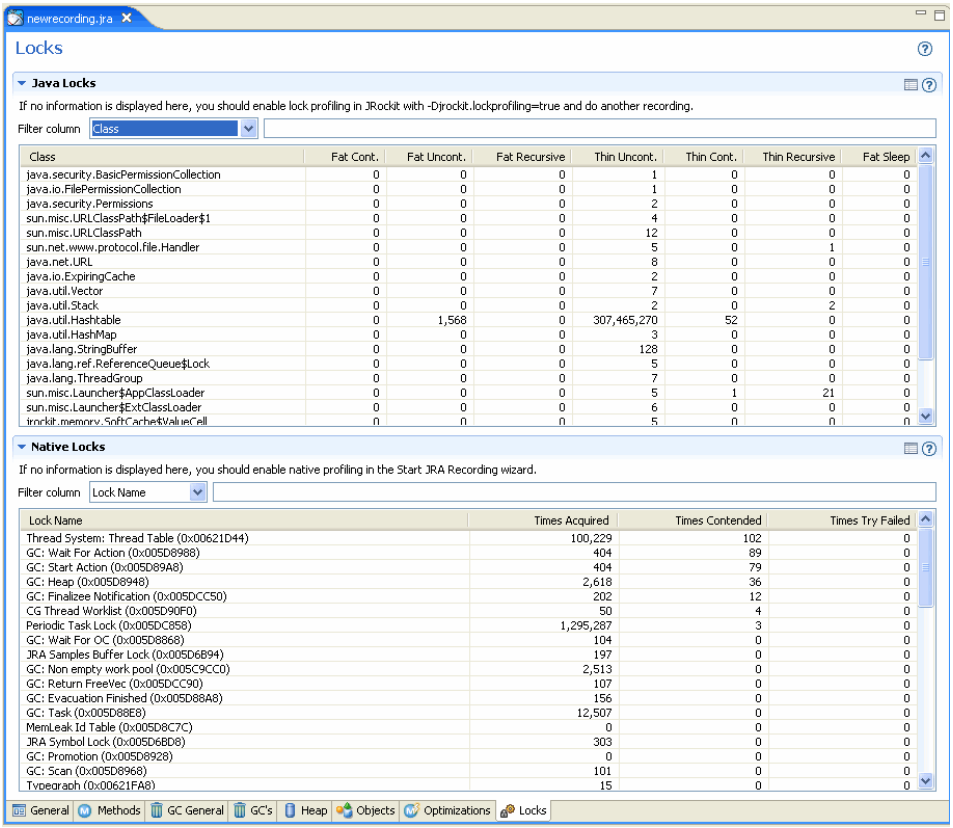
Panel	Description
Start of Recording	This section lists the most common types on the heap at the beginning of the recording.
End of Recording	This section lists the most common types on the heap at the end of the recording.

Fiona can again see that `hashtable` shows the most dramatic growth and is consuming the greatest amount of memory on the heap. This again is a strong indication of not only a memory leak but that said leak involves the `hashtable` object.

Examine Lock Profiling Information

Fiona then checks the lock statistics for clues to performance bottlenecks involving locks. She opens the Locks tab (Figure 23-23) to investigate this information for both her application and the specific JRockit JVM instance.

Figure 23-23 Locks Tab



The Lock Profiling tab is divided into the panels defined in Table 23-7.

Table 23-7 Lock Profiling Tab Panels

Panel	Description
Java Locks	This section lists all locks in the application.
Native Locks	This section lists all JVM internal locks.

By looking at the Java Locks panel, Fiona can see immediately that the `hashtable` type has taken over 300 million uncontended locks, compared to a relative few for other objects. While this information does not point directly to a memory leak, it is indicative of poor performance.

Since the lock is mostly uncontended, Fiona could optimize her application by switching to an unsynchronized data structure such as `hashmap` and provide synchronization only for the few cases where contention may occur.

Detecting a Memory Leak

Since Fiona determined that a memory leak is causing her application to run poorly, she can take advantage of the JRockit Memory Leak Detector to confirm her suspicions and begin corrective action. A memory leak occurs when a program fails to release memory that is no longer needed. The term is actually a misnomer, since memory is not physically lost from the computer. Rather, memory is allocated to a program, and that program subsequently loses the ability to access it due to program logic flaws.

Getting Started

To start the memory leak detection process, Fiona does the following:

Note: This procedure assumes that the application was stopped after the JRA recording was completed. Had Fiona not stopped the application, she would be able to skip step 1 and step 2

1. She starts the application by entering, at the command line:

   ```
   java DemoLeak
   ```

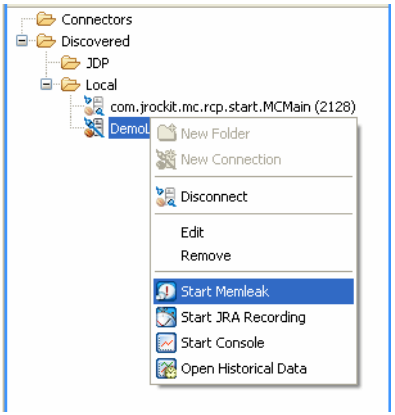
2. While the application starts, she creates a connection to the server on which the application is running.

3. Next, she starts the Memory Leak Detector by doing the following:

a. Right-clicking a Oracle JRockit JVM instance in the JRockit Browser to open a context menu.

b. Selecting Start Memleak (Figure 23-24).

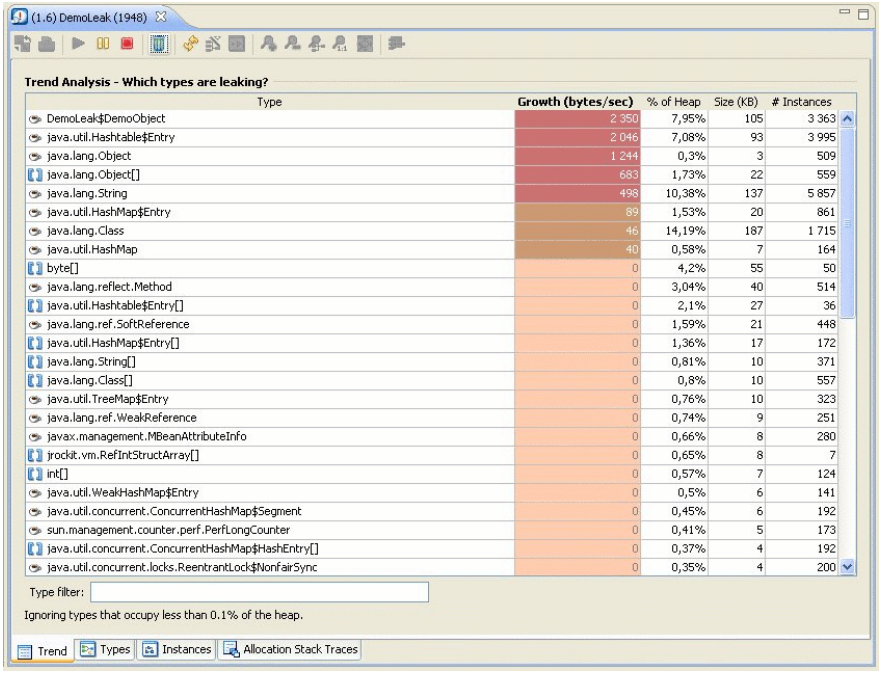
Figure 23-24 Starting The Memory Leak Detector from a Context Menu



Analyze the Java Application

Fiona starts the analysis from the Trend tab (Figure 23-25), which should open when she launches the Memory Leak Detector.

Figure 23-25 Trend Tab



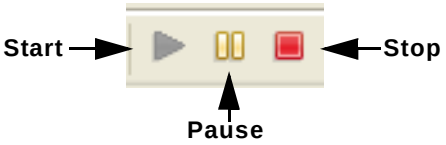
Tip: The trend analysis should be running by default. If it is not running, you can start it by clicking the start symbol among the trend analysis buttons (Figure 23-27).

The trend analysis page shows Fiona the statistics on memory usage trends for the object types within the application. The JVM collects this data during garbage collections, which means that at least two garbage collections must be done before any trends are shown.

Figure 23-26 Garbage Collection Button

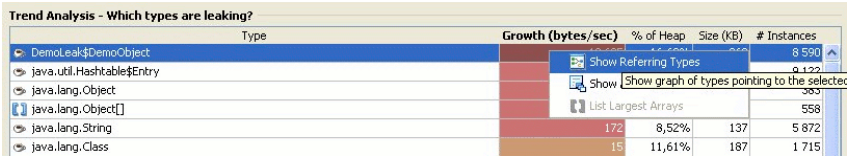
1. In order to speed up the process, Fiona clicks on the Garbage Collection button (Figure 23-26) a couple of times to start some garbage collections.

Figure 23-27 Trends Analysis Buttons



2. Fiona then pauses the trend analysis by clicking the pause symbol among the trend analysis buttons (Figure 23-27).

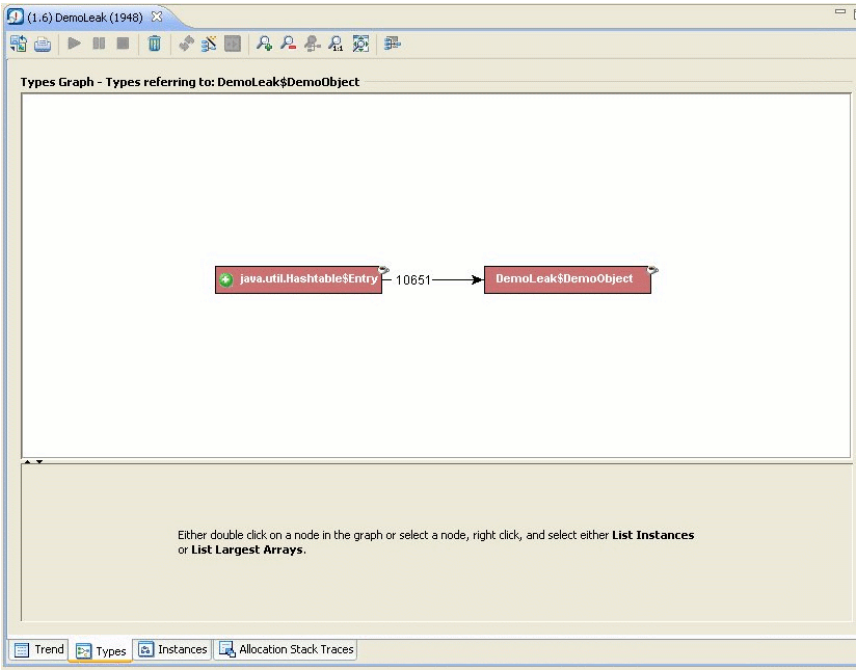
Figure 23-28 Show Referring Types



Fiona finds that a class named DemoObject shows the largest growth.

3. Fiona right-clicks the DemoObject class and selects “Show Referring Types” in the drop-down menu, as seen in Figure 23-28.

Figure 23-29 Types Tab

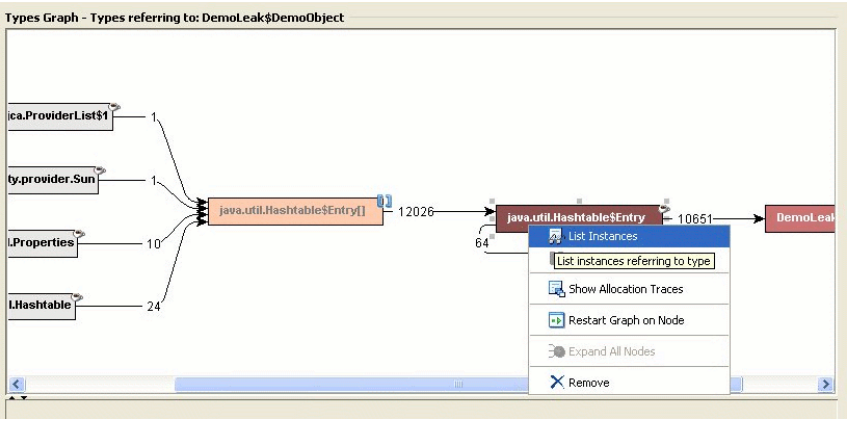


This opens the Types tab (Figure 23-29). Here Fiona can see that the DemoObjects are stored in hashtable entries.

