

BEAJRockit Mission Control™®

BEA JRockit Runtime Analyzer

Mission Control version 2.1® Document Revised: April, 2007

Copyright

Copyright © 1995-2007 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software is protected by copyright, and may be protected by patent laws. No copying or other use of this software is permitted unless you have entered into a license agreement with BEA authorizing such use. This document is protected by copyright and may not be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form, in whole or in part, without prior consent, in writing, from BEA Systems, Inc.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA SYSTEMS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE DOCUMENT IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks and Service Marks

Copyright © 1995-2005 BEA Systems, Inc. All Rights Reserved. BEA, BEA JRockit, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Communications Platform, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Enterprise, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic Log Central, BEA WebLogic Platform, BEA WebLogic Portlets for Groupware Integration, BEA WebLogic Server Process Edition, BEA WebLogic SIP Server, BEA WebLogic WorkGroup Edition, Dev2Dev, Liquid Computing, and Think Liquid are trademarks of BEA Systems, Inc. BEA Mission Critical Support, BEA Mission Critical Support Continuum, and BEA SOA Self Assessment are service marks of BEA Systems, Inc.

All other names and marks are property of their respective owners.

Contents

Welcome to the BEA JRockit Runtime Analyzer (JRA)

How Does the JRA System Work?1	l-1
What is a JRA recording? 1	1-2
What is the JRA tool?	1-2

Using the JRA Tool

Getting Started with the BEA JRockit Runtime Analyzer Tool

Creating a Recording
About JRA Overhead when Recording
Opening a JRA File
Comparing and Contrasting JRA Recordings
Customizing Your JRA Tool
Changing Table Settings
Filtering Information
Collapsing and Expanding a View
Changing View of a Tab

General Information in JRA Recording

Getting Familiar with the General Tab	4-18
Viewing General Information	4-19
Viewing Memory Usage Information	4-20
Viewing VM Arguments Information.	4-20

Viewing Memory Allocation Information	4-21
Viewing Threads Information	4-22
Viewing Exceptions Information	4-22

Methods and Call Trace Information

Getting Familiar with the Methods Tab	5-23
Viewing Hot Methods	5-24
Viewing Predecessors and Successors	5-25

Garbage Collection Events Information

Getting Familiar with the GC Events Tab
Changing Focus on Heap Usage Chart
Viewing Specifics about Garbage Collections
Viewing the Detailed Information About the Garbage Collection 6-31
Viewing Information on the General Garbage Collection Tab
Viewing Information on the GC Method Call Tree Tab
Viewing Information on the Old/Young Collection Tab
Viewing Information on the Cache Lists Tab (Only valid for old collections) 6-34
The Pause Time Tab

General Garbage Collector Information

Getting Familiar with the GC General Tab	7-37
Viewing General Garbage Collection Information	7-38
Viewing Garbage Collection Call Tree Information	7-39
Viewing Garbage Collection Strategy Changes Information	7-39

Java Heap Content Information

Getting Familiar with the Heap Overview Tab	8-41
Viewing the Heap Snapshot at the End of the Recording Information	8-42

Viewing the Heap Contents Information	8-43
Viewing the Free Memory Contribution Information	8-43

Object Statistics Information

Getting Familiar with the Object Statistics Tab	9-45
Viewing Start of Recording Information	9-46
Viewing End of Recording Information	9-47

Code Optimization Information

Getting Familiar with the Optimizations Tab	10-49
Viewing Optimization Information	10-50
Viewing Methods Optimized During Recording Information	10-51

Lock Profiling Information

Getting Familiar with the Lock Profiling Tab 11-53
Java Locks Profiling 11-54
Enabling Java Lock Profiling Data 11-55
Native Lock Profiling 11-55
Enabling Native Locks Information

Start and End Processes Information

Getting Familiar with the Processes Tab	12-57
Snapshot of Processes at Beginning and End of Recording	12-58
Detailed Processes Information	12-59

Thread Latency Viewer Overview

Getting Familiar with the Thread Latency Overview Tab	-61
Event Graph Window	-63
Workflow From Recording to Viewing	-63
Drilling Down to Your Problem	-63

Expanding and Collapsing Thread Nodes	13-63
Customizing Your View	13-63
Using the Filter Functions	13-63
Using the Time Scale on Top of Window	13-63
Getting Familiar with the Events Table Tab	13-63
About Selecting Events for "intressanta mängden"	13-63
Selecting Events for "intressanta mängden"	13-63
Deleting Events from "intressnta mängden"	13-63
	13-63

Getting Familiar with the Events Table Tab

About Selecting Events for "intressanta mängden"	14-65
Selecting Events for "intressanta mängden"	14-65
Deleting Events from "intressnta mängden"	14-65
Filtering Columns	14-65
	14-65

Getting Familiar with the Stack Trace Tree Overview Tab

About the Stack Trace Tree Overview Tab	15-67
Adding and Removing Content in the Tree Table	15-67
Deleting Events from "intressanta mängden"	15-67
Filtering Columns	15-67
	15-67

Adding Comments to a Recording



Welcome to the BEA JRockit Runtime Analyzer (JRA)

The JRA is a JVM profiler and a Java application profiler that is especially designed for BEA JRockit. It has been around for quite some time within the JRockit development team, and was originally created to let the JRockit developers find good ways to optimize the JVM based on real-world applications, but it has proven very useful to developers outside of BEA for solving problems in both production and development.

This section is divided into the following topics:

- How Does the JRA System Work?
- What is a JRA recording?
- What is the JRA tool?

How Does the JRA System Work?

The JRA system consists of two parts (see Figure 1-1): one part inside the JRockit JVM that collects data and saves it, and an analysis tool that visualizes the information to make it useful to the end-users. The JRockit-internal part produces a recording of the system's runtime behavior during a user specified period of time, typically a few minutes. The recording results in an XML file that JRockit writes to disk and automatically launches in the JRA tool once the recording is complete (this behavior is valid for JRockit 5.0; for JRockit 1.4, the file is saved to disk and you need to locate it first to open it).

The recording is a great way to share how JRockit has worked with your application. You can also use several recordings to compare and contrast how different command line options change

the behavior of your application, for example, by creating before-and-after recordings. When sending trouble reports to the BEA JRockit support department, you are required to attach a JRA recording to your trouble report. The recording is analyzed "offline" by the Runtime Analyzer tool.



Figure 1-1 The BEA JRockit Runtime Analyzer System

The recording engine uses several sources of information including the JRockit Hot Spot Detector (also used by the optimization engine to decide what methods to optimize), the operating system, the JRockit Memory System (most notably the garbage collector), and the JRockit lock profiler, if enabled.

What is a JRA recording?

It is a collection of data about the JVM that can be used to analyze the doings and happenings of JRockit. It is also a "flight recording" of what has happened in the JVM when running the JRA.

What is the JRA tool?

The JRockit Runtime Analyzer tool is a Java application that parses a previously produced JRA recording and visualizes the data. This is a convenient way to analyze the data offline. The size of the compressed recording is on the order of a few hundred kilobytes, so a system administrator can easily make a recording of a deployed system and send it to the JVM or application developer who probably is in a better position to analyze it.

The JRA tool shows a top list of the hottest methods where you can select a method and see its call tree, i.e. its predecessors (what other methods have called this method) and successors (what

methods the selected method will call). A percentage for each branch indicates how common a given path is.

As for memory management, there is a graph of the varying heap usage and pause times for the garbage collections. Detailed information about each GC shows exactly how much memory was released in a collection. There are also pie charts showing the distributions in size of free memory blocks and the distribution of occupied memory in small and large object chunks.

Welcome to the BEA JRockit Runtime Analyzer (JRA)



Using the JRA Tool

How to use the JRA Tool is divided into the following topics:

- Getting Started with the BEA JRockit Runtime Analyzer Tool
- General Information in JRA Recording
- Methods and Call Trace Information
- Garbage Collection Events Information
- General Garbage Collector Information
- Java Heap Content Information
- Object Statistics Information
- Code Optimization Information
- Lock Profiling Information
- Start and End Processes Information
- Adding Comments to a Recording

Using the JRA Tool



Getting Started with the BEA JRockit Runtime Analyzer Tool

Before you can view how your application is behaves, you need to create a JRA recording, i.e. collect data from your application. The JRockit recording engine produces a recording of the system's runtime behavior during a specified period of time, typically a few minutes. The recording results in an XML file that opens automatically in the JRA tool upon completion (for JRockit 1.4, the XML file is saved to the disk where JRockit is running). The file can be analyzed "offline" by the Runtime Analyzer tool.

This section describes how you start a recording and how you setup the JRA tool to suit your needs.

The following topics will be covered:

- Creating a Recording
- About JRA Overhead when Recording
- Opening a JRA File
- Comparing and Contrasting JRA Recordings
- Customizing Your JRA Tool

Creating a Recording

There are several ways to start a JRA recording:

• To start a recording within Mission Control

Getting Started with the BEA JRockit Runtime Analyzer Tool

- To start a recording with jrcmd
- Starting a Recording from the JRockit Command Line
- **Note:** If you are running Mission Control on a Windows system, you need to be a member of the **Administrators** or the **Performance Logs** user groups to be able to create a JRA recording. The typical error message, for not being part of either of these groups, can look like this:

[perf] Failed to init virtual size counter:

To start a recording within Mission Control

- 1. Start your Java application with JRockit and add the -Xmanagement option to the command line.
- 2. Start the JRockit Browser and connect to the JRockit instance you just started.
- 3. Make sure that your application is running and is under load.

If you run the application without load, the data captured from that application will not show where there is room for improvements.

4. Click the Start JRA recording button.

The JRA Recording dialog box appears (Figure 3-1).

Figure 3-1 JRA Recording Dialog Box

茨 Start JRA Recording			×
Start a JRockit Runtime An This dialog helps you start new JRA can be opened by a JRA Editor.	alyzer Record recordings and cre	ing ate the resulting files that	- A
Selected connections	Options Local filename Recording time Records sam Use gc samp Use native s Compress re	./recording.jra 300 ples of methods ling amples cording	Browse
		Start	Cancel

5. Select the connection you want to record.

6. Type a descriptive name for the recording in the Local filename field.

The file is created in the current directory of the BEA JRockit process, unless you specify a different path. If an old file already exists, it will be overwritten by the new recording.

7. Set a time for the length of the recording (in seconds) in the **Recording time** field.

Note: If you set a time that is too short, e.g. shorter than 30 seconds, you will probably not get enough sample data for the recording to be meaningful.

- 8. Select none, one, or all of the following sampling options:
 - Records 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

The Selected JRockits field displays which JRockit you will create your recording from.

9. Click Finish.

The JRA recording progress window appears. When the recording is finished, it loads in the JRA tool.