4. Fiona clicks on the plus sign on the java.util.Hashtabe$Entry node to expand the graph to show types referring to the hashtable entries.
5. Fiona continues expanding the graph to the left until she finds nodes in gray that do not show any growth trend.

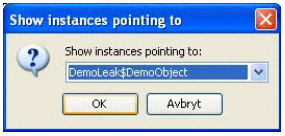
Now Fiona can see that the DemoObjects are indeed stored in hashtable entries. The next step for Fiona is to find out more about the instances holding on to the hashtable entries containing these DemoObjects.

Figure 23-30 List Instances



6. Fiona right-clicks the node that refers to the DemoObjects and selects List Instances in the drop-down menu, as seen in Figure 23-30.

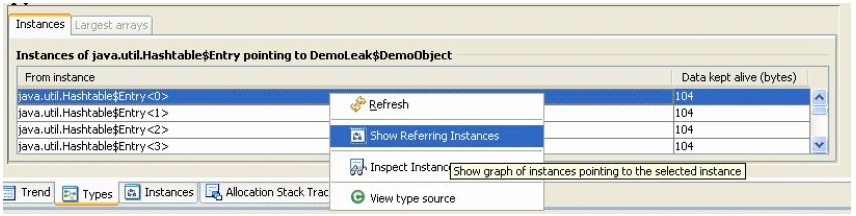
Figure 23-31 Show Instances Pointing To



Not all hashtable entries in the application point to the same type of objects, so Fiona gets a popup that asks her to select the type of references she is interested in.

7. Fiona selects DemoObject in the popup (Figure 23-31), since this is the type that she is interested in, and clicks Ok.

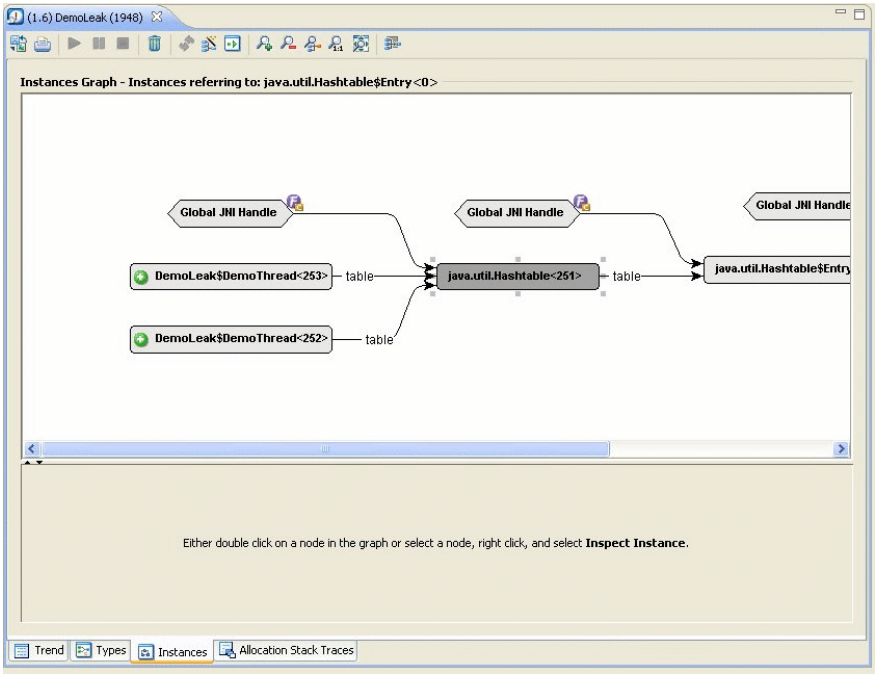
Figure 23-32 Show Referring Instances



At the bottom of the Types tab Fiona gets a list of instances of java.util.Hashtable$Entry that refer to DemoObjects.

8. Fiona right-clicks the topmost instance and selects “Show Referring Instances” (Figure 23-32) to start analyzing the instances that hold on to this particular hashtable entry.

Figure 23-33 Instances Tab



This opens the Instances tab, as seen in Figure 23-33.

9. Just as in the Types tab, Fiona clicks on the plus signs to expand the graph to the left.

Fiona finds that the DemoObject is stored in a hashtable, which is held by a DemoThread. This is the culprit causing the memory leak.

The Leak is Discovered

Judging from the evidence she collected using the Oracle JRockit Mission Control tools, Fiona was able to not only identify her system problem as a memory leak, but was able to locate exactly which object type was leaking the memory. The key to making the identification began with

noting in her JRA recording the increasing length of garbage collections and the types upon which those lengthening garbage collections were occurring. She then ran the Memory Leak Detector to pinpoint which of the questionable types were actually the source of the leak. She was able to spot types that continued to increase in their number of instances and allocations, obviously holding on to memory so that it couldn't be freed for allocating other objects. This, then identified the memory leak and where it was occurring.

Part IV Diagnostics and Troubleshooting

CHAPTER 24

About Diagnostics and Troubleshooting

If you ever encounter a problem running the Oracle JRockit JVM, you have a number of options available to you for resolving those issues. Many problems are within your ability, as a user, to fix yourself while others will require the intervention of Oracle JRockit's support organization.

What this Section Contains

The chapters in Diagnosing and Resolving Problems provide guidelines for selecting the best option for your situation. These chapters examine potential JVM problems based upon the symptoms that you observe when the issue arises. They provide instructions for obtaining greater detail on the problem, including possible sources and causes, and, where possible, solutions you can use to correct the problem. In situations where you need to escalate a problem to the support organization, this document also provides guidelines for the type of information you need to provide and instructions for gathering that data.

This section is compressed of these chapters:

- Diagnostics Roadmap provides an overall scheme to follow when troubleshooting problems with the JRockit JVM. This roadmap outlines the four steps you should follow to arrive at the best solution to your problem.
- The Oracle JRockit JVM Starts Slowly describes how you can recognize and troubleshoot a slow-starting JVM.
- Long Latencies describes how to recognize and troubleshoot long garbage collection pauses that adversely affect system performance.

- Low Overall Throughput describes how to recognize and troubleshoot when the application runs too slowly.
- The Oracle JRockit JVM's Performance Degrades Over Time decibels measures you can take when your application begins to behave erratically, return incorrect results, or throw `OutOfMemory` exceptions.
- The System is Crashing describes what to do when your system, whether the JVM or the application you are running, stops completely sending signals.
- Understanding Crash Files provides information on the crash files that the JRockit JVM creates if the JVM crashes.
- The Oracle JRockit JVM is Freezing describes what to do when the JVM or Java application becomes unresponsive but hasn't completely crashed.
- Submitting Problems to Oracle Support provides the best practices to follow when you need to report a Oracle JRockit JVM problem to Oracle Support.

CHAPTER 25

Diagnostics Roadmap

This chapter serves as a roadmap to how you should approach problems with the Oracle JRockit JVM. At its most basic, this roadmap is comprised of four steps you should follow to arrive at the best solution to your problem. The roadmap is comprised of these step:

- Step 1. Eliminate Common Causes
- Step 2. Observe the Symptoms
- Step 3. Identify the Problem
- Step 4. Resolve the Problem
- Step 5. Send a Trouble Report (Optional)

Step 1. Eliminate Common Causes

Often, if you encounter a problem running an application on the JRockit JVM, the cause is something extremely basic that you might be able to correct with only a small effort. Before diagnosing issues with the JVM, go through the checklist in Table 25-1 to make sure that you have ruled out the most common causes for JVM problems.

Note: The list is not in any particular order, which means that each item can be checked separately.

Table 25-1 Common Causes for JRockit JVM Problems

Have You...	Because...
Tried Reinstalling the JRockit JVM?	Sometimes an installation can have gone wrong, that is why you should try to reinstall the JRockit JVM and then try to reproduce the problem. You find the relevant installation instructions in *Installing* Oracle JRockit Mission Control.
Installed the Latest Patches from Other Software that are Dependent on the JRockit JVM?	The problem might have its origins in the application running on the JRockit JVM and whatever those origins are, they might have been fixed in later releases of that product.
Been Able to Reproduce the Problem with the Latest Updated Version of the JRockit JVM?	The problem that you encounter might have been fixed in a later release of the JRockit JVM. Make sure you have installed the latest release. If you are searching for a specific problem that you need to know if it has been fixed, see the Oracle JRockit JVM *Release Notes*. If you have a service agreement with Oracle, you can ask for a patch.
Been Able to Reproduce the Problem on the Same Machine?	Knowing that this defect occurs every time the described steps are taken, is one of the most helpful things you can know about it and tends to indicate a straightforward programming error. If, however, it occurs at alternate times, or at one time in ten or a hundred, thread interaction and timing problems in general would be much more likely.
Been Able to Reproduce the Problem on Another Machine?	A problem that is not evident on another machine could help you find the cause. A difference in hardware could make the problem disappear; for example, the number of processors. Also, differences in the operating system and application software installed might make a difference to the JRockit JVM. For example, the visibility of a race condition in the JVM or a user Java application might be influenced by the speed at which certain operations are performed by the JVM.
Tried to Disable the Code Garbage Collector?	Code garbage collection can be disabled, while Java heap garbage collection cannot. Although disabling the code garbage collector forces the JRockit JVM to use more memory, if the problem exists in code garbage collection, the JVM should perform normally when it is disabled. To disable code garbage collection, use the command-line option `-XXnoCodeGC` at startup.
Ensured You are Using a Supported Operating System (OS) with the Latest Patches Installed?	It is important to use an OS or distribution that supports the JRockit JVM and to have the latest patches for operating components. For example, upgrading libraries can solve problems.

Table 25-1 Common Causes for JRockit JVM Problems

Have You...	Because...
Ensured You are Running the Latest Version of Third-Party JNI Code?	If you are running JNI code from a third party vendor, make sure you have the latest version of that code installed. Look in the text dump, which you can create by using `jrcmd` (please refer to Using jrcmd for more information) to see the list of libraries (DLLs) that are loaded. From that list you can derive which updates you might need.
Tried `-XnoOpt` to Turn off Optimization?	Code optimization is a default process by which commonly-executed code is recompiled to make it run more efficiently. The first time the JRockit JVM runs a method, the method is compiled into machine code. This compilation is quick, but the resulting code isn't as efficient as it could be. While this code is acceptable for methods that are run once and discarded; however, if a method is used repeatedly, the system can get a performance boost if the code for that particular method is regenerated in a more efficient way. The JRockit JVM re-compiles—or "optimizes"—these methods to make the code as efficient as possible. While optimization makes code more efficient, the optimizing compilation takes longer. If the optimizer itself has bugs, `-XnoOpt` will help you identify problems related to those bugs.
Enabled Crash Dump Files on the JRockit JVM?	The information that is available in the crash dump files are of interest to the Oracle Support organization. If you have a Oracle Support Agreement, you can provide your support agency with crash dump files to better find what problems there might be. **Note:** If you do not have a Oracle Support Agreement, you should not enable crash dumps as they can grow quite large and occupy a lot of your disk space. Crash dump files can be very helpful to Oracle Support. Depending on the platform, crash dump files might not be enabled by default. For more information on crash dumps, please refer to Understanding Crash Files.

If none of these causes corrects your problem, proceed to Step 2. Observe the Symptoms.

Step 2. Observe the Symptoms

In most cases, when you encounter a problem, you will only see what's happening with your application. You won't know the underlying cause of the problem or where the problem is originating. Therefore, you need to be able to identify the symptom of the problem, rather than its root causes, before you can actually begin resolving it. Table 25-2 provides a list of the most

common problem symptoms that users tend to encounter when running into issues with the JRockit JVM. Adjacent to the symptoms will be links to chapters in this section that describe the most likely reason for the problem and provide detailed information on how to handle it.

Step 3. Identify the Problem

Once you have identified the symptom, you can usually identify the problem by following the procedures outlined in the relevant sections of the linked chapter. For example, your symptom might be that the JVM starts up unacceptably slow. By following the instructions outlined in the chapter linked to that symptom, you will be able to determine if your problem is with optimization or is for some other reason. Knowing the origin (and, where possible, location) of the problem will be critical to successfully resolving it.

Table 25-2 list common symptoms associated with JVM problems and where, in this guide you can find information pertaining to that symptom.

Table 25-2 Symptom/Chapter Matrix

If this is happening...	Go to...
The JVM crashes and produces dump information	The System is Crashing
The JVM takes too long to start up	The Oracle JRockit JVM Starts Slowly
Some transactions take too long to execute even though the overall throughput is good	Long Latencies
The overall throughput is too low	Low Overall Throughput
After running the JVM successfully for a while, it begins to perform poorly; for example, the following is happening: • The overall throughput degrades • The overall throughput is unstable • The JVM starts reporting the wrong results • The JVM is throwing exceptions where it shouldn't	The Oracle JRockit JVM's Performance Degrades Over Time
The JVM is freezing without crashing	The Oracle JRockit JVM is Freezing

Step 4. Resolve the Problem

Depending upon the type of problem you are encountering, you might be able to resolve it yourself. The chapter linked to the symptom (Table 25-2) will provide the steps you need to follow to effect that resolution. If the problem requires escalation to Oracle Support, the chapter linked to the symptom will describe the sort of information you will need to provide when you open a case with them.

Step 5. Send a Trouble Report (Optional)

If you are a licensed JRockit JVM user and you can't resolve the problem on your own, you can send a Trouble Reports to the Oracle Support organization (see Submitting Problems to Oracle Support).

CHAPTER 26

The Oracle JRockit JVM Starts Slowly

One major benefit of the Oracle JRockit JVM is that it is a Just-In-Time (JIT) compiling JVM (see How the JRockit JVM Compiles Code for information on how code is compiled), meaning the first time the JVM runs a method it is compiled into machine code. The JVM compiles all the classes to native code the first time they are called. This slows down the application run at startup when a lot of new methods are compiled, but in return the methods will run fast already the second time they're invoked. The methods most often run will later be recompiled by the JRockit JVM and therefore optimized further for the following runs, ensuring that your application runs even faster.

This section describes how you can recognize and troubleshoot a slow-starting JVM. It includes information on the following subjects:

- Possible Causes Behind a Slow Start
- Diagnosing a Slow JVM Startup
- Diagnosing a Slow Application Startup
- Timing with nanoTime() and currentTimeMillis()
- Recommended Solutions for a Slow Start

Possible Causes Behind a Slow Start

There are several possible causes for your application to seem slow in the beginning.

- One possible cause could be that your application is waiting for files to import or because a large number of methods need compiling at the beginning of your program.
- On rare occasions, the problem might lie in code optimization.
- Also, the problem might be caused by your Java application and not the JVM. This sort of problem often has to do with method or resource synchronization and probably can best be handled by the Java developer responsible for the application.

Special Note If You Recently Switched JVMs to the JRockit JVM

If you've recently switched from another JVM to the JRockit JVM, you might think the JVM is starting too slowly. This is particularly noteworthy if you've switched to the JRockit JVM as your production JVM from a third-party development JVM. Actually, what might appear to be a slow start is normal operation. The JRockit JVM is designed for use with long-running applications. As such, you should expect longer start-up times as code is compiled and optimized. The JVM is not starting slow, it just has more information to process during startup. Once all of the methods are compiled, the JVM should run much faster.

Diagnosing a Slow JVM Startup

To see if your application compiles a lot of methods in the beginning of the run, you can start it with the option `-Xverbose:codegen`.

With this option set, the following information will be shown about the method that is being compiled: name, memory location, duration of the compilation, and the amount of time that has passed since the compilation started. See Listing 26-1 for an example of a `-Xverbose:codegen` print out.

Listing 26-1 -Xverbose:codegen

```
[codegen] #775 1 (0x2) n jrockit/memory/AtomicInt.<init>(I)V
[codegen] #775 1 (0x2) n @0x7D62ED90-0x7D62ED9C  0.09 ms (277.99 ms)
[codegen] #776 1 (0x2) n jrockit/memory/AtomicInt.set(I)V
[codegen] #776 1 (0x2) n @0x7D62EDA0-0x7D62EDAA  0.08 ms (278.06 ms)
```

Conversely, if you have a lot of methods that need to be compiled during startup, you will have a fairly long startup time compared to if you would have few.

Once the JIT compilation is complete, the compiled methods will no longer need as much time to run; however, the JRockit JVM continuously optimizes frequently-used methods during the application run which, at times, can give the appearance of the JVM running slowly.

Diagnosing a Slow Application Startup

The startup time can also be extended if you have an application that is searching for a certain file, for example, a data file. If you suspect that your application is causing the slow start, create a JRA recording and analyze the application data in the JRockit Runtime Analyzer. This tool is included in Oracle JRockit Mission Control.

If you are running the JRockit JVM R27.1 or later with JRockit Mission Control 2.0 or later, complete instructions for creating and interpreting a JRA recording are available from the Oracle JRockit Mission Control online help.

Timing with nanoTime() and currentTimeMillis()

To measure timing inside your application you can use the methods `System.nanoTime()` and `System.currentTimeMillis()` in your application. Inserting these methods in your application will, of course, consume resources at runtime but the performance impact should be minimal.

System.nanoTime()

This method returns a monotonic timer value by using the most precise available system timer. The returned value is in nanoseconds, however the factual resolution of the timer can vary between OS and hardware. Note that there is no conventional zero point to which you can relate the timer value. Hence, you must take the time at least twice in order to get any meaningful data.

`nanoTime()` uses different methods on different operating systems:

- Windows: `QueryPerformanceCounter()`
- Solaris: `gethrtime()`
- Linux: `clock_gettime()` in `librt` if available, else `gettimeofday()`.

To get information about timer resolution and, on Linux, the method used to get a time value, start the JRockit JVM with the option `-Xverbose:timing`.

Here is an example of a verbose timing report on Windows:

```
[INFO ][timing ] Counter timer using resolution of 1779720000Hz
```

System.currentTimeMillis()

This method returns the current time in milliseconds. The current time is defined as the time since 00:00:00 UTC, January 1, 1970.

Milliseconds and nanotime at application startup

To get the values of `System.currentTimeMillis()` and `System.nanoTime()` at the time the JVM started, use the command line option `-Xverbose:starttime`. Verbose output for `starttime` might look like this:

```
[startti] VM start time: 1152871839957 millis 171588375730523 nanos
```

The `millis` value is the same value that `System.currentTimeMillis()` would provide and the `nanos` value is the same value that `System.nanoTime()` would provide.

Recommended Solutions for a Slow Start

This section provides information on possible solutions for a slow start.

- Tune for Faster Startup
- Eliminate Optimization Problems
- Eliminate Application Problems
- Open a Case with Oracle Support

Tune for Faster Startup

Sometimes the problem may be with how the JVM is tuned using command line options. See Tuning For Faster JVM Startup for tips on how to tune the JVM for faster startup.

Eliminate Optimization Problems

Since, on some rare occasions, optimization can be the cause of a slow start, you should eliminate it as a cause before you move on to any other solution.

If you suspect that the problem is with optimization, you can disable optimization completely by starting the JVM with the `-XnoOpt` startup command. This command tells the JVM not to

optimize any code. If the JRockit JVM starts more quick after running with `-XnoOpt`, you can assume your are experiencing optimization problems. You should report this to Oracle Support.

As a workaround you can try to exclude methods that take too long to optimize. To do so, use the Ctrl-Break Handler `print_threads` to make a thread dump (please refer to Running Diagnostic Commands for more information). This output will identify any methods that are causing optimization problems. You can then use an *optfile* to exclude that method from the optimization process (please refer to Creating and Using an Optfile for more information).

Eliminate Application Problems

If you determine that the slow start is due to problems in your Java application, you need to investigate what is causing that problem from the application viewpoint. The problem will most likely be with a method that is the victim of unnecessary synchronization or an insufficient number of synchronized resources. Try to locate the methods that are causing the bottleneck and, if possible, rewrite the code of your Java application.

Open a Case with Oracle Support

If you feel that it is the Oracle JRockit JVM that is taking too long to generate the code for each method or if none of the tuning solutions suggested in Tuning For Faster JVM Startup resolve the problem, you will need to open a case with Oracle Support. You can find instructions on how to report a problem to Oracle, including the sort of information to include, in Submitting Problems to Oracle Support

CHAPTER 27

Long Latencies

Long latencies may for example manifest as single transactions that time out in a transaction based application while the overall performance is good. The problem usually lies in uneven performance and non-deterministic latencies.

This chapter includes these topics:

- The Problem is Usually with Tuning
- If All Else Fails, Open a Case With Oracle Support

The Problem is Usually with Tuning

Long latencies often indicate that your application is not tuned for short and deterministic pause times. Before engaging in time-consuming troubleshooting and mitigation tasks, you should retune the Oracle JRockit JVM to optimize for short pause times and restart the application. For detailed instructions on how to tune the JVM for short pause times, please refer to Tuning For Low Latencies.

There is a certain trade-off between low latencies and high overall application throughput. High latencies that cause transactions to time-out are often caused by garbage collection pauses. To reduce the individual garbage collection pauses the garbage collector runs in a *mostly concurrent* mode, where the garbage collection is, for the most part, done while the Java threads are still running. This causes some extra work for the garbage collector, which has to keep track of changes during the concurrent phases of the garbage collection. The garbage collections will also be less efficient, since objects that are allocated during the concurrent garbage collection will not be garbage collected until the next garbage collection. This can force the JVM to collect garbage

more often. Also, the heap might become more fragmented when compaction is limited to reduce the pause times caused by compaction. All this reduces garbage collection pauses, but it will also have a negative impact on overall throughput.

You can increase the overall throughput while keeping the latencies low by allowing longer garbage collection pauses or by manually tuning the garbage collection using the tips in Tuning For Better Application Throughput.

Troubleshooting Tips

This section lists some latency troubleshooting hints that apply for mostly concurrent garbage collection, for example `-XgcPrio:deterministic`, `-XgcPrio:pausetime`, `-Xgc:gencon` and `-Xgc:singlecon`.

GC Trigger Value Keeps Increasing

The garbage collection trigger (gctrigger) value determines how much free heap space should be available when a concurrent garbage collection starts in order to allow the Java threads to continue allocating objects during the entire garbage collection. The gctrigger value changes in runtime in order to avoid situations where the heap becomes full during the concurrent garbage collection. Monitor the gctrigger value in `-Xverbose:memdbg` outputs or in JRA recordings. If the gctrigger value keeps increasing, the load on the application is to high for the concurrent garbage collector. Decrease the load on the application.

GC Reason for Old Collections is Failed Allocations

Monitor the garbage collection reasons for the old collections with `-Xverbose:memdbg` or JRA. The normal garbage collection reason for a mostly concurrent old collection is “heap too full”. If the old collections are frequently triggered due to failed object allocation, the GC trigger is too low. Increase the GC trigger value using the command line option `-XXgcTrigger`, or decrease the load on the application.

Long Young Collection Pause Times

Monitor the pause times for young collections in `-Xverbose:gcpause` outputs or JRA recordings. If the young collection pause times are too long, decrease the nursery size using the `-Xns` command line option or run a single generational garbage collector.

Long Pauses in Deterministic Mode

Monitor the garbage collection pause times in a JRA recording. Check the pause parts for pause times that are too long. If the pause parts for `Compaction` are too long, decrease the pause target. If the pause parts in `Mark:Final`, especially the ones concerning `RefrenceQueues`, are too long you may have a problem with many `java.lang.ref.Reference` objects in your application. The best way to handle this would be to re-design the Java application using fewer reference objects. You could also try decreasing the heap size, which will cause reference objects to be handled more often and reduce the amount of reference objects to handle at each old collection.