To start a recording with jrcmd

1. Make sure that your application is running and is under load.

If you run the application without stress, the data captured from that application will not show where there is room for improvements.

2. Use one of the following commands to initiate a recording:

Windows platforms:

```
bin\jrcmd.exe <pid> jrarecording time=<jrarecording time>
filename=<filename>
```

Unix platforms:

bin/jrcmd <pid> jrarecording time=<jrarecording time> filename=<filename>

Where the arguments are:

- jrarecording time—the duration of the recording in seconds (a good length is 300 seconds, i.e., five minutes).

- filename—the name of the file you want to save the recording to (for example jrarecording.xml.zip). The file will be created in the current directory of the JRockit process. It will be overwritten if it already exists.

For example: bin\jrcmd.exe <pid> jrarecording time=300 filename=c:\temp\jra.xml.zip Starts a JRA recording of 300s and stores the result in the specified file.

After the recording is initiated, BEA JRockit prints a message indicating that the recording has started. When the recording is done, it will print another message; it is now safe to shut down your application.

Starting a Recording from the JRockit Command Line

Use the -XXjra command in combination with an option listed in Table 3-1, for example, -XXjra:recordingtime to specify the duration of the recording.

Option	Description
delay	Amount of time, in seconds, to wait before recording starts.
recordingtime	Duration, in seconds, for the recording. This is an optional parameter. If you don't use it, the default is 60 seconds)
filename	The name of recording file. This is an optional parameter. If you don't use it, the default is jrarecording.xml.
sampletime	The time, in milliseconds, between samples. Do not use this parameter unless you are familiar with how it works. This is an optional parameter.
nativesamples	Displays method samples in native code; that is, you will see the names of functions written in C-code. This is an optional parameter.
methodtraces	You can set this to false to disable the stack trace collection that otherwise happens for each sample. The default value is true.
tracedepth	Sets the number of frames that will be captured when collecting stack traces. Possible value are 0 through 16. The default value is 16.

Table 3-1 Command Line Startup Options

Note: Setting methodtraces to false can still result in some stack traces being captured. These stack traces are captured as part of JRockit's dynamic optimizations and will have a depth of 3. If optimizations are turned off (-xnoopt) these traces will not be captured.

You can view the startup options that you have set in the JRA recording, see Viewing VM Arguments Information. Listing 3-1 shows an example of how you can setup a JRA recording.

Listing 3-1 An example of using the -xxjra startup command:

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

would result in a recording that:

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

About JRA Overhead when Recording

The overhead while recording is very low—typically less than two percent. However, since JRA is forcing a full garbage collection at the beginning and at the end of the recording to generate the heap histogram data, there may be a spike at the beginning and at the end of a recording.

Opening a JRA File

Once you have created a JRA file, you can open it in the JRA tool to view what has happened in your application and in JRockit.

Note: If you have previously viewed a JRA recording in the JRA tool, it will automatically load when you open JRockit Mission Control.

To open a JRA file by dragging and dropping

- 1. Locate the JRA recording on your system.
- 2. Drag and drop the file to JRockit Mission Control.

To open a JRA file from JRockit Mission Control

- 1. In JRockit Mission Control, click File > Open file > Open JRA Recording.
- 2. Locate and select the recorded file and click Open.

3. Click OK.

The **JRA General** tab now opens and you can view the data in the recording (see Figure 4-1).

Note: If you have opened a recording that has been recorded with an older version of the JRA, some fields may not have any relevant data, since that data was impossible to obtain. That data will appear as "N/A".

Comparing and Contrasting JRA Recordings

The new JRA is excellent to use for comparing and contrasting recording. For example, you want to try different startup options in JRockit to see how they affect the running of your application.

To compare and contrast JRA recordings

- 1. Create two recordings, one for each setting you wish to try.
- 2. Open both recordings and lay them out in the JRA tool next to each other, by simply dragging the name tab to the toolbar in the JRA tool window.

Customizing Your JRA Tool

The following can be set to change the way you view a recording:

- Changing Table Settings
- Filtering Information
- Collapsing and Expanding a View
- Changing View of a Tab

Changing Table Settings

The JRA tool lists a lot of information in different tables. These tables can be customized to display information of your choice. You can also preset the width of the columns in the tables.

Note: You need to change the settings per table, i.e. there is no global change to all tables since they contain different types of information depending on the tab you are looking at.

To change the settings of the table

1. Click the Table settings button (see Figure 3-2).

Figure 3-2 Table settings button

		\frown			
Top Hot Methods					
The methods where JRockit spent the most time executing. Total number of samples are 1,645.					
Filter column Method 💉 💌					
Method	%	#Samples			
🞯 java.util.Hashtable.put(Object,Object)	54.10%	890			
🞯 java.util.Hashtable.remove(Object)	17.33%	285			
DemoLeak\$DemoThread.put(int)	7.60%	125			
💿 java.lang.Thread.yield()	6.20%	102			
OPpemoLeak\$DemoThread.remove(int)	5.47%	90			

A Table settings window appears (see Figure 3-3).

Figure 3-3 Table settings Window

🔲 Table setti	ngs			X
Table setting Configure table	s			
Visible		Min. width	Weight	Initial sort order
Method		240	240	Descending
✓ %		60	60	Descending
🖌 #Samples		60	60	Descending
Optimized		60	60	Descending
Native		60	60	Descending
Method Id		60	60	Descending
Minimum width 240				
Weight	240			
🔲 Initial sort o	rder			
				ОК

- 2. Select what you want displayed in the table.
- 3. Set the Min. width and Weight of the column (optional) to a pixel value of your choice.
- 4. Select Initial sort order for a table item that you want the table to be sorted by.
- 5. Click OK.

Filtering Information

Some of the information tables can contain lengths of data that can be hard to scroll through. Instead of scrolling through the long tables, you can filter for the information that you are interested in viewing.

To filter information

- 1. Select a table column name for which you want to filter the information. In this example, Figure 3-4, Pause Time was selected.
- 2. Enter a number or text for the information you want to see. In this example, Figure 3-4, 60* was used to see all Pause Times that contains a value starting with 6 and 0 (zero).

Figure 3-4 Filtering information

Garbage Col	lections				
Garbage collections that occured during the recording					
Filter column	Pause Time	✓ 60*			
Index	Pause Time	Generation			
79	609 ms	Old collection			
72	607 ms	Old collection			
60	605 ms	Old collection			
62	605 ms	Old collection			
52	603 ms	Old collection			
77	603 ms	Old collection			

Collapsing and Expanding a View

Sometimes the information on a tab can be cumbersome to work with, then it is good to collapse the view of some fields.

To collapse/expand a view

• Click on the small arrow next to a description field (see highlight in Figure 3-5) to collapse the view of the General Information field.

Figure 3-5 Collapsing a view

 General Information 			
This section describes general information about the record			
JRA file format version:	3.3		
Recorded on:	Mon Mar 20 09:19:07 2006		
Expected recording time:	100 same		

Changing to view less values by right clicking a field. The next time you start the JRA tool, you will not see the specific field.

Changing View of a Tab

Sometimes the method names are hard to view in the default horizontal layout, therefore, you might want to change the layout to a vertical view instead.

To change the layout of a tab

• Click either the **Horizontal layout** or the **Vertical layout** button in the right hand corner of the tab that you are viewing (see Figure 3-6).

Note: Not all tabs have this functionality.

Figure 3-6 Horizontal and Vertical layout buttons



Getting Started with the BEA JRockit Runtime Analyzer Tool



General Information in JRA Recording

The JRA recording contains a lot of data about the application's behavior, information about JRockit itself, such as JRockit version and which commands were used at the startup of JRockit, etc. All that information is displayed on the **General tab** in the JRA tool. As soon as you have made a recording, your JRA tool automatically opens the recording and displays general information on the **General tab**.

For recordings that have been generated with a JRockit that is older than R26.4, you might still be able to open them up in this version of the JRA tool; however, some fields may be blank, since older versions of JRockit did not have the same recording capabilities as this newer release.

Note: Only text fields that require extra explanations have been covered in this documentation.

This section is divided into the following topics:

- Getting Familiar with the General Tab
- Viewing General Information
- Viewing Memory Usage Information
- Viewing VM Arguments Information
- Viewing Memory Allocation Information
- Viewing Threads Information
- Viewing Exceptions Information

Getting Familiar with the General Tab

The **General** tab (see Figure 4-1) contains information on both JRockit, your system, and your application.

Figure 4-1 The General tab

🐹 jrarecording.xml.zip 🗙	jennyapp.xm	l.zip 🚫 R27.1.0-4	1_3.xml	🔀 jrarecording.xml			0	
General	-			-				^
 General Information 		0	 Alloc 	ation (4)			0	
This section describes gen	eral information about	the recording.	Memory	allocation information.				
JRA file format version:	B.6		Thread	local area size:	8.000	kB		
Recorded on:	Mon Aug 21 21:	34:47 2006	#TLAs	allocated:	20,12	0,258		
Expected recording time:	600 s		#large	objects allocated:	1,247	,512		
Actual recording time:	600 s		Size of	largest object allocated:	64.01	6 kB		
JRockit version:	BEA JRockit(R)		Freque	ncy - large objects:	22.15	3 MB/s		
	R27.1.0-10-655 1-1131-linux-ia3	49-1.5.0_07-2006081 2	Freque	ncy - small objects:	261.9	83 MB/s		
Operating System:	Red Hat Enterpr	ise Linux AS release 4	Averag	e size of large objects:	10.91	0 kB	5	
	(Nahant Update 2,6,9-34,Fl smp	3) Linux version	Ratio o	f bytes for large/small objects:	0.085		5	
	(bhcompile@hs2	0-bc1-7.build.redhat.	#free li	st misses:	0		5	
	(Red Hat 3.4.5-	2)) #1 SMP Fri Feb 24						
the share of grants	16:54:53 EST 20	IO6 (i686)	 Three 	ads 5			0	
Number of CPUs:	4		Thread	Thread specific information gathered during the		e recording.	_	
Total physical memory:	3.957 GB		Total n	Total number of threads before recording:		94		
VM information:	compiled mode		Total n	Total number of threads after recording:		94		
Maximum heap:	2.637 GB		Number of daemon threads before recording		cording:	93		
Number of code blocks:	1		Number of daemon threads after recording:		ording:	93		
Total size of code blocks:	11.483 MB		Number of threads started during recording:		ording:	0		
Unused space in code bloc	:ks: 1.050 MB		System	total of #context switches/se	cond:	36.694		
Memory lisage	\bigcirc	0	T Exce	ntions 6			0	
Memory information about	the JRockit Process.	U	Exception	ons related information.			0	
	Poforo compling	After complete	Total #	exceptions thrown during reco	rding:	226.510		
Virtual memory usage:	2.869 GB	2.869 GB	#hardv	vare generated exceptions thr	own:	0	=	
Physical memory usage:	2.822 GB	2.822.68	Number	of exceptions/second:	5	377.517	-	
Committed java heap:	2.637.CR	2.637.08				0771017		
Page faults:	2.037 30	2.037 GD						
	0	0						
	\bigcirc						_	
 VM Arguments 	9						0	
startup arguments for the	. PTVL	dla : bi : in :						
-Dcom.sun.xml.namespa -Xverbose:opt,gc,gcpau -XXheapParts=128 -Dwe	ce.QName.useCompa se,memdbg,compactio blogic.socket.Muxer=	tibleSerialVersionUID=1. n -XXcallprofiling=false - 1	0 -Xms2700 XXinternalC	im -Xmx2700m -XXaggressive - iompactRatio=1 -XXexternalCo	Xnoopt: mpactRa	1320 atio=10	~	
-Djavax.xml.parsers.Doc -Djava.library.path=/loc	umentBuilderFactory= alhome/tests/specian	com.sun.org.apache.xe 04/wls910/wls910/webl	rces.intern ogic91/serv	al.jaxp.DocumentBuilderFactor er/native/linux/i686:/localhome	ryImpl e/irockits	Jr27.1.0 R		~
General Methods GC Even	ts GC General Heap	Overview Object Statis	tics Optim	izations Lock Profiling Comm	ents			

The General tab is divided into the following sections:

- 1. General Information—contains all general information about the JVM, operating system, recording time, etc.
- 2. Memory Usage—contains information on how JRockit is using the memory.