If All Else Fails, Open a Case With Oracle Support

If none of the tuning solutions suggested in Tuning For Low Latencies or in Troubleshooting Tips resolve the problem, you will need to open a case with Oracle Support. You can find instructions on how to report a problem to Oracle, including the sort of information to include, in Submitting Problems to Oracle Support.

CHAPTER 28

Low Overall Throughput

Low overall throughput manifests for example as a low score in benchmarks, too few transactions executing per minute in a transaction based system or long processing times for large batches of data.

This chapter includes these topics:

- The Problem is Usually with Tuning
- If All Else Fails, Open a Case With Oracle Support

The Problem is Usually with Tuning

A low overall throughput usually means that your JVM is not tuned to maximize application throughput. Before engaging in time-consuming troubleshooting and mitigation tasks, you should retune the Oracle JRockit JVM to optimize application throughput and restart the application. For detailed instructions on how to tune the JVM for optimal throughput, please refer to Tuning For Better Application Throughput

There is a certain trade-off between overall application throughput and low individual latencies. A JVM that is tuned for optimal overall throughput spends as little CPU time in garbage collection and memory management as possible to allow the Java application to run as much as possible. To minimize unnecessary overhead and extra work during garbage collection, the Java application should be paused for the duration of the entire garbage collection. This might cause long individual pauses; however, in the long run, it will maximize the overall throughput. You can reduce the latencies without losing too much overall throughput by, for example, limiting the

compaction or using a generational garbage collector. See Tuning For Low Latencies for tips on how to reduce the garbage collection pauses.

If All Else Fails, Open a Case With Oracle Support

If none of the tuning solutions suggested in Tuning For Better Application Throughput, resolve the problem, you will need to open a case with Oracle Support. You can find instructions on how to report a problem to Oracle, including the sort of information to include, in Submitting Problems to Oracle Support.

CHAPTER 29

The Oracle JRockit JVM's Performance Degrades Over Time

The Oracle JRockit JVM is designed to run in large server environments. Because of that, the JVM needs to provide constant, even performance throughout an application run. Occasionally, you might find that this performance begins to slip the longer the JVM runs. For example, your application works fine early in its run but, after a while it starts to run slower and show an unstable performance. This section provides information on how to recognize and address performance degradation.

The Problem is Usually With Tuning

When system performance begins to falter the problem is often with tuning. Incorrectly tuned compaction may for example cause the performance to degrade periodically as the fragmentation on the heap increases until the garbage collector must perform a full compaction in order to avoid throwing an OutOfMemoryError.

Before engaging in time-consuming troubleshooting and mitigation tasks, you should tune the JRockit JVM for stable performance and restart the application. For detailed instructions on how to tune the JVM for stable performance, please refer to Tuning For Stable Performance.

You Could be Experiencing Optimization Problems

Occasionally, when you encounter increasingly poor performance, you might be experiencing optimization problems. These problems tend to show up after the program has been running fine for a while and usually result in something like the following happening:

- The JVM crashes (see The System is Crashing for more information).
- `NullPointerExceptions` get thrown from unexpected points in the program.
- A method starts returning the wrong results.

Generally, optimization won't cause the sort of problems that you are experiencing with poor performance; however, before you report the issue, you can run the JVM with `-XnoOpt` enable to turn off code optimization just to eliminate this as a possible cause of the problem. If the application runs properly after turning off optimization, you can assume the problem was there. You should then follow the procedures for isolating and excluding the mis-optimizing method described in Exclude the Offending Method.

You Could Be Experiencing a Memory Leak in Java

A memory leak in Java causes the application to run slower and slower over time, as the garbage collector will have to work harder to free memory. In the end the JVM will throw an OutOfMemoryError, but applications with a small memory leak can sometimes run for days until that happens.

To look for initial signs of a memory leak you can do a JRA recording and check the heap usage after each old collection. If this memory usage keeps increasing you may be looking at a memory leak. If you are running the JRockit JVM R27.1 or higher with Oracle JRockit Mission Control 2.0 or later, complete instructions for creating and interpreting a JRA recording are available from the JRockit Mission Control built-in help. If you are using an earlier version of the Oracle JRockit JVM, please refer to Creating a JRA Recording with JRockit Mission Control 1.0 for these instructions.

You can diagnose memory leaks using the Memory Leak Detector, which will help you pinpoint the class that causes the memory leak. Complete instructions for using the Memory Leak Detector are available in the built-in help in Oracle JRockit Mission Control 2.0 and later releases.

If All Else Fails, Open a Case with Oracle Support

If none of the tuning solutions suggested in Tuning For Stable Performance resolve the problem, you will need to open a case with Oracle Support. You can find instructions on how to report a problem to Oracle, including the sort of information to include, in Submitting Problems to Oracle Support

CHAPTER 30

The System is Crashing

A Java application may stop running for several reasons. The most common reason is of course that the application finished running or was halted normally. Other reasons may be Java application errors, unhandled exceptions or irrecoverable Java errors like OutOfMemoryError. Occasionally you may encounter a JVM crash, which means that the JVM itself has encountered a problem from which it hasn't managed to recover gracefully. You can identify a JVM crash by the dump information that the Oracle JRockit JVM prints out in case of a crash.

This document describes how to diagnose and resolve JVM crashes. It includes information on the following subjects:

Notifying Oracle Support

Note that even if you do not have a Support contract with Oracle, you should notify Oracle Support if you have encountered a problem in the JRockit JVM. This way, Oracle can make sure

that the problem is fixed in the next release. For information on communication with Oracle Support, please refer to Submitting Problems to Oracle Support.

Classify the Crash

The first step in diagnosing and resolving a JVM crash is to classify the crash, i.e. trying to determine where and why the crash occurred.

Using a Crash File

Whenever the JRockit JVM crashes, it creates a snapshot of the state of the computer and the JVM process at the time of the crash and writes this information into one of these "crash files":

- **`dump` file**: The `dump` file is a text file that is like an executive summary of the full memory image and the environment in which the JVM was run at the time of the crash. This file is produced by the JVM itself when it crashes and is useful for classifying crashes; it can also sometimes be used to identify problems that have already been fixed. This file rarely reveals enough information to actually find the cause for the problem.
- **Core file**: The `core` file is a binary crash file, as described in Understanding Crash Files. By default this is a complete copy of the whole JVM process at the time of the crash. `Core` files are produced on Unix-like systems, such as Linux and Solaris. The file name of the `core` file is usually something like `<pid>.core`.
- **`mdmp` file** or a "minidump" file: This is the Windows version of the core file.

Binary crash files (`core` files and `mdmp` files) are very helpful to the Oracle support organization when solving JRockit JVM problems; however, if you don't have a service agreement with Oracle Systems, these files will not be of much help to you.

Determine the Crash Type

You can sometimes determine where and why the crash occurred by retrieving the text `dump` file and reviewing it for information that points to the crash type. Checking the size of the binary dump file may help in some cases, as well as checking the setup of the operating system. Table 30-1 lists symptoms you can look for and the probable crash types corresponding to these symptoms.

Table 30-1 Crash Symptoms and Crash Types

Symptoms	Probable Crash Type
The `dump` file indicates that the JVM process has run out of virtual memory. See Understanding Crash Files for details.	Out Of Virtual Memory Crash
The `core` file or `mdmp` file size is close to the maximum virtual memory size of the process on the OS. See Understanding Crash Files for details.	Out Of Virtual Memory Crash
The `dump` file indicates that stack overflow errors have occurred. See Understanding Crash Files for details.	Stack Overflow Crash
For Linux users only: The `dump` file indicates that `LD_ASSUME_KERNEL` is set. See Understanding Crash Files for details.	Unsupported Linux Configuration Crash
For Linux users only: You are using a non-standard or unsupported Linux configuration.	Unsupported Linux Configuration Crash
None of the above apply or help solve the problem.	JVM Crash

Out Of Virtual Memory Crash

The JVM reserves virtual memory for many purposes; for example the Java heap, Java methods, thread stacks and JVM internal data structures. In addition, native (JNI) code can also allocate memory. The process size consists of all the memory reserved by the JVM and is limited according to the operating system limitations. If the virtual memory allocation of the JVM process exceeds these limitations, the JVM will run out of virtual memory, which may cause it to crash. This section discusses the following topics:

- Verify the Out Of Virtual Memory Error
- Troubleshoot the Out Of Virtual Memory Error

Verify the Out Of Virtual Memory Error

Before you can start debugging an Out Of Virtual Memory Error, you should first verify that the error is indeed due to the JVM process running out of virtual memory. This section contains information on:

Virtual Memory Maximums

Table 30-2 shows the maximum virtual memory available to a single process on the various 32-bit operating systems. Virtual memory is practically unlimited on 64-bit platforms.

Table 30-2 Approximate Maximum Virtual Memory Available to IA32 Architectures

OS	Max Process Virtual Memory
Windows	2GB
Windows /3GB Startup Option	3GB
Linux (normally)	3GB

Checking the Text dump File

The text `dump` file, if such has been created by the JVM, may indicate that memory allocations have failed. See Understanding Crash Files for details. This is a strong indication that the JVM process has run out of virtual memory.

Checking the Binary mdmp or core File

When the JRockit JVM crashes, it generates a binary crash file. By default this file contains a copy of the entire JVM process. Check the size of this file to determine that the JVM process has indeed run out of virtual memory.

1. Verify that the binary crash file size has not been limited with the command line option `-XXdumpSize` or with the operating system command `ulimit` (Linux and Solaris only). Use the command `ulimit -a` to verify that the crash file size is unlimited on Linux and Solaris. If the size of the binary crash file has been limited, you can not use it to verify that the JVM process has run out of virtual memory.
2. Compare the size of the `mdmp` or `core` file with the size of the heap and ensure that it is larger than your heap size. This is a sanity check to verify that the binary crash file has not been truncated for example due to limited disk space.

3. Determine if the size of the `mdmp` or `core` file is close to the maximum process size allowed by the particular OS.

Troubleshoot the Out Of Virtual Memory Error

When you have verified that the JVM process has run out of virtual memory, you can start troubleshooting in order to fix the problem. This section covers the following topics:

- Upgrade to the Latest JRockit JVM Version Available
- Reduce the Java Heap Size
- Use the Windows /3GB Startup Option
- Check for Memory Leaks in JNI Code
- Record Virtual Memory Usage
- If All Else Fails, Open a Case with Oracle Support

Upgrade to the Latest JRockit JVM Version Available

Make sure you are running the latest available JRockit JVM version. There have been many fixes to address and reduce memory usage in the JVM over time. Using the latest JVM version for a specific major JDK will ensure that you are running the most memory efficient one.

Reduce the Java Heap Size

The Java heap is only a part of the JVM's total memory usage. If the Java heap is too large, the JVM may fail to start or run out of virtual memory when Java methods are compiled and optimized or native libraries are loaded. If this happens, you should try lowering the maximum heap size setting.

Use the Windows /3GB Startup Option

On Windows 2000 Advanced Server and Datacenter, Windows 2003 and Windows XP you have the option of starting the operating with the /3GB option by specifying so in `BOOT.INI`. This option changes the maximum virtual memory process size from 2GB to 3GB.

Check for Memory Leaks in JNI Code

Check any JNI code you are using for memory leaks. Incorrectly written or used JNI code may be leaking memory. This will grow the Java process until it reaches the maximum virtual memory size on the platform.

Record Virtual Memory Usage

Recording virtual memory usage shows memory usage growth, which will help Oracle Support identify and diagnose problems with running out of virtual memory. This section describes how you can collect virtual memory usage statistics on:

- Windows
- Linux

Windows

Use the Windows tool **perfmon** to record the PrivateBytes process counter. Collect information on the amount reserved virtual memory the JVM process. To do this:

1. Open Performance Monitor, which you can find in the Administrative tools.
2. Click **+** to open the Add Counters dialog box.
3. Open the Performance Object drop-down list and select **Process**.
4. Select the counter **Private Bytes** in the Process list.
5. Select the process that you want to monitor and click **Add**.

Linux

Create a script to record the virtual memory usage with a regular interval; for example:

```
top -b -n 10 > virtualmemory.log
```

This script will do "`top`" every ten seconds and put the data in a file called `virtualmemory.log`. The virtual memory usage for all running processes can be found in the VIRT column in that file. To see just the current status, type `top` and press `[Shift]-[M]` to sort the output by memory usage. This usually puts the JVM process(es) at the top of the output.

Creating a recording like `virtualmemory.log` can be useful as it allows you to see that the JRockit JVM process is actually growing and provide evidence to Oracle Support that the growth is there.

If All Else Fails, Open a Case with Oracle Support

If none of these solutions works, you will need to open a case with Oracle Support. Please refer to Submitting Problems to Oracle Support for details on what kind of information you need to provide and how to submit that information.

Stack Overflow Crash

A stack overflow crash occurs when the JRockit JVM cannot gracefully handle a stack overflow error. According to the J2SE Javadoc, a "gracefully" handled `java.lang.StackOverflowError` is a `java.lang.VirtualMachineError` thrown "to indicate that the JVM is broken or has run out of resources necessary for it to continue operating." For more information, please refer to these J2SE `java.lang` Javadocs:

- Java SE 6:
  - `Class StackOverflowError`
  - `Class VirtualMachineError`
- For J2SE 5.0:
  - `Class StackOverflowError`
  - `Class VirtualMachineError`
- For J2SE 1.4.2
  - `Class StackOverflowError`
  - `Class VirtualMachineError`

The JRockit JVM R26 (and higher) dump files includes information on the number of stack overflow errors thrown.

Verify the Stack Overflow Crash

A stack overflow crash is easy to identify: The text `dump` file says, `Error Message: Stack overflow` somewhere near the top of the file. Other indications might be an extremely long stack trace in the crash file or, paradoxically, no stack trace at all. If the dump file says something like `StackOverFlow: 2 StackOverFlowErrors occured`, this is an indication that the crash might be triggered by a previous stack overflow problem.

Troubleshoot a Stack Overflow Crash

This section describes some possible solutions to stack overflow errors.

Application Level Changes

Often, a stack overflow error is caused by the application being coded to require stack space that exceeds the JRockit JVM's memory limits. Examine the stack trace in the `.dump` file to determine if the Java code can be changed to use less stack space.

Increase the Default Stack Size

If changing the stack requirements of the application is not possible, you can change the thread stack size by using the `-Xss` option at JVM startup; for example:

`-Xss:<value>[k|m]`

Make the JRockit JVM More Robust Against Stack Overflow Errors

`-XcheckedStacks` makes the JRockit JVM more robust against stack overflow errors. It usually prevents the JVM from dumping and throwing a `java.lang.StackOverflowError`. There is a slight performance penalty when using this option as the JVM touches pages on the stack.

Unsupported Linux Configuration Crash

If your application crashes while running the JRockit JVM on Linux, even if the stack trace indicates a reason the crash occurred, you should ensure that you are running on a supported Linux configuration, as this might be contributing to the reason for the crash. You should do the following:

- Verify that the OS Version is Supported
- Verify that You Have Installed the Correct glibc Binary

Verify that the OS Version is Supported

The JRockit JVM is generally only supported on generally available products from OS vendors. The JRockit JVM does not support custom built kernels. To verify that your version of Linux is supported, please refer to the specific section for your version of the JVM in Oracle JRockit JDK *Supported Configurations*.

Verify that You Have Installed the Correct glibc Binary

Linux on IA32 must be configured to use the glibc compiled for i686 architecture, otherwise you will see hangs and crashes with the JRockit JVM.

You can check what glibc is installed by running:

```
rpm -q --queryformat '\n%{NAME} %{VERSION} %{RELEASE} %{ARCH}\n' glibc
```

If the output says something like “i386”; for example:

```
glibc 2.3.4 2.25 i386
```

you are using an unsupported glibc. You need to upgrade your glibc version to one that doesn’t say “i386”. Output from a supported system will say something like:

```
glibc 2.3.4 2.25 i686
```

Examine the Thread Library

If you have a core file in gdb, you can get a hint of what thread library you are using by running:

```
info shared
```

Look at the path of the loaded `libpthread<x>.so`.

If it is in `/lib/`, then you should ask for the result of the rpm command. If the output says something like “i386”, you are using an unsupported glibc; you need to upgrade your glibc version to one that doesn’t say “i386”.

JVM Crash

A JVM crash is caused by a programming error in the JRockit JVM. Identifying and troubleshooting a JVM crash can help you find a temporary workaround until the problem is solved in the JRockit JVM. It may also help Oracle Support to identify and fix the problem faster.

Code Generation Crash

This section describes how to identify and troubleshoot a code generation crash. It contains the following subjects:

- Identify a Code Generation Crash
- The Problem Might Lie With an External Instrumentation Tool
- If All Else Fails, Open a Support Case

Identify a Code Generation Crash

The most common cause for a code generation crash is a mis-compiled method. If the JRockit JVM mis-compiles a method, either the JVM will crash or the method will do something other than what the source code says. If the JVM crashes while generating code, the text `dump` file should identify which method was being compiled at the time of the crash. It should be identified on a line towards the top starting with `Method`.

Knowing which method was causing the problem is the first step in resolving the problem.

Troubleshoot the Code Generation Crash

If the JRockit JVM mis-compiles a method, the fault is likely to be with the JVM's optimizing compiler. To determine whether or not optimization itself is responsible, you can disable it by restarting the application with the `-XnoOpt` command-line option specified; for example:

```
java -XnoOpt myApp
```

If the JRockit JVM executes your program as expected, the problem is with code optimization.

Exclude the Offending Method

If disabling optimization stopped the JRockit JVM from crashing, you should next try excluding the offending method from optimization; you might be able run your application with almost full optimization if you can prevent just that method from being optimized. If this does not work, contact Oracle Support. Alternatively, try to use the `-XXpreOpt` command at startup to use the optimizing compiler for everything (be aware that using the optimizing compiler all the time can slow down the JVM startup).

You can exclude a method from optimization by using an *optfile*. If your application can run successfully without the offending method being optimized, this workaround should solve your problem.

Creating and Using an Optfile

An optfile is nothing more than a text file that contains *directives*, a single-character code that tells the optimizer that certain methods should either not be optimized or be forced optimized. Once you've created the file, you then use the `-Djrockit.optfile=<filename>` property (where `<filename>` is the optfile) to indicate the name and location of the optfile.

The structure of the file is illustrated in Listing 30-1.

Listing 30-1 Sample optfile

```
- java/lang/FloatingDecimal.dtoa
- java/lang/Object
- sun/awt/windows/WComponentPeer.set*
```

In Listing 30-1, the "-" at the beginning of each line tells the optimizing compiler to *never* optimize this method. Thus, in this example, the "-" directive tell the optimizer to never optimize the following:

- All methods called `dtoa` in the `FloatingDecimal` class.
- All methods in the `Object` class.
- All methods in the `sun.awt.windows.WComponentPeer` class beginning with `set`.

Note: If you are using a version of the JRockit JVM earlier than R26.4, using "-" will not disable regeneration of the method completely. If a method `m` is marked with a "-" and the hotspot detector thinks it is a hotspot, it will regenerate that method but not optimize it further.

"-" is the only useful directive with this workaround. The other directives are `h` (allow this method to be optimized by the hotspot detector, but do not preoptimize it), `p` (preoptimize this method and do not allow the hotspot detector to optimize it), and `+` (preoptimize this method and allow the hotspot detector to optimize it), however, they are not useful in this workaround.

Verifying optfile Response

If you want to make sure your optfile does as you expect, use `-Xverbose:opt` and check the output. You should not see the method you're excluding.

Setting and optfile with a Ctrl-Break Handler

You can also use a Ctrl-Break Handler to set the optfile. The handler is called `run_optfile` and takes a `<filename>` argument that is a regular optfile; for example:

`run_optfile optfile=<filename>`

When `ctrl-break` is pressed, any method matching the "-" directives in the optfile will not be optimized.

Rules for Directives

When you create an optfile, the following rules apply:

- Conflicting directives are applied from the top, the first match is used.
- Wildcards ("*") can be used last in a class or method name.
- If a directive is given without specifying a method or descriptor, the directive will apply to all methods and descriptors.

The Problem Might Lie With an External Instrumentation Tool

If you have eliminated a mis-compiled method as the problem for the crash and you are using an external instrumentation tool (for example JProbe or OptimizeIt), you might want to investigate whether this tool is causing the problem. These tools can alter bytecode, which can cause unexpected behavior. In some instances, the problem lies directly with the tool; however, the JRockit JVM might have issues with the tool that are causing the crash. To eliminate tools as a cause for the crash, disable the tool(s) and rerun the application. If the crash happens again, your problem is not with the instrumentation tool. If the application runs as expected, you should consider using a different tool or running without the tool.

If All Else Fails, Open a Support Case

If the optfile workaround doesn't alleviate the problem or if you cannot run the application successfully without the problematic method optimized, you will need to open a case with Oracle Support. You can find instructions on how to report a problem to Oracle, including the sort of information to include, in Submitting Problems to Oracle Support.

For code generation crashes, you will need to provide the following data to Oracle Support:

- The core (or `.mdmp`) file.
- The `.dump` file
- The class file containing the method that was being generated.
- The source code for the class containing the method that was being generated.

Garbage Collection Crash

This section describes how to identify and troubleshoot crashes in garbage collection. It contains the following information:

- Consider Upgrading to the Latest Version of the JRockit JVM
- Try One of These Workarounds
- If All Else Fails, Open a Case with Oracle Support

Identify a Garbage Collection Crash

You can identify a garbage collection crash by looking at the stack trace in the text `dump` file. If garbage collection functions appear in the stack trace, or if the thread that caused the crash is one of the garbage collection threads, the crash is most likely to have occurred during garbage collection. Garbage collection functions in the stack trace are identified by prefixes like `mm`, `gc`, `yc` and `oc`.

Consider Upgrading to the Latest Version of the JRockit JVM

If you are experiencing garbage collection crashes, the simplest—and most strongly-recommended—solution is to upgrade your version of the JRockit JVM to the latest one available. This is because further diagnosis of the problem can be a a very complex and time-consuming exercise. You can avoid the problem by upgrading because it might have been fixed in the latest version of the JVM.

Try One of These Workarounds

If you do not (or cannot) upgrade to the latest version of the JVM or if you are already using the latest version, try using any of the following workarounds to prevent garbage collection crashes:

- Change the Garbage Collector
- Disable Compaction
- Disable Inlining
- Use the Optimizing Compiler

Change the Garbage Collector

It is possible that the garbage collector you are using has bugs that you can avoid by changing to another garbage collector. Be aware though, that if you change collector, you will not receive the same performance profile from the Oracle JRockit JVM.

If you are using deterministic garbage collection, you cannot change to another garbage collector and retain the deterministic garbage collection guarantees. Instead of changing your garbage collector, you should open a case with Oracle Support.

- If you are having problems with the default dynamic garbage collector `-Xgcprio:throughput`, try switching to `-Xgc:parallel` or `-Xgc:genpar`.
- If you are having problems with the dynamic garbage collector `-Xgcprio:pausetime`, try switching to `-Xgc:gencon` or `-Xgc:singlecon`.
- If you are using one of the static garbage collectors, you might want to try a different one. For example, if you are using `-Xgc:singlecon` try switching to `-Xgc:gencon` or `-Xgc:singlepar`
  - Single-spaced concurrent (`-Xgc:singlecon`)
  - Single-spaced parallel (`-Xgc:singlepar` or `-Xgc:parallel`)
  - Generational concurrent (`-Xgc:gencon`)
  - Single-spaced generational (`-Xgc:genpar`)

  For more information on using static garbage collectors, please refer to Selecting a Static Garbage Collection Strategy.