- 3. VM Arguments—lists all startup options that were used.
- 4. **Allocation**—contains information on how your application allocates memory on the Java heap.
- 5. Threads—contains information on thread usage.
- 6. Exceptions—contains exceptions related information.

Viewing General Information

This section displays (see Figure 4-2) information about the JRockit version, the operating system version, number of CPUs that has been used during the recording, etc.

- The value Actual recording time can differ from expected recording time, e.g. if the application that runs on BEA JRockit finished while a recording was still in progress.
- The Maximum heap size is set with a JRockit command-line option.
- The VM information can be information regarding the garbage collection that has been used.
- The value **Number of codeblocks** is a JVM internal value. All generated code is divided into (non-heap) memory blocks called code blocks.

Figure 4-2 General Information section

 General Information 	0
This section describes general	information about the recording.
JRA file format version:	3.3
Recorded on:	Mon Mar 20 09:19:07 2006
Expected recording time:	100 s
Actual recording time:	100 s
JRockit version:	BEA JRockit(R) R26.0.0-189-53463-1.5.0_04-20051122-2041-wi n-ia32
Operating System:	Microsoft Windows XP version 5.1 Service Pack 2 (Build 2600)
Number of CPUs:	1
Total physical memory:	1,023.227 MB
VM information:	N/A
Maximum heap:	256.000 MB
Number of code blocks:	1
Total size of code blocks:	589.013 kB
Unused space in code blocks:	0 bytes

Viewing Memory Usage Information

This section (see Figure 4-3) shows a snapshot of the memory usage before and after the recording.

• The value **Committed java heap** was the current total heap size at the beginning and the end of the recording. It is less than or equal to the maximum heap size.

Figure 4-3 Memory Usage section

 Memory Usage 			?
Memory information about	t the JRockit Process.		
	Before sampling	After sampling	
Virtual memory usage:	333.688 MB	349.328 MB	
Physical memory usage:	26.441 MB	287.227 MB	
Committed java heap:	256.000 MB	256.000 MB	
Page faults:	16,442	103,278	

Viewing VM Arguments Information

This section displays (see Figure 4-4) the different command-line options that were used when starting JRockit. The options that have been used in the example are the following:

- The JRA recording time (XXjra) has been set (100 seconds).
- The name of the recorded file has been set (filename) and native sampling has been switched on.
- The initial, minimum and maximum Java heap has been set (-xms and -xmx)
- The management server is started (-Xmanagement)
- The default dynamic garbage collector has been turned off and the static parallel garbage collector is used instead (-XXsetgc)

There are many more command-line options that can be set. For comprehensive information on the different command-line options, please see the BEA JRockit Reference Manual.





Viewing Memory Allocation Information

This section displays (see Figure 4-5) information about how JRockit is allocating memory on the Java heap.

- The **Thread local area (TLA) size** is a JRockit internal value. It is a small memory area, local to a thread, where the JVM can allocate small objects without having to take the heap lock.
- Ratio of bytes for large/small objects. Per default, JRockit considers an object to be large if it is larger than the thread local area size; it is small if it would normally fit in a thread local area. Large objects are always allocated in the old space (second generation) of the heap, never in the nursery.
- The Number (#) free list misses is a JRockit internal value. JRockit has a list of free memory blocks on the Java heap. During allocation, an object is normally put in the first free block on the "free list." If it does not fit there, JRockit will try the next block, and the next, etc. Each block where the code block did not fit is considered a "free list miss."

Figure 4-5 Allocation section

 Allocation 	?
Memory allocation information.	
Thread local area size:	2.000 kB
#TLAs allocated:	5,646,632
#large objects allocated:	12
Size of largest object allocated:	6.000 MB
Frequency - large objects:	122.852 kB/s
Frequency - small objects:	110.286 MB/s
Average size of large objects:	1,023.766 kB
Ratio of bytes for large/small objects:	0.001
#free list misses:	3

Viewing Threads Information

This section displays (see Figure 4-6) information on the number of Java threads that existed both before and after the recording.

- The value of **Number of deamon threads before/after recording** is the number of deamon threads. A deamon thread is a thread that runs in the background to support the runtime environment, for example, a garbage collector thread. The JVM exists when all non-daemon threads have completed.
- The value **System total of #** (*number*) **context switches per second** is fetched from the operating system. An unusually high context switch value compared to other applications may indicate contention in your application.

Figure 4-6 Threads section

 Threads 	0
Thread specific information gathered during the	e recording.
Total number of threads before recording:	10
Total number of threads after recording:	15
Number of daemon threads before recording:	9
Number of daemon threads after recording:	13
Number of threads started during recording:	6
System total of #context switches/second:	229,707.979

Viewing Exceptions Information

This section displays (see Figure 4-7) information on the total number of Java exceptions that are thrown during a recording. This includes both caught and uncaught exceptions. Excessive exception throwing can be a performance problem. Hardware generated exceptions are originating from a "trap" in the hardware and are usually the most "expensive" kinds of exceptions.

Figure 4-7 Exceptions information

 Exceptions 	0
Exceptions related information.	
$\label{eq:total_total} \ensuremath{\texttt{Total}}\xspace \ensuremath{\texttt{#exceptions}}\xspace \ensuremath{\texttt{total}}\xspace \texttt{$	10
#hardware generated exceptions thrown:	0
Number of exceptions/second:	0.1



Methods and Call Trace Information

Methods where JRockit spends most of its time are called hot. Once you have identified such a method, you might want to investigate it to see if it is a "bottleneck" for the application or not. The way that BEA JRockit collects method information is via a sampling thread that is called the hotspot detector. It uses statistical sampling to find Java methods that are candidates for optimization. The samples are collected by iterating through the Java threads in the virtual machine and suspending them one at a time. The current instruction pointer of the suspended thread is used to lookup in which Java method the thread is currently executing. The invocation count of the method is incremented and the method is added to a queue of methods to be optimized if the invocation count exceeds a certain threshold.

The JRA recording system makes use of the hotspot detector by setting it to a high sampling frequency during the recording and directing the samples to the .jra file.

This section is divided into the following topics:

- Getting Familiar with the Methods Tab
- Viewing Hot Methods
- Viewing Predecessors and Successors

Getting Familiar with the Methods Tab

The **Methods tab** is divided into three sections (see Figure 5-1) that lists the top hot methods with its predecessors and successors during the recording.

n Hat Methods			Brederoscore
a mathada where 10 and a most the most time of	e e die e		Predecessors (2)
e medious where skockit spenit the most time ex	ecuality.		jrockit.vm.ArrayCopy.copy_checks_done1(Object,int,Object,int,int)
ter column Method			5 100.00% [860] jrockit.vm.ArrayCopy.copy_checks_done1(Ob)
Method	%	#Sam 🔨	3 74.30% weblogic.servlet.internal.ChunkOutput.write(byb)
ArrayCopy.copy_checks_done1()	2.43%	ç -	9.65% oracle.net.ns.NetOutputStream.write(int) (OPT)
FloatingDecimal.dtoa()	2.31%	5	4.19% oracle.net.ns.NetOutputStream.write(byte[],int,in
AbstractStringBuilder.append()	2.30%	ę.	Image: Source and Source and Source and Source and Image: Image: Image: Image: Source and Image:
DecimalFormat.subformat()	1.55%	e	B 3.02% oracle.net.ns.NetInputStream.read(byte[],int,int)
🖉 String.getBytes()	1.50%	Ę	100 100 100 100 100 100 100 100 100 100
BomEntPK.hashCode()	1.41%	Ę	1.28% weblogic.servlet.internal.ResponseHeaders.writer
StringBuilder.toString()	1.25%		B 0.58% weblogic.utils.io.UnsyncByteArrayOutputStream.
ArrayCopy.copy_checks_done2()	1.06%		B 0.35% oracle.jdbc.driver.T4CNumberAccessor.copyRow
DBManager.preInvoke()	1.05%		B 0.23% java.nio.HeapByteBuffer.put(byte[],int,int) (OPT)
ConcurrentHashMap.getEntry()	1.04%		
DecimalFormat.format()	0.98%		
FloatingDecimal.readJavaFormatString()	0.87%	:	European A
SpecUtils.formatCurrency()	0.87%		Successors 3
DigitList.set()	0.81%		Successors for
ChunkOutput.print()	0.80%	:	procedure and a second and a second a
VetOutputStream.write()	0.79%	:	100.00% [860] jrockit.vm.ArrayCopy.copy_checks_done1(O)
HashMap.put()	0.73%		74.30% weblogic.servlet.internal.ChunkOutput.write(byt)
ChunkOutput.write()	0.73%	1	T4.30% weblogic.servlet.jsp.JspWriterImpl.write(byt)
HashMap.addEntry()	0.66%	2	9.65% oracle.net.ns.NetOutputStream.write(int) (OPT)
DebugLogger.isDebugEnabled()	0.65%		3.49% oracle.jdbc.driver.T4C8Oall.marshalPisdef() (
SpecUtils.formatNumber()	0.61%		2.09% oracle.jdbc.driver.T4CMAREngine.value2Buffe
VetInputStream.read()	0.60%	2	1.86% oracle.jdbc.driver.T4C8Oall.marshalAll(T4CTT)
JTSConnection.getOrCreateConnection()	0.59%	2	0.93% oracle.jdbc.driver.T4CMAREngine.marshalUB1
HashMap.get()	0.59%		 9 0.70% oracle.jdbc.driver.T4CTTIoac.marshal() (OPT)
DBManager.getBeanFromRS()	0.59%	2	0.35% oracle.jdbc.driver.T4C8Oall.marshalBinds(int[]
			1 1.23% oracle.idbc.driver.PhysicalConnection.commit

Figure 5-1 The Methods tab

The Methods tab is divided into the following sections:

- 1. **Top Hot Methods**—a listing of the top hot methods. Click on the different table headings to get a different sort order.
- 2. **Predecessors**—a listing of all preceding methods to the method that you have selected in the **Top Hot Methods** list. If you have selected many methods, there will not be any information shown in this section.
- 3. Successors—a listing of all succeeding methods to the method that you have selected in the **Top Hot Methods** list. If you have selected many methods, there will not be any information shown in this section.