Disable Compaction

Bugs in heap compaction can sometimes cause trouble leading to crashes in garbage collection. You can disable it by setting `-XXnoCompaction` at startup. Be aware that using this option can lead to heap fragmentation and should only be used for troubleshooting purposes. If the heap becomes too fragmented, you might encounter Out of Memory Errors.

Disable Inlining

Erroneous inlining may cause broken code, which makes the garbage collector lose track of live objects. You can disable inlining by using the command-line option combination `-XXnoJITInline -XnoOpt`. You must use both options because `-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.

Note: If `-XnoOpt` (without `-XXnoJITInline`) resolves the problem, your issue might be with code optimization. Conversely, if `-XXnoJITInline` without `-XnoOpt` resolves the problem, you should notify Oracle Support about this.

Use the Optimizing Compiler

You might be experiencing garbage collection crashes because the non-optimizing JIT compiler is generating broken code that makes the garbage collector lose track of live objects. Use the

`-XXpreOpt` command at startup to use the optimizing compiler for everything. Be aware that using the optimizing compiler can slow down the JVM startup.

If All Else Fails, Open a Case with Oracle Support

If none of the above workarounds resolve the crash issue, you will need to open a case with Oracle Support. You must include the following information:

- For crashes in garbage collection, you must include a complete `.dump` or `.mdmp` file, otherwise the support staff won't be able to resolve your issue. Verify that the core `.dump` or `.mdmp` file is at least as big as the Java heap.
- If you can a reproduce the crash, include the steps you used to do so.
- If you tried using another garbage collector, as described in Change the Garbage Collector, indicate if one garbage collector worked better than another or if crashes continued regardless of the collector used.
- Include information on any workaround you attempted.

For information on communication with Oracle Support, please refer to Submitting Problems to Oracle Support.

CHAPTER 31

Understanding Crash Files

Your JVM process has suddenly crashed, which has resulted in the Oracle JRockit JVM creating a snapshot, crash files, of the state of the computer and the JVM process at the time of the crash. These crash files are written to disk and are called a bit different depending on which platform you are using and the kind of information contained within the file (see Figure 31-1).

You can use some the information in the crash files to determine the nature of the problem that caused the JVM to crash. The information contained in the crash files is also essential for the support organization within the JRockit JVM to help solving problems with the JVM. When submitting problem reports to the Oracle support organization, you need to include these crash files (see Submitting Problems to Oracle Support for more information).

This section provides information on the differences between the crash files and how to enable and disable them. You will also find example crash files and diagnostics leads on how to interpret the information in the text crash file, which will enable you to identify solutions to how to either fix your Java application or the way you use and set up the JRockit JVM.

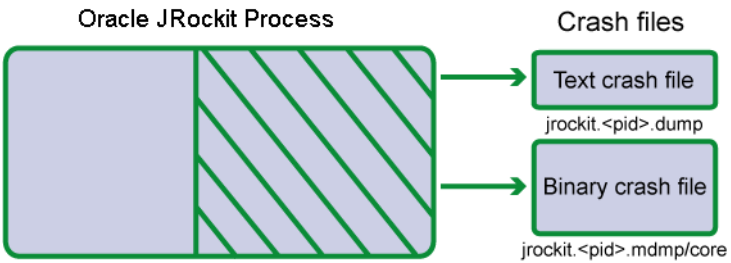
This section includes information on the following subjects:

- Differences Between Text dump Files and Binary core/mdmp Files
- Location of Crash Files
- Enabling Binary core Crash Files on Linux and Sun Solaris
- Enabling Binary mdmp Crash Files on Windows
- Binary Crash File Sizing

Differences Between Text dump Files and Binary core/mdmp Files

When the JRockit JVM crashes it creates two types of crash files (see Figure 31-1): a text crash file called *dump* and a binary crash file called *mdmp* (a Windows platform minidump) or *core* (a Linux and Sun platform core file). The format of the two types of crash files is: `jrockit.<pid>.dump` and `jrockit.<pid>.mdmp/core`, where *pid* is the process id that appears as a number, for example, `jrockit.72.dump`.

Figure 31-1 JRockit JVM creates two types of crash files



The information contained in the text `dump` file is information about the JVM at the point of time for the crash. This information can give hints and leads to why the JVM has crashed. See Listing 31-1 through Listing 31-4 for an example of the information contained in a text `dump` file. The text `dump` file can be viewed in a regular text editor. To turn the creation of the text `dump` file off, see Disabling Text dump Files.

The information in the binary `core` or `mdmp` crash file contains information about the entire JRockit JVM process and needs to be opened in a debugger. The size of the binary crash file is usually quite large, so you need to make sure there is enough disk space for the file to be completely written to disk. By default, the JVM records a full binary crash file. To set the size of the binary crash file, see Binary Crash File Sizing. To turn the creation of the binary crash file off, see Disabling the Binary Crash Files.

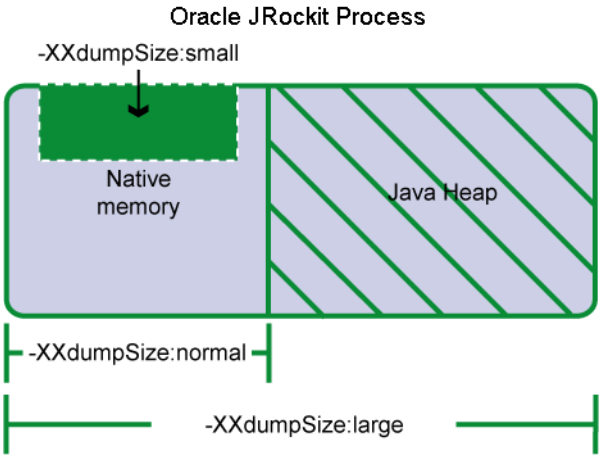
Binary Crash File Sizing

The size of the binary `core` or `mdmp` crash files vary tremendously. The text `dump` file is normally a very small file, usually under 100kB. The binary crash files, on the other hand, are, by default, set to log the entire JRockit JVM process (see Figure 31-2) and therefore become very large, which demands greater storage capacity on your system.

You can se the crash file size using the command line option `-XXdumpSize`. The default setting is `-XXdumpSize:large`. A large dump contains the entire JVM process including the Java heap. While `small` and `normal` settings are also available, neither of these sizes are adequate for troubleshooting a JVM crash since the Java heap is excluded.

If you don't want a binary crash file, you can set `-XXdumpSize:none`. For more information on this option, see `-XXdumpSize` in the Oracle JRockit JVM *Command-Line Reference*.

Figure 31-2 Difference in information saturation between small, normal, and large binary crash files



Note: Remember that you need to provide a `<large>` binary `core` or `mdmp` crash file with any problem that you submit to the Oracle JRockit support organization.

If there is not enough memory on the disk where the crash file is written for the entire binary crash file, the file will become as large as possible, i.e. fill up the disk.

Location of Crash Files

Both the binary crash files and the text `dump` files are saved to your current working directory (see Enabling Binary core Crash Files on Linux and Sun Solaris). If you want to save the crash files

in a different location, use the environment variable JROCKIT_DUMP_PATH and specify where you want the information located. The path that you specify must exist and be writable.

Enabling Binary core Crash Files on Linux and Sun Solaris

For the Linux and Sun Solaris systems, you need to set `ulimit -c <value>` to something greater than zero, for example, `ulimit -c unlimited`. This value is measured in blocks, with each block equaling one kilobyte. You can set the value from either the command line or in a shell script. To disable crash files on Linux and Solaris, set `ulimit -c 0`.

To specify the where the JRockit JVM should put the binary `core` crash file, enter the following:

```
export JROCKIT_DUMP_PATH=<path to directory>
```

Enabling Binary mdmp Crash Files on Windows

Binary `mdmp` files are always created on Windows. If you do not specify the variable `JROCKIT_DUMP_PATH`, the crash files end up in the current directory (the directory from where you start the JVM).

To specify the where JRockit JVM should put the crash file, enter the following:

```
set JROCKIT_DUMP_PATH=<path to directory>
```

Disabling Crash Files

The crash files are enabled by default so that you will be sure to get as much information as possible about your application and the JRockit JVM process as possible in the event of a crash; however, there might be times when you do not want the JVM to create any type of crash file, for example, if you have limited disk space.

Before turning the creation of the crash files off, please remember that the text `dump` file is a small and good source of information to get an initial grasp of what has gone wrong with the JVM and the binary `core` or `mdmp` crash file is essential when creating a support case for the Oracle JRockit support organization.

Disabling Text dump Files

If you suspect problems with the creation of text `dump` files you can turn off the text `dump` file by using the option: `-XXnoJrDump`.

Disabling the Binary Crash Files

You can turn off the binary crash file by using the option: `-XXdumpSize:none`.

Extracting Information From a Text dump File

A JVM crash means that something has happened that the JVM couldn't handle gracefully. The cause can be a programming error in the JVM, but a crash may also indicate a problem in the Java code or in the JVM setup. The `dump` file is a good source of initial troubleshooting information to help you identify the problem.

Symptoms to Look For

A `dump` file will not necessarily tell you exactly why the crash occurred, but it describes the environment where the JRockit JVM was running and the state of the JVM when the crash occurred. This gives some hints on where to start looking and can give an idea of the crash type. There are however some easily identified symptoms that you can look for in the `dump` file:

1. Check the `StackOverFlow` field in the `dump` file. If this field indicates that stack overflow errors have occurred, the crash is likely to have been caused by a stack overflow. See Stack Overflow Crash for information on how to deal with stack overflow crashes.
2. Check for reports of `Stack Overflow` in the `Error message` field or in the stack trace near the bottom of the `dump` file. Such occurrences may also indicate a stack overflow. See Stack Overflow Crash for information on how to deal with stack overflow crashes.
3. Check the `C Heap` field in the `dump` file. If this field indicates that memory allocations have failed, the process may have run out of virtual memory. See Out Of Virtual Memory Crash for information on how to deal with out of virtual memory errors.
4. Check the `LD_ASSUME_KERNEL` field in the `dump` file. Having this Linux-specific environment variable set usually leads to the JRockit JVM using unsupported system libraries and becoming unstable. See Unsupported Linux Configuration Crash for information on dealing with unsupported Linux configuration problems.

If none of these symptoms apply to your `dump` file you can try troubleshooting further using the information in JVM Crash or open a support case. To open a support case you must have a service agreement with Oracle.

Example of a Text dump File

This section shows an example of what a text dump file might look like and explains the parts of the dump file. The text dump file that has been used in this example is a combination of many different dump files, so that more crash scenarios can be covered.

Remember that the text crash files are not full descriptions of what has happened in the crash and the layout of the text crash file the JVM produces might differ from the examples given here.

The Beginning of the Text dump File

Listing 31-1 shows an example of the first part of a dump file.

Listing 31-1 The initial information of a text dump file

JRockit JVM Dump (Crash File) Produced	`===== BEGIN DUMP ====================================`
	
	`JRockit dump produced after 0 days, 22:10:26 on Thu Jul 27 10:57:54 2006`
	
	`Additional information is available in:`
	`/usr/bea/user_projects/domains/jal/jrockit.17727.dump`
	`/usr/bea/user_projects/domains/jal/core or core.17727`

Support Case Information	`If you see this dump, please open a support case with Oracle and supply as much information as you can on your setup and the program you were running. You can also search for solutions to your problem at http://forum/Oracle in the forum jrockit.developer.interest.general.`
Error Message from OS	`Error Message: Illegal memory access. [54]` `Signal info : si_signo=11, si_code=2 si_addr=0x20`

JRockit JVM Version and Garbage Collector Information	`Version : BEA JRockit(R) R26.4.0-63-63688-1.4.2_11-20060626-2259-linux-ia64`
	`GC : gencon : mmHeap->data = 0x2000000000ae0000, mmHeap->top = 0x20000000bc2e0000 :`
	`The nurserylist starts at 0x200000009fe9e610 and ends at 0x20000000a62bb358 : mmStartCompaction = 0x2000000000ae0000, mmEndCompaction = 0x200000000c660000`

CPU and Memory Information	`CPU : Intel Itanium 2`
	`Number CPUs : 1`
	`Tot Phys Mem : 8502312960 (8108 MB)`

Operating System Version Information	`OS version : Red Hat Enterprise Linux ES release 4 (Nahant Update 3) Linux version 2.6.9-34.EL (bhcompile@altix2.build.redhat.com) (gcc version 3.4.5 20051201 (Red Hat 3.4.5-2)) #1 SMP Fri Feb 24 16:49:08 EST 2006 (ia64)`
Thread and State Information	`Thread : NPTL` `State : JVM is running`

JRockit JVM Dump (Crash File) Produced

Listing 31-1 shows the start of the text `dump` file. This section contains information about when the crash occurred and for how long the JVM has been running. The file locations refer to the location of both text and binary crash files.

Support Case Information

If you have a service agreement with Oracle, you can file a support case with your Oracle service provider. If you do not have a service agreement, you can post your problem on the Oracle JRockit JDK developer forum under `jrockit.developer.interest.general`. Posting your problem on the forum is no guarantee to get it resolved, but the forum is a good source for information on JRockit JVM problems.

Error Message from OS

The information block that starts with *Error Message*, shows the actual error message that the operation threw at the time of the crash. Check your operating system vendor users information to find out more about the error.

JRockit JVM Version and Garbage Collector Information

Use the *Version* description to see if you are running the latest version of the JRockit JVM, see Oracle JRockit *Supported Configurations* for a complete list of supported versions.

GC stands for *Garbage Collector* and this description states which garbage collector that has been used. In this example, the generational concurrent (gencon) garbage collector has been used. Since the generational garbage collector has been used, there is a listing of where in the memory the nursery starts and ends. The `mmStartCompaction` and `mmEndCompaction` values are a description of where in the heap the compaction of the heap has taken place during the current or latest garbage collection.

For more information on memory management in the JRockit JVM, see Understanding Memory Management.

CPU and Memory Information

The information block that starts with *CPU*, describes how many CPUs that have been used and how much memory (*Tot Phys Mem*) that has been consumed by the Java process, application, or the JRockit JVM.

Operating System Version Information

The *OS version* states the operating system version that you are running on. Please make sure that you are running on a supported operating system version, see Oracle JRockit JDK *Supported Configurations*.

Thread and State Information

The *Thread* field indicates the thread system that the JRockit JVM used at the time of the crash. In this example the JVM used the Native POSIX Thread Library (NPTL).

The *State* of the JVM at the time of the crash was that it was *running*. Other valid states could be *starting up* or *shutting down*.

Command Line and Environment Information

The second part of the example, see Listing 31-2, starts with the text `Command Line`.

Listing 31-2 Looking at information about command line options in a text crash file

Command Line Option Information	`Command Line : -Xms256m -Xmx256m -Xns:128m -Xgc:gencon -Dweblogic.Name=AdminServer -Dweblogic.ProductionModeEnabled=true -Djava.security.policy=/product/bea815/weblogic81/server/lib/weblogic.policy weblogic.Server` `java.home : /product/download/jrockit-142_11/jre` `j.class.path : /product/download/jrockit-142_11/lib/tools.jar:/product/bea815/weblogic81/server/lib/weblogic_sp.jar:/product/bea815/weblogic81/server/lib/weblogic.jar::/product/bea815/weblogic81/common/eval/pointbase/lib/pbserver44.jar:/product/bea815/weblogic81/common/eval/pointbase/lib/pbclient44.jar:/product/download/jrockit-142_11/jre/lib/rt.jar:/product/bea815/weblogic81/server/lib/webservices.jar::/product/lib/db2java.zip:/product/lib/db2jcc.jar:/product/lib/db2jcc_license_cu.jar` `j.lib.path : /product/download/jrockit-142_11/jre/lib/i386/jrockit:/product/download/jrockit-142_11/jre/lib/i386:/product/download/jrockit-142_11/jre/../lib/i386:/product/bea815/weblogic81/server/lib/linux/i686:/product/bea815/weblogic81/server/lib/linux/i686/oci920_8`
JAVA_HOME and _JAVA_OPTIONS	`JAVA_HOME : /product/download/jrockit-142_11` `_JAVA_OPTIONS: <not set>`
LD_LIBRARY_PATH	`LD_LIBRARY_PATH: /product/download/jrockit-142_11/jre/lib/i386/jrockit:/product/download/jrockit-142_11/jre/lib/i386:/product/download/jrockit-142_11/jre/../lib/i386:/product/bea815/weblogic81/server/lib/linux/i686:/product/bea815/weblogic81/server/lib/linux/i686/oci920_8`

LD_ASSUME_KERNEL, C Heap, StackOverFlow, and OutOfMemory	`LD_ASSUME_KERNEL: <not set>`
	`C Heap : Good; no memory allocations have failed`
	`StackOverFlow: 0 StackOverFlowErrors have occured`
	`OutOfMemory : 0 OutOfMemoryErrors have occured`

Command Line Option Information

Command Line lists all startup options that were sent to the JRockit JVM at startup. The example in Listing 31-2 has used command line options for setting a generational concurrent garbage collector (`-Xgc:gencon`), with an initial minimum and maximum Java heap (`-Xms` and `-Xmx`), and a nursery size setting (`-Xns`).

JAVA_HOME and _JAVA_OPTIONS

`JAVA_HOME` is the path to your Java home catalog; that is, where the JRockit JVM is installed. `_JAVA_OPTIONS` is a list of command line options that will be automatically passed to all newly started JRockit JVMs.

LD_LIBRARY_PATH

`LD_LIBRARY_PATH` is a Linux /Solaris specific environment variable that can make the JRockit JVM find libraries other than the default system libraries. Sometimes, you need to set this variable for running JNI code. You can set this variable so that the JVM starts using unsupported libraries on otherwise supported platforms.

LD_ASSUME_KERNEL, C Heap, StackOverFlow, and OutOfMemory Information

The information block that starts with `LD_ASSUME_KERNEL` lists information of what might have gone wrong prior to the crash. See Symptoms to Look For for more information on what to look for in these fields.

Registers and Stack Information

The third part of the example, see Listing 31-3, starts with the text *`Registers`*.

Listing 31-3 Verifying that the text crash file is correct.

Registers

```
Registers (from context struct at 0x80d77ac/0x80d7874):

EAX = 00000009  EBX = b683c774
ECX = 226fad50  EDX = 226fad50
ESI = 226fad50  EDI = 00000000
ESP = b683c658  EIP = b73188b8
EBP = b683c658  EFL = 00010283
```

Stack Information

```
Stack:
b683c658 :b683c678 b73188f2 226fad50 0000000d b683c698
b7312383
b683c670 :209e2468 b3070c4c b683c6a8 b7312383 226fad50
b3070c4c
b683c688 :b3070c90 b3070c48 b683c774 b3070c7c b683c6c8
b732166e
b683c6a0 :09548670 b683c774 b683c6d8 b7312461 b683c774
b3070c7c
b683c6b8 :b683c770 091c7278 b683c774 b683c774 b683c770
b732173b
b683c6d0 :b3070c48 b683c774 b683c708 b73124a0 b683c774
b683c774

-----------------------------------
```

Registers

The Registers section is only useful to Oracle Support personnel troubleshooting your issues. You can ensure that the text crash file is not corrupt if the register ESP and the first number of the stack match (Listing 31-3). If these two numbers do not match, you can suspect that the text `dump` file itself is incorrect.

Stack Information

If the stack information says "`unreadable`" instead of showing numbers, the crash is probably due to stack overflow. The Stack information section is usually much longer than the one shown in Listing 31-3.

Stack Trace Information

The information block that starts with *Stack*, see Listing 31-4, describes what has happened and where it happened in the thread stack at the point of the crash.

The *Thread Stack Trace* shows what the crashing thread was doing when the JRockit JVM crashed; for example, Listing 31-4 shows that a crash occurred during code generation.

Listing 31-4 Stack trace in the text crash file

```
Stack 0: start=0x108c64000, end=0x108cc4000, guards=0x108cb0000 (ok),
forbidden=0x108ca8000 Stack 1: start=0x108c44000, end=0x108c64000,
guards=0x108c58000 (ok), forbidden=0x108c60000

Thread Stack Trace:
at get_constant_alen+576()@0x2000000000631110<!-- this is the crashing point
at get_constant_alen+256()@0x20000000000630fd0
at alength_opt+96()@0x2000000000631300
at optStrengthReduction+320()@0x2000000000631c80
at optmanOptimizeMIR+656()@0x20000000005efbf0
at generateMethodWithStage+256()@0x20000000004a2920
at cmgrGenerateMethodFromPhase+320()@0x20000000004a2a70
at cmgrGenerateNormalMethod+240()@0x20000000004a27b0
at cmgrGenerateCode+672()@0x20000000004a2310
at generate_code2+896()@0x2000000000702830
at codegenThread+1312()@0x20000000007031c0
at tsiCallStartFunction+48()@0x20000000007b3640
at tsiThreadStub+336()@0x20000000007b53d0
at ptiThreadStub+80()@0x20000000007995c0
at start_thread+352()@0x20000000001317f0
at __clone2+208()@0x20000000002fb9f0
-- Java stack --

Additional information is available in:
       /usr/bea/user_projects/domains/jal/jrockit.17727.dump
       /usr/bea/user_projects/domains/jal/core or core.17727

If you see this dump, please open a support case with Oracle and supply as much
information as you can on your system setup and the program you were running.
You can also search for solutions to your problem at http://forums.bea.com in
the forum jrockit.developer.interest.general.

Extended, platform specific info:
libc release: 2.3.4-stable
Elf headers:
libc          ehdrs: EI:    7f454c4601010100000000000000000 ET: 3 EM: 3 V: 1
ENTRY: 004e8f20 PHOFF: 00000034 SHOFF: 001346c4 EF: 0x0 HS: 52 PS: 32 PHN; 10
SS: 40 SHN: 70 STIDX: 67
libpthread    ehdrs: EI:    7f454c4601010100000000000000000 ET: 3 EM: 3 V: 1
ENTRY: 00704840 PHOFF: 00000034 SHOFF: 000107c4 EF: 0x0 HS: 52 PS: 32 PHN; 9 SS:
40 SHN: 39 STIDX: 36
libjvm        ehdrs: EI:    7f454c4601010100000000000000000 ET: 3 EM: 3 V: 1
```

```
ENTRY: 0004ff30 PHOFF: 00000034 SHOFF: 0029d7c4 EF: 0x0 HS: 52 PS: 32 PHN; 3 SS:
40 SHN: 20 STIDX: 17

===== END DUMP ==========================================================
```

CHAPTER 32

The Oracle JRockit JVM is Freezing

When the Oracle JRockit JVM or Java application becomes unresponsive but hasn't crashed, it is considered to be frozen. A frozen system occurs when the application stops answering requests but the process is still there. A system can freeze in either the JVM or the application. Your first action is to determine where it is freezing.

- If you can get thread dumps from the JVM then the JVM is functional and the freeze is in the Java application. See Java Application Freeze
- If you cannot get thread dumps by using the procedures described below, then it is probably freezing in the JVM, as it has stopped handling signals. See JVM Freeze.

Most often, a JVM freeze requires that you contact Oracle JRockit Support for resolution. This section describes how to diagnose where a system is freezing and how to collect information that will assist the support team resolve your problem.

This section includes information on the following subjects:

- Java Application Freeze
- JVM Freeze

Diagnosing Where the Freeze is Occurring

To determine whether the freeze is occurring in the application or in the JVM, try to generate a thread dump by doing one of the following:

- **On Windows**, press Ctrl-Break

- **On Linux and Solaris**, send `SIGQUIT` (`kill -3`) to the parent Java Process ID.