Viewing Hot Methods

The method sampling in JRockit is based on CPU sampling. This requires that you put load on the system to get any samples. The **Top Hot Methods** lists (see Figure 5-2) all methods sampled

during the recording and sorts them with the most sampled method s first. These are the methods where most of JRockit's time is spent.

Figure 5-2 Top Hot Methods shown

i op noc Piechous			
The methods where JRockit spent the most time e	xecuting.		
Method	%	#Sam	Optimi
🞯 java.util.Hashtable.put(Object,Object)	38.66%	75	Yes
🞯 java.util.Hashtable.remove(Object)	9.28%	18	Yes
🞯 DemoLeak\$DemoThread.put(int)	6.70%	13	Yes
🞯 DemoLeak\$DemoThread.remove(int)	4.12%	8	Yes
💿 java.util.Hashtable.rehash()	3.09%	6	No
💿 DemoLeak\$DemoThread.run()	3.09%	6	No
💿 java.util.Hashtable.put(Object,Object)	2.06%	4	No
of irockit.vm.Locks.monitorEnter(Object)	2.06%	4	Yes
🌀 jrockit.vm.Allocator.innerAllocate(int,int,in	2.06%	4	No
💿 jvm.dll#_qBitSetClear	2.06%	4	No
🞯 jrockit.vm.Allocator.innerAllocate(int,int,in	2.06%	4	Yes
Q DemoLeak\$DemoObject.equals(Qbject)	2.96%		No

If your recording has native sampling enabled during the recording, you can see methods prefixed by jvm, which are native methods in the JVM.

Viewing Predecessors and Successors

By selecting a method in the **Top Hot Methods** list, you can see its sampled **Predecessors** and **Successors** (see Figure 5-3) in the tree views to the right (or below if you have changed the layout of the tab). These are the methods that call the selected method and the methods the selected method calls.



Figure 5-3 Viewing Predecessors and Successors

The number within brackets is the number of sampled call traces of which the method is part. The percentage shows how common a particular path is in the method tree. If you see methods that are called a lot from JRockit, you might want to investigate if that method is causing your application to run slower than necessary.



Garbage Collection Events Information

The **GC Events** tab shows detailed information about each garbage collection (GC) event that has occurred. The tab contains a graph for Java heap usage before and after each garbage collection as well as detailed garbage collection information for each collection.

This section is divided into the following topics:

- Getting Familiar with the GC Events Tab
- Changing Focus on Heap Usage Chart
- Viewing Specifics about Garbage Collections
- Viewing the Detailed Information About the Garbage Collection

Getting Familiar with the GC Events Tab

The **GC** Events tab is divided into six sections (see Figure 6-1) that pictures how the garbage collector has performed during the recording.



Figure 6-1 The GC Events tab

The GC Events tab is divided into the following sections:

- 1. GC Events Overview chart—this chart shows the entire recording in its full length (when you initially open your recording). You can use this to refocus the Heap Usage graph, see Changing Focus on Heap Usage Chart.
- 2. **Heap Usage** graph—this graph shows heap usage compared to pause times and how that varies during the recording. If you have selected a specific area in the GC Events Overview, you will only see that section of the recording. You can change the graph content in the **Heap Usage** drop-down list (marked 6 in Figure 6-1) to get a graphical view of the references and finalizers after each old collection.
- 3. Garbage Collections events—this list shows all garbage collection events that have taken place during the recording. When you click on a specific event, you will see a corresponding flag in the Heap Usage graph for that particular event, see Viewing Specifics about Garbage Collections.
- 4. **Details**—this section contains all the details about the specific garbage collection round. When you select a garbage collection in the **Garbage Collection** list, the tabs in the **Details** section changes depending on if you have selected an old collection or a young collection.

- 5. Chart Configuration—this section allows you to change the appearance on the active chart.
- 6. **Heap Usage**—this list allows you to toggle the view on the **Heap Usage** chart to view References and finalizers. It shows different types of reference counts after each collection.

Changing Focus on Heap Usage Chart

Depending on how long your JRA recording is, the **Heap Usage** chart can be quite cumbersome to view in full mode; therefore, you can refocus the chart. by dragging the handles on the slide bar to the section of the recording that you want to view. Once you have set the side on the slide bar, you can slide that section to the position of the chart that you are interested in studying.

To change focus on the Heap Usage chart

1. Click and drag the handles on both sides on the GC Events Overview chart (see Figure 6-2).

Figure 6-2 The GC Events overview zoom function



2. Drag the GC Events overview chart into the desired position for the Heap Usage chart (see Figure 6-3).

Figure 6-3 The GC Events overview chart



Viewing Specifics about Garbage Collections

The **Garbage Collections** section on the **GC Events** tab is a list of all garbage collections that have taken place during the recording. It lists all garbage collection events during the recording, provided that the garbage collection sampling was enabled. A garbage collection can be an *old*

collection, which is a garbage collection in the old space of the Java heap or a *young collection*, which is a garbage collection in the young space (nursery).

To view one garbage collection in the Heap Usage chart and Details section

- 1. Scroll in the Garbage Collection list to the garbage collection you want to view.
- 2. Click on that garbage collection.

The garbage collection index number is now visible in the **GC Chart** and the **Details** section has also changed to show all the specifics about that garbage collection.

The **Details** section changes name depending on if you selected an old collection or a young collection (see Figure 6-4).



Figure 6-4 Viewing garbage collection in Heap Usage chart and Details section

To view many garbage collections in the Heap Usage chart

- 1. Scroll in the Garbage Collections list.
- 2. Click and hold either the Shift key or Ctrl key to select multiple collections.

The garbage collection index numbers are now visible in the GC Chart (see Figure 6-5).


Figure 6-5 Viewing multiple garbage collections in Heap Usage chart

Viewing the Detailed Information About the Garbage Collection

When you select a garbage collection, the **Details** section of the **GC Events** tab changes name to either **Details - Old Collection** or **Details - Young Collection** depending on the type of garbage collection you have selected (this is only possible when selecting one garbage collection). You will also see different sets of tabs that contain specific information about the garbage collection that you have selected (see Figure 6-6).

Figure 6-6 Tab differences when viewing old and young collections

Details - Old Collection		1	Details - Young Collection	
General GC method call tree	Old Collection Cache Lists Pause time		General GC method call tree	Young Collection
Generation:	Old collection		Generation:	Young collection
Pause Time:	68 ms	1	Pause Time:	63 ms
Start Time:	1:00:06 AM 719 ms		Start Time:	1:00:14 AM 412 ms
End Time:	1:00:06 AM 788 ms		End Time:	1:00:14 AM 475 ms
Heap Usage Before:	38.488 MB		Heap Usage Before:	53.445 MB
Heap Usage After:	27.550 MB		Heap Usage After:	51.403 MB
# Final References:	142	5	# Final References:	1
# Phantom References:	0	2	# Phantom References:	0
10000		1	10000	
Details for old collection		5	Details for young collection	DN

Each one of these tabs are described here. As much of the information in the tabs are fairly self-explanatory, those types of details will not be covered in the documentation.

This section describes the following tabs:

- Viewing Information on the General Garbage Collection Tab
- Viewing Information on the GC Method Call Tree Tab
- Viewing Information on the Old/Young Collection Tab
- Viewing Information on the Cache Lists Tab (Only valid for old collections)
- The Pause Time Tab

Viewing Information on the General Garbage Collection Tab

The **General** tab (see Figure 6-7) displays information such as start time and end time of the garbage collection.

Figure 6-7 The General garbage collection tab

General GC method call tree	ol	d Collection	Cache Lists	Pause time	
Generation:		Old collecti	on		1
Pause Time:	1	94 ms			1
Start Time:	Ì	1:00:08 AN	1710 ms		-
End Time:	Ì	1:00:08 AN	1 804 ms		3
Heap Usage Before:	Ì	52.691 MB			1
Heap Usage After:		41.576 MB			
# Final References:		253			
# Phantom References:		0			-
# Soft References:		8			
# Weak References:	ĺ	4			1
Committed Heap Size:	ĺ	61.652 MB			
Finalizer Queue Length:		111 Junior			

- Generation—Indicates whether the garbage collector performed an old or young collection (see the Memory Management user guide for more information on generational garbage collection). If a parallel garbage collector has been used, there will be only old collections in the Garbage Collections list.
- **Pause Time**—the time in milliseconds that the garbage collector stops all threads in JRockit. This is not the same as end time-start time in the case of a concurrent garbage collector.
- **Start/End Time**—the times when the garbage collection started and ended, counted in milliseconds from when JRockit started.

- Heap Usage Before/After—the used heap size before or after the garbage collection.
- Number of References—there are different types of references collected during a recording. For information on what a reference is, see the Memory Management user guide.
- **Committed Heap Size**—the total size of the heap (used plus unused memory) after the garbage collection.
- Finalizer Queue Length (and Before)—the finalizer queue length.

Viewing Information on the GC Method Call Tree Tab

The **GC Method Call Tree** tab (see Figure 6-8) shows an aggregation of the call traces of the threads triggering a garbage collection.

Figure 6-8 The GC Method Call Tree tab



Viewing Information on the Old/Young Collection Tab

The name of this tab is dynamically changed when you select a garbage collection instance in the **Garbage Collections** section. Here you find information about nursery, mark and sweep pause times, etc. (see Figure 6-9).

General GC method call tree	Old Collection Cache Lists Pause time
Nursery Size Before:	2.287 MB
Nursery Size After:	2.058 MB
Nursery Start Position:	N/A
Nursery End Position:	N/A
Mark Phase Time:	144 ms
Sweep Phase Time:	97 ms
Compacted Size:	4.509 MB
Compaction Ratio:	6.25%
Desired Contraction Amount:	0
Actual Contraction Amount:	0
# Compacted Parts:	8
Is Compaction Exceptional:	No
Pause Time Ref. Updates:	0 ms
Rause Jarget Rof -Undates:	كالمحالي ومحارب فالمراجع والمحاور والمحاوم والمحافظ ومحافظ والمحاف والمحافي والمحافظ والمحافي والمحافي والمحافي والمحافي

Figure 6-9 The Old/Young Collection tab

• Nursery Size Before/After—indicates the size of the young space on the heap before and after the garbage collection (in some cases the nursery size can increase).

The information below is only valid for old collections:

- Nursery Start/End Position—the starting and ending position in the memory address of nursery.
- Mark/Sweep Phase Time—the time spent in the marking and sweep phases, measured in milliseconds.
- Compacted Size—the size of the heap that has been compacted in the garbage collection.
- **Compaction Ratio**—the ratio of heap size before and after the compaction, measured in percent.
- **Desired/Actual evacuation**—the desired evacuation is the size of the area on the Java heap that you want to evacuate and the actual evacuation is the size of the area that JRockit managed to evacuate. The value for actual evacuation can be smaller than the desired due to temporarily pinned objects (objects that are not allowed to be moved during garbage collection). The evacuation takes place during compaction or shrinking of the Java heap.
- GC Reason—indicates the reason for doing this garbage collection.

Viewing Information on the Cache Lists Tab (Only valid for old collections)

The **Cache Lists** tab (see Figure 6-10) the specification for the different cache lists. Each cache list contains settings for upper and lower cache size.

Figure 6-10 The Cache Lists tab

General	GC	method call	tree Old Co	llection	he Lists Pa	use time
Index		# free	Cache	Avg. f	Low limit	High limit
0		0	0 bytes	0 bytes	2.000 kB	8.000 kB
1		0	0 bytes	0 bytes	8.000 kB	64.000
2		0	0 bytes	0 bytes	64.000	512.00

- Index—this is the identification number for the cache list.
- #free blocks—the number of free blocks in the cache list.
- Cache size—the total size of this cache list.
- Avg free block size—the average size of each free memory block in the cache list.
- Low limit—the lower limit of a free memory block. There will be no smaller memory block than this in the selected cache list.
- **High limit**—the upper limit of a free memory block. There will be no larger memory blocks than this in the selected cache list.

The Pause Time Tab

The information under the **Pause Time** tab is mainly intended for BEA JRockit internal use when you have sent a JRA recording for analysis to the JRockit engineering team.

General GC method call tree Old Colle	ection Cache Lis	ts Pause time	
GC pause	Length [ms]	Start [ms]	End [ms]
🗏 🚊 OC:Main	145.675	1,158,83	1,158,83
Mark:ClassRoots	0.516	1,158,83	1,158,83
🕔 Mark:ThreadRoots	0.214	1,158,83	1,158,83
Mark:HandleRoots	0.54	1,158,83	1,158,83
🕔 Mark:Objects	126.416	1,158,83	1,158,83
Mark:ReferenceQueues	0.816	1,158,83	1,158,83
🕔 Sweep:Sweep	13.762	1,158,83	1,158,83
Compaction:External	0.81	1,158,83	1,158,83
Compaction:UpdateRefer	er 0.232	1,158,83	1,158,83
🗉 🗧 OC:Main	145.675	1,158,83	1,158,83

• GC Pause—this column displays the names of the pauses (the main entry in the tree structure). If you are running a parallel garbage collector, then there will only be one pause per garbage collection. For the concurrent garbage collector, there can be several pauses during one garbage collection. The pauses consists of pause parts that can help the JRockit engineering staff to analyze why certain pauses are longer than others.

Note: During a pause, the application is standing still.

Garbage Collection Events Information

- Length—this is the length, measured in milliseconds, of the pause.
- Start/End—this is the absolute time, measured in milliseconds, since January 1, 1970.



General Garbage Collector Information

The **GC General** tab shows an overview of information about all garbage collections (GC) that took place during the recording. The information includes, amongst other, the total number of pause times and when and how the garbage collector has changed strategy.

This section is divided into the following topics:

- Getting Familiar with the GC General Tab
- Viewing General Garbage Collection Information
- Viewing Garbage Collection Call Tree Information
- Viewing Garbage Collection Strategy Changes Information

Getting Familiar with the GC General Tab

The GC General tab (see Figure 7-1) is divided into three sections that gives you information about the garbage collection at a glance.

GC General	-					
General	(1)					0
General garbage collection	information					
Total no of garbage collect	ions:	1,538				
Number of young collection	ns:	1,526				
Number of old collections:	Ē	12				
Total pause time:		3.683 s				
Average old collection pau	se time:	109 ms				
Garbage collection call	tree (2)					0
All call traces sampled durin (a) (1) 100.00% (1,538) (1,531) jrockit (1,512) jrockit	ng garbage col Garbage Coller .vm.Allocator.r ockit.vm.Alloca	ection tion hativeGetNewTLA tor.getNewTLAAn	(int,int) dAlloc(int,int,int)			
All call traces sampled durin (a) (100.00% [1,538] (b) (5,152] yrodk (c) (5,152] yrodk (c) (5,152] yrodk (c) (5,152] yrodk.ym. (c) (c) (c) (c) (c) (c) (c) (c) (c) (c)	ng garbage colle Garbage Collee .vm.Allocator Allocator.alloct dit.vm.Allocator. Allocator.alloct dit.vm.Allocato	ection tion hativeGetNewTLAin allocObject(int) (C argeArray(int), int) r. allocArray(int, int)	(int,int) dAlloc(int,int,int) PT) ; ;			
All call traces sampled durin (a) (100.00% [1,538] (b) (5,151) rodsk (c) (c) (c) (c) (c) (c) (c) (c) (c) (c)	ng garbage coll Garbage Collec .vm.Allocator.i .vm.Allocator. Allocator.alloci it.vm.Allocator dit.vm.Allocator .vm.Allocator .vm.Allocator	ection tion nativeGetNewTLAAn tor.getNewTLAAn allocObject(int) (C argeArray(int,int) r.allocArray(int,int)	(int,int) dAlloc(int,int,int,int) PT) ;)			
All call traces sampled durin all 100.00% [1,533] b [1,53] yodk b [1,53] yodk c [1,5] yodk. c [1,5] yodk. c strategy changes Garbage collection stratege Time New Strat	ng garbage coll Garbage Colles .vm.Allocator. .vm.Allocator. Allocator.alloct it.vm.Allocator .allocator.alloct it.vm.Allocato Gener	ection tion hatveGeNewTLAAn acr.getNewTLAAn allocObject(int) (c) argeArray(int,int) . allocArray(int,int) * Mark Phase	(int,int) (int,int,int,int,int))PT))) Sweep Phase	GC Prio	Needs Nur	
All call traces sampled durin (a) 100.00% [1,538] (b) 6 [1,52] yr (c) 6 [19] yroldt (c) 6 [19] yroldt (c) 6 [19] yroldt (c) 7 [17] yroldt.vm. (c) 6 [266] yrold (c) 8 [266] yrol	ng garbage cole Garbage Cole .vm.Allocator Allocator Allocator Allocator Mallocator Allocator (3) y changes al Gener No	ection tion hativeGetNewTLAin allocObject(int) (c .argeArray(int,int) .allocArray(int,int) Mark Phase Parallel	(int,int) dAlloc(int,int,int,int) PT))) Sweep Phase Parallel	GC Prio Throughput	Needs Nur Yes	Low M true

Figure 7-1 The GC General tab

The GC General tab is divided into the following sections:

- 1. **General**—this section shows overall statistics about the garbage collections during the entire JRA recording.
- 2. Garbage Collection Call Tree—this section is a collection of all call traces that were sampled for all garbage collections for the JRA recording.
- 3. GC Strategy Changes—this section lists when a garbage collection strategy took place and how it changed.

Viewing General Garbage Collection Information

The **General** section (see Figure 7-2) shows general garbage collection information such as the total number of garbage collections during the recording and the duration of all pause times. You can use this information to, for example, see whether your application is coming down to desired pause time averages or not.



0
1,538
1,526
12
3.683 s
109 ms

Viewing Garbage Collection Call Tree Information

The **Garbage Collection Call Tree** section (see Figure 7-3) shows all call traces during the recording that triggered a garbage collection. The number within the brackets (next to the garbage bin icon) is the total number of garbage collection rounds that were performed during the JRA recording. Expand the call tree to see in which methods the garbage collection has taken place.

Figure 7-3 Garbage Collection Call Tree Information



Viewing Garbage Collection Strategy Changes Information

The Garbage Collection Strategy Changes section (see Figure 7-4) shows when the garbage collector has changed strategy, for example, JRockit has been set to run for best throughput (-Xgcprio:throughput, GC Prio in Figure 7-4), then JRockit changes strategy in runtime to best reach this goal (New Strategy). The strategy change can, for example, be from singleParPar to genParPar. The strategy changes are listed under New Strategy. The old strategies are listed under Generational, Mark Phase, and Sweep Phase.

Note: These strategy changes only happen if you are running JRockit with the default garbage collector option, -Xgcprio.

Figure 7-4 Garbage Collection Strategy Changes Information

Gc strategy	changes				
Garbage colle	tion strategy changes				
Filter column	Time	*			
Time	New Strategy	Generational	Mark Phase	Sweep Phase	GC Prio
14.297 s	genParPar	No	Parallel	Parallel	Throughput

In the example seen in Figure 7-4, there has been one strategy change for the garbage collector.



Java Heap Content Information

The **Heap Overview** tab gives a quick overview of what the memory in the Java heap consists of in you application. You get a quick overview of how the heap looked at the end of the recording and also compiled information about the status of the heap during the entire recording.

This section contains the following topics:

- Getting Familiar with the Heap Overview Tab
- Viewing the Heap Snapshot at the End of the Recording Information
- Viewing the Heap Contents Information
- Viewing the Free Memory Contribution Information

Getting Familiar with the Heap Overview Tab

The Heap Overview tab is divided into three sections (see Figure 8-1).



🚫 recording.jra 🗙			- 8
Heap Overview			
Heap Snapshot at the end of the record	ling		
Snapshot of the heap at the end of the of th	e recording.		
Committed heap size: (1)	64.000 MB		
Memory in large object chunks:	2.870 MB		
Memory in pinned object chunks:	8.016 kB		
Number of pinned object chunks:	1		
Number of unused free blocks (dark matter):	20,775		
Average dark matter block size:	269 bytes		
Heap Contents	□ ?	Free Memory Contributi	on 🔲 🕐
		3	
Content Percent	age 🔼 📩	Block type	Percen Memory
Large object chunks 4.4	7%	Small, 2K to 8K	15.75% 8.324 MB
Small object chunks 4.6	0%	Medium, 8K to 64K	10.64% 5.625 MB
Dark matter 8.3	2%	Large, 64K to 512K	39.28% 20.765
	170	very large, > 512K	34.33% 18.149
General Methods GC Events GC General He	ap Overview	Object Statistics Optimizations	Lock Profiling Comments

The Heap Overview tab is divided into the following sections:

- 1. Heap Snapshot at the End of the Recording—this section contains all the specifics about your heap at a glance.
- 2. **Heap Contents**—this graph gives a visual overview of the distribution of different sizes of objects. The table below the graph gives the exact data for each category of memory.
- 3. Free Memory Contribution—this graph gives a visual overview of the distribution of the different chunks of free memory that there is on the heap. The table below the graph gives the exact data for each category of memory.