On all platforms you can also get the thread dump by using `jrcmd`; for example:

```
jrcmd nnnn "print_threads nativestack=true"
```

where *nnnn* is the PID of the Java process. To get a list of the PIDs of all Java processes running on the machine, run `jrcmd` without any command line parameters.

The result of the any of these procedures will indicate the kind of freeze happening:

- If the system responds with a Java thread dump, then the system is freezing in the application. To resolve this type of freeze, see Java Application Freeze.
- If you cannot force a thread dump by using one of the above procedures, your application is most likely freezing in the JVM. Please go to JVM Freeze for information on handling this sort of freeze.

Java Application Freeze

An application freeze occurs when something in the Java application causes the system to become unresponsive. This section describes how to handle a Java application freeze. It contains information on the following subjects:

- Resolving a Java Application freeze
- If This Did Not Help

Resolving a Java Application freeze

Here are two ways you can work around a Java application freeze.

- Use the thread dumps to view locks and deadlocks. If you are running the Oracle JRockit JVM 1.4.2 or later, look at the bottom of the thread dump and see if you have lock chains that could be the cause of the freeze.
- Review the thread dumps to see if you can determine the cause of the freeze. If it is not something you can easily fix from your end, you will have to submit a trouble report against the offending application component.

If This Did Not Help

If none of the suggested workarounds corrects the situation, you will need to open a case with Oracle Support, according to the process described in Submitting Problems to Oracle Support.

When creating a support case for a Java application freeze, you need to include the following information:

- Three thread dumps from the application when it works fine.
- Three thread dumps from the application when it has frozen.
- One 120s JRA recording from the application when it works fine (see Creating a JRA Recording with JRockit Mission Control 1.0)
- One 120s JRA recording from the application when it has frozen.

JVM Freeze

If you cannot get thread dumps with Ctrl-Break or by sending `SIGQUIT (kill -3)` to the parent Java process PID after a few tries, the JVM has stopped handling signals and is really freezing. When this happens, you should open a case with Oracle Support and work with them to rectify the problem. This section describes the ways you can force the JRockit JVM to crash in order to create the necessary crash file. It contains information on the following subjects:

- Collect Information About the JVM Freeze
- Submit the Information to Oracle JRockit Support
- Non-responding NFS Shares

Collect Information About the JVM Freeze

Since resolving a JVM freeze requires that you open a case with Oracle Support, you need to collect as much information as you can about the state of the process that was running when the JVM froze. Normally, this information would be written to a crash file, but since the JRockit JVM technically hasn't crashed, that file won't be created unless you force the JVM to crash. This section explains how to collect the necessary state information based upon your JVM deployment.

- Collecting Information on Linux Systems
- Collecting Information on Windows Systems
- Collecting State Information if the JRockit JVM is Running as a Service

Collecting Information on Linux Systems

To collect information for an JRockit JVM running on Linux, use one of these procedures:

- Force a Crash with the Diagnostics Command enableforce_crash
- Use SIGABRT (modern Linux Systems)

Force a Crash with the Diagnostics Command enableforce_crash

On versions of the JRockit JVM later than R25, you can use the command `enableforce_crash` to force a crash and thus produce a crash file with the necessary state information. Add the following option to the command line and use `jrcmd` to force a crash:

```
-Djrockit.ctrlbreak.enableforce_crash=true
```

Use SIGABRT (modern Linux Systems)

`SIGABRT` aborts the signal sent to the process and causes a crash, thus producing a crash file. Invoke `SIGABRT` as described here:

1. Find the PID of the JRockit JVM process by using the following procedure:

   Note: The following instructions are valid on 2.6 kernel-based Linux systems and on Red Hat Enterprise Linux 3.0.

   a. Assuming that the JRockit JVM process runs as the Linux user webadmin, enter the following command:

      ```
      pstree -p webadmin | grep java
      ```

   b. If you get excessive garbage from `pstree`, try unsetting the `LANG` environment variable first:

      ```
      unset LANG
      ```

   c. If pstree gives you only one Java process, you're done. Go to step 2.

   d. If `pstree` gives you several processes, print the command line parameters for any process by entering:

      ```
      cat /proc/nnnn/cmdline | xargs --null -n1 echo
      ```

      (where *nnnn* is the PID you are interested in; for example, *1234*)

2. Use the command:

   ```
   ls -l /proc/1234/cwd
   ```

to see where the `core` file (the core dump crash file) will be created (assuming that the PID is *1234*).

3. Create the core file (and terminate the process) by entering:

   ```
   ulimit -c unlimited
   kill -SIGABRT 1234
   ```

   (again assuming that the PID is *1234*).

Collecting Information on Windows Systems

To collect information for JRockit JVM running on Windows, use one of these procedures:

- Force a Crash with the Diagnostic Command enableforce_crash
- Use windbg

Force a Crash with the Diagnostic Command enableforce_crash

On versions of JRockit JVM later than R25, you can use the diagnostic command `enableforce_crash` to force a crash and thus produce a crash file with the necessary state information. Add the following option to the command line and use `jrcmd` to force a crash:

```
-Djrockit.ctrlbreak.enableforce_crash=true
```

Use windbg

The `windbg` command can also create the necessary core file. Enter the command:

```
windbg.exe -Q -pd -p nnnn -c ".dump /u /ma hung.mdmp; q"
```

(where *nnnn* is the PID)

Note: `windbg` is included in the Debugging Tools for Windows package that you can download from:

`http://www.microsoft.com/whdc/devtools/debugging/default.mspx`

Collecting State Information if the JRockit JVM is Running as a Service

If the JRockit JVM starts as a service, you can collect thread dumps by doing one of the following:

- If you are collecting information from your own machine, run `jrcmd` with the `print_threads` Ctrl-Break handler. See Using jrcmd for background on `jrcmd` and Available Diagnostic Commands for information on using `print_threads`.

- If you are collecting information from another machine, use Oracle JRockit Mission Control with the diagnostics bean.
- If you are running the JRockit JVM with Oracle WebLogic Server, you might want to use `beasvc –dump` to obtain thread-dumps from the JVM. For more information, please refer to Setting Up a WebLogic Server Instance as a Windows Service.
  - If you want to see the messages that a server instance prints to standard out and standard error (including stack traces and thread dumps), redirect standard out and standard error to a file, please refer to Setting Up a WebLogic Server Instance as a Windows Service.
  - To cause the WebLogic Server instance to print a thread dump to standard out, do either of the following:
    - Use the `weblogic.Admin THREAD_DUMP` command.

    or

    - Open a command prompt and enter the following command:

    `WL_HOME``\bin\beasvc -dump -svcname:`*`service-name`*

    where *`WL_HOME`* is the directory in which you installed WebLogic Server and *`service-name`* is the Windows service that is running a server instance; for example.:

    ```
    D:\bea\weblogic81\server\bin\beasvc -dump -svcname:mydomain_myserver
    ```

Submit the Information to Oracle JRockit Support

Open a case with Oracle Support, who will continue to troubleshoot your JVM freeze until it is resolved. Be sure to include any information pertinent to the freeze, including the all thread dumps you've collected. For information on communication with Oracle Support, please refer to Submitting Problems to Oracle Support.

Non-responding NFS Shares

Note: This information is for Linux users, only

In some cases, the freeze might be caused by a problem with the Network File System (NFS). NFS is a protocol used by Linux computers to share disks across a network. An *NFS share* is any disk on NFS set up to be shared by other computers. If you are using a machine that has NFS shares (look for `nfs` in `/etc/fstab`), your application might be freezing because a Java application is trying to access data on a non-responding NFS share. When that happens, the default behavior for the thread doing the file access is not to respond to any signals until the NFS

server comes back up. This means that the Oracle JRockit JVM cannot suspend the thread, and it might freeze trying to suspend a thread until NFS services are restored.

Verify that your NFS server is configured correctly if you're having NFS problems.

CHAPTER 33

Submitting Problems to Oracle Support

Note: This chapter applies to customers who have a service level agreement with Oracle Systems that includes the Oracle JRockit JDK. If you do not have one of those agreements, you will not be able to open a case directly with Oracle. If you do have an agreement, you need to go through your normal channels to get in contact with your service agreement partner.

This chapter provides information on how which steps to take before submitting a trouble report. You will also find information on how to help Oracle improve the Oracle JRockit JVM and its performance. To get a quicker response to your problems, you must try all suitable diagnostics that are provided in this document and you must also provide Oracle with as much information about your problem as possible.

This chapter contains information on the following subjects

Check the Oracle JRockit JVM Forums First

Occasionally, you won't be the only JRockit JVM user experiencing the specific problem. Oracle's vast user community provides mutual support to other users through various Oracle JRockit JVM forums. You can post questions you might have about an issue to these forums and participate in discussions about how to best use the JRockit JVM. To find a list of Oracle JRockit JVM user forums, please see the Oracle JRockit News Group jrockit.developer.interest.general at Dev2Dev Oracle JRockit Newsgroups page:

http://forums.bea.com/bea/category.jspa?categoryID=2010

Filing the Trouble Report

If you determine that you need to file a trouble report, this section discusses what you need to do before opening the case to ensure that you supply the support personnel assigned to you issue as complete picture of what is wrong as possible. The more information you can provide, the more quickly will the support staff be able to resolve your issue.

Trouble Reporting Process Overview

When you encounter a problem with the JRockit JVM and can't resolve it using the information provided in JRockit JDK Tools, you need to collect the information indicated in the chapter of that section that best describes your problem and open a case with Oracle Support. If you have a service agreement with Oracle, the normal process is to contact your Front Line service provider, who will make the initial attempts to correct the problem. If the case cannot be solved by the Front Line staff, it is escalated to the Back Line staff, who will draw on their particular expertise to get your JVM running again. For serious problems, the issue will be handled by the Engineering staff (the JRockit JVM developers) which is located in Stockholm, Sweden (GMT + 1 hour and uses Daylight Savings Time).

Identify Your Problem Type

Is you JVM process crashing? Is it running slowly or returning unpredictable results? These are the kind of symptoms that indicate a problem with the JRockit JVM. Being able to identify what kind of problem you are experiencing will help you know what kind of information you need to include when you open the trouble report. The chapters in JRockit JDK Tools are organized by problem type, or "symptom," and list the sort of information you need to include with a trouble report.

Verify That You're Running a Supported Configuration

Before submitting a bug, verify that the environment where the problem arises is a supported configuration. Please see Oracle JRockit JDK *Supported Configurations* at:

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

You should also verify that you are running the latest service pack or patch levels for your respective operating system.

Verify the Problem has Not Been Fixed in a Subsequent Version of the JRockit JVM

When a new update of the JRockit JVM becomes available, it becomes the default and many problems with earlier versions might have been resolved. Therefore, you need to verify that the problem you've encountered doesn't exist on the latest version. Please look for the latest download of the Oracle JRockit JVM at the following location:

http://commerce.bea.com/products/weblogicjrockit/jrockit_prod_fam.jsp

The download site includes links to the relevant documentation for the release you are looking for. Please look at the release notes to find lists of the bug fixes in the release. If you suspect a bug, then you should, as an early step in the diagnosis, check the list of fixes that are available in the most recent update release. Sometimes it is not obvious if an issue is a duplicate of a bug that is already fixed, so where possible, you should test with the latest update release to see if the problem persists.

Collect Enough Information to Define Your Issue

In addition to testing with the latest update release, use the following guidelines to prepare for submitting a trouble report:

- Collect as much relevant data as possible. For example
  - If a deadlock occurs, generate a thread-dump.
  - In a crash occurs, locate the core file (where applicable) and the appropriate error file.

  In all cases you must document the environment and the actions performed just before the problem was encountered.
- Where applicable, try to restore the original state and reproduce the problem using the documented steps. This helps to determine if the problem is reproducible or an intermittent issue.
- If the issue is reproducible, try to narrow the problem. In some cases, a bug can be demonstrated with a small standalone test case. Bugs demonstrated by small test cases will typically be easy to diagnose when compared to test cases that consists of a large complex application.