Viewing the Heap Snapshot at the End of the Recording Information

When the JRA stops recording, it calculates the value of the committed heap size, which is how much heap the application has been allowed to use. This size can be set by the -xmx flag.

The memory that is considered **large object chunks**, is the total amount of memory on the heap that the Java application is allowed to use for large objects (64 KB to 512 kB).

The memory for the **pinned object chunks** is the amount of memory that is occupied by pinned objects. A pinned object is both referenced by another object in the application and is not allowed to be moved for compaction purposes, for example, i/o buffers that are accessed from native methods (native i/o). The **number of pinned object chunks** shows a value of how many object that are pinned.

Dark matter is memory that is free, but cannot be used due to the physical layout of the memory chunk (i.e. it might be too small for the application to allocate). Dark matter can cause fragmentation on the disk.

Viewing the Heap Contents Information

The **Heap Contents** pie chart gives a graphic overview of the distribution of objects on the heap. The color coding helps you determine how much of the heap that consists of large, small, and pinned object chunks as well as how much memory is considered dark and how much is free. The amount of dark matter indicates how much space in the Java heap that is wasted due to fragmentation of the Java heap. It is normal to have a certain amount of dark matter on the heap.

For information on how to minimize the dark matter, see Minimize Dark Matter in the BEA JRockit Configuration and Tuning Guide.

Below the chart, there is a table that lists all objects with the exact data: memory in MB and percentage that they occupy of the heap.

Viewing the Free Memory Contribution Information

The **Free Memory Contribution** pie chart gives a graphic overview of how the free memory is distributed in free blocks of different sizes on the Java heap. The block sizes are categorized by the following entities: small, medium, large, and very large. The block sizes are multiples of the minimum block size set at startup (default 2kB). You set the minimum block size with the option -XXminblocksize. Below are the multiples used for the different block sizes:

- Small: 1-4
- Medium: 4–32
- Large: 32-256
- Very large: 256 and up

Java Heap Content Information



Object Statistics Information

The **Object Statistics** tab (see Figure 9-1) displays the most common types and classes occupying the Java heap at the beginning and at the end of the JRA recording.

This section is divided into the following topics:

- Getting Familiar with the Object Statistics Tab
- Viewing Start of Recording Information
- Viewing End of Recording Information

Getting Familiar with the Object Statistics Tab

At the beginning and 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 which instances in total occupy the most memory. The results are shown on the **Object Statistics tab** (see Figure 9-1). Abnormal results in the object statistics might help you detect the existence of a memory leak in your application.

1 1 M 10 11 11				
oject Statistics				
Start of recording - most c	ommon types in the h	eap		
ynat occupy more than 0.5%	6 of the used heap space.			
Iter column Class Name	*			
Class Name		% of Used Heap	#Instances	Total Siz
char[]		25.92%	1,052,871	73.000 M
java.lang.String		8.01%	985,823	22.564 M
java.lang.String[]		2.83%	148,297	7.974 M
spec.jbb.Stock		2.01%	148,588	5.668 M
spec.jbb.Orderline		1.65%	76,316	4.658 M
java.math.BigDecimal		1.30%	120,260	3.670 M
java.lang.Integer		0.92%	169,410	2.585 M
java.util.HashMap.Entry		0.67%	82,527 180 20,045	1.889 M 1.176 M 783.008 k
java.util.HashMap.Entry[]		0.42%		
java.math.BigInteger		0.27%		
spec.jbb.Orderline[]		0.27%	8,216	770.906 k
End of recording - most cor	nmon types in the hea	0.27%	8,216	770.906 k
End of recording - most cor at occupy more than 0.5% terrorumn Class Name	nmon types in the hea 6 of the used heap space 7 *	0.27%	8,216	770.906 k
End of recording - most cor and of recording - most cor and of recording - most cor and class Name Class Name	nmon types in the hea 6 of the used heap space. * * % of Used Heap	0.27%	8,216 Total Size	Difference Siz
End of recording - most cor Case to occupy more than 0.5% ter count Class Name Class Name char[]	where the search of the used heap space.	0.27%	8,216 Total Size 73.000 MB	Difference Siz
End of recording - most cor to coupy more than 0.5% to coupy more than 0.5% Class Name Class Name char[] iva-Jang.String	where the sed heap space.	0.27% #Instances 1,052,871 965,823	8,216 Total Size 73.000 MB 22.564 MB	770.906 kl
End of recording - most cor to coupy more than 0.5% to coupy more than 0.5% to coupy more than 0.5% to class Name (class Name thar[] ava.lang.String]	* * * * * * * * * * * * * * * * * * *	0.27% #Instances 1,052,871 965,823 148,297	8,216 Total Size 73.000 MB 22.564 MB 7.974 MB	Difference Siz -645528 byte +191.484 ki +6.664 ki
End of recording - most cor end of recording	* * % of Used Heap 25.92% 8.01% 2.83% 2.01% 2.01%	0.27% #Instances 1,052,871 965,823 148,297 148,588	8,216 Total Size 73.000 MB 22.554 MB 7.974 MB 5.668 MB	770.906 k
End of recording - most cor and of recording - most cor and a coupy more than 0.5% ter outurn [lass Name] char[] ava.lang.String] ava.lang.String] spec.jbb.Stock spec.jbb.Stock	* % of the used heap space. * % of Used Heap 25,92% 8.01% 2.83% 2.01% 1.65%	0.27% #Instances 1,052,871 965,823 148,297 148,588 76,316	8,216 Total Size 73,000 MB 22,564 MB 7,974 MB 5,668 MB 4,658 MB	770.906 k
End of recording - most cor end of recording	* * % of Used Heap 25.92% 8.01% 2.83% 2.83% 2.01% 1.65% 1.30%	0.27% #Instances 1,052,871 965,823 148,297 148,588 76,316 120,260	8,216 Total Size 73,000 MB 22,564 MB 5,568 MB 4,658 MB 3,670 MB	270.906 k
End of recording - most cor 2 of occupy more than 0.5% the coupy more than 0.5% the coupy more than 0.5% Class Name char[] java.lang.String] java.lang.String] java.lang.String] java.lang.String] java.lang.theger java.lang.theger	**************************************	0.27% #Instances 1,052,871 995,823 140,297 140,588 76,316 120,260 169,410	8,216 Total Size 73.000 MB 22.564 MB 5.668 MB 4.658 MB 3.670 MB 2.5855 MB	770.906 k
End of recording - most cor Class Name char[] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String] yev_lang_String]	mmon types in the head % of the used heap space. ✓ * * % of Used Heap 25,92% 8.01% 2.83% 2.01% 1.65% 0.62% 0.67%	0.27% #Instances 1,052,671 965,823 148,598 76,316 120,260 169,410 82,527	8,216 Total Size 73.000 MB 22.554 MB 5.668 MB 4.658 MB 3.670 MB 2.585 MB 1.889 MB	770.906 k
End of recording - most cor tere of a coupy more than 0.5% tere of the coupy more than 0.5% tere of the coupy more than 0.5% tere of the coupy more than 0.5% Class Name Class Name Class Name Class Name (Class Name (Class Name (Class Name) (Class Name) (C	* % of Used Heap % of Used Heap % of Used Heap % of Used Heap 25.92% 8.01% 2.83% 1.65% 1.30% 0.67% 0.42%	0.27% #Instances 1,052,871 995,803 148,297 148,288 76,316 120,260 169,410 82,527 180	8,216 Total Size 73,000 MB 22.564 MB 7.974 MB 5.668 MB 4.658 MB 3.670 MB 2.585 MB 1.689 MB 1.176 MB	2770.906 k Difference Siz -645528 byte +191.484 k +174.219 k +174.219 k +29.688 k +98.391 k -9048 byte +80 byte
End of recording - most cor Part occupy more than 0.59 tercourm Class Name char[] java.lang.String] java.lang.String] java.lang.String] java.lang.String] java.lang.String] java.lang.String] java.lang.String] java.lang.Integer java.utl.HashMap.Entry] java.utl.HashMap.Entry] java.utl.HashMap.Entry]	mmon types in the head work of the used heap space. w * * % of Used Heap 25.92% 6.01% 2.83% 6.01% 2.01% 1.65% 0.62% 0.67% 0.42% 0.42%	0.27% #Instances 1,052,871 965,823 148,287 148,258 76,316 120,260 169,410 82,527 180 20,045	8,216 Total Size 73.000 MB 22.564 MB 5.668 MB 3.670 MB 2.585 MB 1.689 MB 1.176 MB 783.008 kB	770.906 ki Difference Size -645528 byte +191.484 ki +29.688 ki +29.688 ki +99.391 ki +99.391 ki +99.391 ki +30 kbyte +440.052 ki

Figure 9-1 The Object Statistics tab

The **Object Statistics** tab is divided into the following sections:

- 1. **Start of Recording**—this section lists the most common types on the heap at the beginning of the recording.
- 2. End of Recording—this section lists the most common types on the heap at the end of the recording.

Viewing Start of Recording Information

When the JRA starts a recording it looks at the Java heap to see which types occupy the most memory in the used heap space. That information is listed under the **Start of Recording** section (see Figure 9-2).

Figure 9	1-2	Start	of	Recording	section
----------	-----	-------	----	-----------	---------

Start of recording - most common types in the heap Types that occupy more than 0.5% of the used heap space. Filter column Class Name Y *						
Class Name	% of Used Heap	#Instances	Total Size			
char[]	25.92%	1,052,871	73.000 MB			
java.lang.String	8.01%	985,823	22.564 MB			
java.lang.String[]	2.83%	148,297	7.974 ME			
spec.jbb.Stock	2.01%	148,588	5.668 MB			
spec.jbb.Orderline	1.65%	76,316	4.658 MB			
java.math.BigDecimal	1.30%	120,260	3.670 ME			
java.lang.Integer	0.92%	169,410	2,585 MB			
java.util.HashMap.Entry	0.67%	82,527	1.889 ME			
java.util.HashMap.Entry[]	0.42%	180	1.176 ME			
java.math.BigInteger	0.27%	20,045	783.008 kE			
spec ibb Orderline[]	0.27%	8 216	770, 906 kB			

Viewing End of Recording Information

Right before the JRA stops a recording it looks at the Java heap to see which types occupy the most memory in the used heap space. That information is listed under the **End of Recording** section (see Figure 9-3).

Figure 9-3 End of Recording section

 End of recording - most common 	n types in the heap			
Types that occupy more than 0.5% of t	he used heap space.			
Filter column 🛛 Class Name 🛛 👻 🔭	:			
Class Name	% of Used Heap	#Instances	Total Size	Difference Size
char[]	25.92%	1,052,871	73.000 MB	-645528 bytes
java.lang.String	8.01%	985,823	22.564 MB	+191.484 kB
java.lang.String[]	2.83%	148,297	7.974 MB	+6.664 kB
spec.jbb.Stock	2.01%	148,588	5.668 MB	+174.219 kB
spec.jbb.Orderline	1.65%	76,316	4.658 MB	+29.688 kB
java.math.BigDecimal	1.30%	120,260	3.670 MB	+89.281 kB
java.lang.Integer	0.92%	169,410	2.585 MB	+98.391 kB
java.util.HashMap.Entry	0.67%	82,527	1.889 MB	-9048 bytes
java.util.HashMap.Entry[]	0.42%	180	1.176 MB	+80 bytes
java.math.BigInteger	0.27%	20,045	783.008 kB	+44.062 kB
spec.jbb.Orderline[]	0.27%	8,216	770.906 kB	+2.156 kB

Object Statistics Information



Code Optimization Information

The **Optimizations** tab (see Figure 10-1) displays the methods that were optimized by the adaptive optimization system in JRockit during the recording.

This section is divided into the following topics:

- Getting Familiar with the Optimizations Tab
- Viewing Optimization Information
- Viewing Methods Optimized During Recording Information

Getting Familiar with the Optimizations Tab

The JRA records all optimization events that occur during the course of the recording. JRockit uses JIT compilation for the initial conversion to machine code. The most commonly used methods are then further optimized during the application run. This information is then displayed in the Optimizations tab (see Figure 10-1).



Optimizat				
	ion			C
(1)				
\mathbf{U}		Refore sampling Af	ter sampling	
Number of o	ntimizations:		or	
	permeasurement	1/1	00	
Time spent o	ptimizing:	19455 2	0127	
Number of J	IT-compilations:	4.268	.268	
Time coept 1	IT-compling methods:			
nine spendu	ar-compling methods:	1.344 s	.344 s	
Methods	Optimized During R	ecording		
2)				
ter column	Index V *			
cor condition				
Index	Method		Size Before	Size Afte
1	java.lang.String.e	quals(Object)	201 bytes	191 byte
2	java.util.TreeMap	\$Entry.getValue()	9 bytes	12 byte
3	spec.jbb.Delivery	Transaction.setupDeliveryLog()	205 bytes	2.262
4	spec.jbb.StockLev	elTransaction.init()	125 bytes	3.850
5	java.util.Date.get	Time()	11 bytes	733 byte
6	java.util.TreeMap	\$SubMapEntryIterator.next()	78 bytes	269 byte
7	java.math.BigDec	mal.negate()	106 bytes	1.281
8	spec.jbb.StockLev	elTransaction.initializeTransactionLog()	122 bytes	4.196
0	spec.jbb.Delivery	Transaction.init()	120 bytes	3.818 k
2	java.math.Mutabl	eBigInteger.divideOneWord(int,MutableBigIn	1.140 kB	1.210
10	java.math.Mutabl	eBigInteger.divide(MutableBigInteger,Mutab	2.233 kB	3.394
10 11		.values()	69 bytes	120 byte
10 11 12	java.util.TreeMap	And a set of the standard blad and a set	208 bytes	117 byte
10 11 12 13	java.util.TreeMap jrockit.vm.Reflect	\$LCIassBlock.isAssignable(long,long)		
10 11 12 13 14	java.util.TreeMap jrockit.vm.Reflect java.math.BigInte	ger.divide(BigInteger)	197 bytes	5.459 k
10 11 12 13 14 15	java.util.TreeMap jrockit.vm.Reflect java.math.BigInte com.sun.org.apac	stClassBlock.isAssignable(long,long) ger.divide(BigInteger) he.xerces.internal.dom.DocumentImpl.modil	197 bytes 24 bytes	5.459 k 246 byte

The **Optimizations** tab is divided into the following sections:

- 1. **Optimization**—this section displays the before and after scenario of the optimizations that have taken place.
- 2. **Methods Optimized During Recording**—this section displays which methods that have been optimized during the recording, i.e. this is necessarily not a full list of all optimizations that are performed for your application.

Viewing Optimization Information

The **Optimizations** section (see Figure 10-2) contains information on how many optimizations have taken place and the total duration of the optimizations. You can also see how many JIT compilations have been performed and the time JRockit took to compile those. For more information on JIT compilation, see the Introduction to BEA JRockit JDK.

Figure 10-2 Optimization section

Optimization			(
	Before sampling	After sampling	
Number of optimizations:	171	186	
Time spent optimizing:	19455	20127	
Number of JIT-compilations:	4,268	4,268	
Time spent JIT-compiling methods:	1.344 s	1.344 s	

Viewing Methods Optimized During Recording Information

The **Methods Optimized During Recording** section (see Figure 10-3) lists all methods that were optimized during the JRA recording. Here you can study the size changes of each method that has been optimized.

Note: Some optimizations, such as inlining, causes the method size to increase

Figure 10-3 Methods Optimized During Recording section

 Methods 	Optimized During Recording		
Filter column	Index 💌 *		
Index	Method	Size Before	Size Afte
1	java.lang.String.equals(Object)	201 bytes	191 byte
2	java.util.TreeMap\$Entry.getValue()	9 bytes	12 byte
3	spec.jbb.DeliveryTransaction.setupDeliveryLog()	205 bytes	2.262 k
4	spec.jbb.StockLevelTransaction.init()	125 bytes	3.850 k
5	java.util.Date.getTime()	11 bytes	733 byte
6	java.util.TreeMap\$SubMapEntryIterator.next()	78 bytes	269 byte
7	java.math.BigDecimal.negate()	106 bytes	1.281
8	spec.ibb.StockLevelTransaction.initializeTransactionLog()	122 bytes	4.196
9	spec.jbb.DelivervTransaction.init()	120 bytes	3.818.

Code Optimization Information



Lock Profiling Information

The **Lock Profiling tab** (see Figure 11-1) shows comprehensive information about lock activity for both the application JRA is monitoring (Java locks) and JRockit itself (native locks). You need to enable the recording capability before you start the profiling of your application. If you have not enabled the lock profiling data recording, the lock profiling sections are left blank on the **Lock Profiling** tab. For more information on locks, please refer to About Thin, Fat, Recursive, and Contended Locks in BEA JRockit.

This section is divided into the following topics:

- Getting Familiar with the Lock Profiling Tab
- Java Locks Profiling
- Enabling Java Lock Profiling Data
- Native Lock Profiling
- Enabling Native Locks Information

Getting Familiar with the Lock Profiling Tab

The **Lock Profiling** tab displays lock information for both your application and JRockit (see Figure 11-1).

Java Locks (1)							
f no information is displaye	ed here, you sho	uld enable lock pr	rofiling in JRockit	with -Djrockit.lock	profiling=true ar	nd d	0
nother recording.							
ilter column Class	*						
Class	Fat Cont.	Fat Uncont.	Fat Recursiv	e Thin Unco	nt. Thin Co	nt.	
java.lang.Object	0	118		0	94	0	
javax.management	0	0		0	1	0	
java.net.Datagram	0	0		0	20	0	
java.net.Socket	0	0		0	8	0	
java.net.SocksSock	0	0		0	6	0	
sun.rmi.transport.T	0	0		0	7	0	
com.sun.jmx.interc	0	0		0	2	0	
jrockit.management	0	0		0	3	0	
javax.management	0	0		0	1	0	
java.util.logging.Lo	0	0		0	2	0	
Native Locks (2)	0	0		0	4	0	
iava.util.LinkedList Native Locks 2 f no information is displays ilter column Lock Name	o ed here, you sho	0 uld enable native	profiling in the S	0 tart JRA Recordin	4 g wizard.	0	
Native Locks 2 f no information is displaye ilter column Lock Name	o ed here, you sho v	0 uld enable native	profiling in the S Times Acquired	0 tart JRA Recordin Times Conten	g wizard.	0	
iava.util.LinkedList Native Locks f no information is displaye iter column Lock Name Bootstran loader (19-0065	o ed here, you sho *	0 uld enable native	profiling in the S Times Acquired	0 tart JRA Recordin	g wizard.	0 III	
iava.util.LinkedList Native Locks 2 f no information is displays ilter column Lock Name Lock Name Bootstrap Loader (0x:0062 Revelancins (0x:00526E1)	o ed here, you sho v *	0 uld enable native	profiling in the S Times Acquired 4 23	0 tart JRA Recordin Times Conten	g wizard.	0 led 0	
iava.util.LinkedList Native Locks f no information is displays iter column Lock Name Bootstrap loader (0x0062 Breakpoints (0x00504E11 CG Thread Worklist (0x01)	0 ed here, you sho (* (CSF4))) (SE4340)	0 uld enable native	profiling in the S Times Acquired 4 23 0	0 tart JRA Recordin Times Conten C	g wizard.	0 [] [] [] [] [] [] [] [] [] [] [] [] []	
iava.util.LinkedList Native Locks 2 f no information is displays ther column Lock Name Lock Name Bootstrap loader (0x0062 Breakpoints (0x005D4E11 CG Thread Worklist (0x00 Class library: Cache (0m)	0 ed here, you sho (CSF4))) (SE4340) (SE2798)	0 uld enable native	profiling in the S Times Acquired 4 23 0 56	0 tart JRA Recordin Times Conten C C C	g wizard.	0 led 0 0 0	
iava.util.LinkedList Native Locks C fino information is displays liter column Lock Name Bootstrap loader (0x0066 Breakpoints (0x00504E1 CG Thread Worklat (0x0 Loss litrary: Cache (0x0 Loss litrary	0 ed here, you sho (CSF4))) (SE4340) (DSE2798) (DSE2798) (DSE2688)	0 uld enable native	profiling in the S Times Acquired 4 23 0 56 20	o tart JRA Recordin Times Conten C C C C C C	g wizard.	0 led 0 0 0	
iava.util.LinkedList Native Locks no information is displays inter column Lock Name Bootstrap loader (0x:0063 Breekpoints (0x:0050HE11 Gras library: Free list (0x Class library: Class field Class library: Class field	0 ed here, you sho v * (CSF4) 0) (S5E4340) 005E2798) 005E2678) with (by005D08C	0 uld enable native	profiling in the S Times Acquired 4 23 0 56 20 0	0 tart JRA Recordin Times Conten C C C C C C C C C C C C C C C C C C	g wizard.	0 led 0 0 0 0 0	
iava.util.LinkedList Native Locks To information is display. Rer colum Lock Name Lock Name Bootstrap loader (0x0056 Breakpoints (0x0050FeII) Cass library: Cache (0x0 Cass library: Cache (0x0 Cade Memory: Class GC)	0 ed here, you sho (CSF4) 0) (CSF4) 0) 005E2798) 005E2678) st (0:005D086C st (0:005D086C	0 uld enable native	profiling in the S Times Acquired 4 23 0 56 20 20 0 0 0	0 tart JRA Recordin Times Conten C C C C C C C C C C C C C C C C C C	g wizard.	0 led 0 0 0 0 0 0	
iava.util.LinkedList Native Locks C fino information is displays fino information is displays filter column Lock Name Bootstrap loader (0x:0050/4E11 CG Thread Worklat (0x:00 Gass library: Free lst (0x:006 Memory: Gass GE1 Code Memory: Free lst (0x:006 Memory: Free lst (0x:006 Memory: Free lst)(0x:006 Memory: Free lst)(0	0 ed here, you sho (CSF4))) (CSF4))) (SE4340) (DSE2798) (DSE2678) ist (0x005DD66C) phore (0x005DC	0 uld enable native	Times Acquired 4 23 0 566 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 tart JRA Recordin Times Conten C C C C C C C C C C C C C C C C C C	g wizard.	0 led 0 0 0 0 0 0 0 0 0 0	
iava.util.LinkedList Native Locks 2 in o information is display. Iter colum Lock Name Lock Name Bootstrap loader (0x0056/EEI Breakpoints (0x0050/EEI Col Thread Worklist (0x00 Cass library: Cache (0x0) Cass library: Free list (0x Code Memory: Free Set (0x) Code Memory: Free Set (0x) Code Memory: Set (0x) Code Memory: Set (0x)	0 ed here, you sho v (CSF4) 1) 15E4340) 15E2798) 10552678) 105526780 10552066C) 10055D06C) 10055D06C)	0 uld enable native	profiling in the S Times Acquired 4 23 0 56 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 tart JRA Recordin Times Conten C C C C C C C C C C C C C C C C C	g wizard.	0 led 0 0 0 0 0 0 0 0 0 0 0 0 0	
Iava.util.LinkedList Native Locks f no information is displaye ther column Lock Name Bootstrap loader (0x0056 Breakpoints (0x00504E1) Gas library: Free list (0x Code Memory: Gal tat (0x) Code Memory: Gal tat (0x)	0 ad here, you sho CSF4) 0) 552343(0) 05522793) 005522678) st (0:x005D06C) 005D004C) 005D004C)	0 uld enable native	profiling in the S Times Acquired 4 23 0 56 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 tart JRA Recordin Times Conten C C C C C C C C C C C C C C C C C C	g wizard.	0 iled 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	
iava.util.LinkedList Native Locks 2 Fro information is display. Iter column Lock Name Lock Name Bootstrap loader (0x0056/E1 Co Thread Worklot (0x00 Cas library: Cache (0x0 Cas library: Free lst (0) Code Memory: Free lst (0) Code Memory: Free Sen Code Memory: Free Sen Code Memory: Released	0 ad here, you sho (CSF4))) (SSE340) (SS	0 uld enable native	r profiling in the S Times Acquired 4 23 0 0 56 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 tart JRA Recordin Times Conten C C C C C C C C C C C C C C C C C C	g wizard.	0 led 0 0 0 0 0 0 0 0 0 0 0 0 0	

Figure 11-1 Lock Profiling tab

The Lock Profiling tab is divided into the following sections:

- 1. Java Locks-this section lists all locks in your application.
- 2. Native Locks—this section lists all locks in JRockit.

Java Locks Profiling

The information that is displayed under the **Java Locks** chart (see Figure 11-2) shows the number of locks of the threads in your application. You see information on the number of fat uncontended and contended locks, thin uncontended and contended locks, thin and fat recursive locks, and fat sleeping locks.

Figure 11-2 Java Locks

🔻 Java Locks							
If no information is displayed recording.	d here, you sha	ould enable lock	cprofiling in JRc	ickit with -Djroc	kit.lockprofiling=	true and do an	other
Filter column Class	*						
Class	Fat Cont.	Fat Uncont.	Fat Recu	Thin Unc	Thin Cont.	Thin Rec	Fat Sleep
java.lang.Object	0	60	0	1	0	0	0
java.io.OutputStream	0	0	0	366	0	122	0
java.io.BufferedOutp	0	0	0	305	0	0	0
java.io.PrintStream	0	0	0	61	0	244	0
	Anne and		adaa a		a sea and		

Enabling Java Lock Profiling Data

To record Java lock profiling data, you need to enable it from the command line when you start JRockit. If your the Java Locks section is blank, it is not enabled.

To enable Java lock profiling data

• Issue the command -Djrockit.lockprofiling at the JRockit command line.

For example:

```
java -Djrockit.lockprofiling=true -XXjra:<AnyJRAParam> -jar MyApplication.jar
```

Native Lock Profiling

If you are looking at a recording of JRockit J2SE 5.0 or later, the recording includes information about native locks (see Figure 11-3). Native locks are locks in the JRockit internal code and is nothing your application can control.

Figure 11-3 Native Locks

 Native Locks 			
If no information is displayed here, you should enable nativ	e profiling in the Start JRA I	Recording wizard.	
Filter column Lock Name 💉 *			
Lock Name	Times Acquired	Times Contended	Times Try Failed 🔥
Bootstrap loader (0x00637F60)	1	0	0
Breakpoints (0x005E03AC)	31	0	0
CG Thread Worklist (0x005EF988)	0	0	0
Class library: Cache (0x005EE008)	57	0	0
Class library: Free list (0x005EE04C)	12	0	0
Code Memory: Class GC list (0x005E86AC)	0	0	0
Code Memory: Free Semaphore (0x005E8598)	0	0	0
Code Memory: Free list (0x005E860C)	_ 0	0.0	. Q.
Viewand and second and the second sec	and a second	and the same state of the same	and a support of the

If you find high contention on a JRockit internal lock that might be causing issues for your application, either contact BEA support or contact JRockit through the BEA JRockit news group at the dev2dev web site.

Enabling Native Locks Information

Lock profiling data can only be generated from the command line. If you have no information displayed in the

To enable native locks profiling data

- 1. Select a JRockit from the JRockit Browser.
- 2. Click Mission Control > JRA > Start JRA Recording.
- 3. Select Use native samples.



Start and End Processes Information

The **Processes** tab (see Figure 12-1) lists which processes were running during both the start and the end of the JRA recording.

This section is divided into the following topics:

- Getting Familiar with the Processes Tab
- Snapshot of Processes at Beginning and End of Recording
- Detailed Processes Information

Getting Familiar with the Processes Tab

The Processes tab displays start and end information of running processes (see Figure 12-1).

Figure 12-1 Processes tab

he same proc ilter column	ess. Executable	~					-	
Path	Execu	utable	At start	At end		Process ID	Command Lin	e 🔼
C:\Program F	iles acrotr	ay.exe	Yes	Yes		2,872	N/A	
C:\Program F	iles Applic	ation Lau	Yes	Yes		2,860	N/A	
C:\WINDOW	S\s ctfmor	n.exe	Yes	Yes		2,832	N/A	_
C:\Program P	iles Yahoo	Messeng	Yes	Yes		2,708	N/A	
C:\Program P	iles msms	gs.exe	Yes	Yes		2,644	N/A	
C:\Program P	iles iPodSe	ervice.exe	Yes	Yes		2,616	N/A	
C:\Program F	iles gttask	.exe	Yes	Yes		2,592	N/A	
C:\PROGRA-	-1\ OUTLO	DOK.EXE	Yes	Yes		2,588	N/A	
C:\Program P	iles iTunes	sHelper.exe	Yes	Yes		2,548	N/A	
	J11 VPTra	v eve	Vec	Vec		2 432	M/A	~
rocess 2) ::\Program F	=iles\Adobe\A	dobe Acrobat 6.	0\Distillr\acrotray.	exe			
Path: Executable: At start: At end:) ::\Program f acrotray.exe Yes	Files\Adobe\A	dobe Acrobat 6.	0\Distillr\acrotray.	exe			
Path: Executable: At start: At end:) ::\Program F acrotray.exe Yes Yes	Files\Adobe\A	dobe Acrobat 6.	D\Distillr\acrotray.	exe			
Path: Executable: At start: At end: Process ID:) 	Files\Adobe\A	dobe Acrobat 6.	0\Distillr\acrotray.	exe			

The **Processes** tab is divided into the following sections:

- 1. Snapshot of the processes running on the machine at the start and at the end of the recording—this section lists all processes that were active either during the start or the end of the recording or both.
- 2. Process—this section details the processes information.

Snapshot of Processes at Beginning and End of Recording

The information that is displayed under the **Snapshot** view (see Figure 12-2) shows all processes that were running at the start of the recording and at the end of the recording.

Figure 12-2 Snapshot view

 Snapshot of the 	processes runn	ing on the mac	hine at the start and	the end of the reco	rding.
Processes with the sa the same process.	ame pid, path, exe	cutable name and	command line at the sta	art and end of the reco	rding are assumeq
Filter column Execu	table 🔽				
Path	Executable	At start	At end	Process ID	Command Line
C:\WINDOWS\S	svchost.exe	Yes	Yes	244	N/A
C:\WINDOWS\s	ati2evxx.exe	Yes	Yes	1,692	N/A
C:\WINDOWS\s	ati2evxx.exe	Yes	Yes	3,260	N/A
C:\WINDOWS\s	WISPTIS.EXE	Yes	Yes	7,816	N/A
C:\WINDOWS\s	cmd.exe	Yes	Yes	32,924	N/A
advision of the	L-Minnes I- carbona	a della secola della	and Mercury - and		Latta - Later - Later

Detailed Processes Information

When selecting a process in the **Snapshot** view, you see a listing of all details for that process at the bottom of the tab (see Figure 12-3). The path, the name of the executable, if the process was present during start and end, the process ID, and also if the process was started with a command-line option.

Figure 12-3 Detail process view

Process	
Path:	c:\program files\adobe\framemaker7.2\framemaker.exe
Executable:	FrameMaker.exe
At start:	Yes
At end:	Yes
Process ID:	30,516
Command Line	
A static second of	

Start and End Processes Information



Thread Latency Viewer Overview

The **Thread Latency Overview** tab (see Figure 13-1) displays an overview of all running threads during the recording.

This section is divided into the following topics:

- •
- .
- •
- •

Getting Familiar with the Thread Latency Overview Tab

At the beginning and 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 which instances in total occupy the most memory. The results are shown on the **Thread Latency Overview tab** (see Figure 13-1). Abnormal results in the object statistics might help you detect the existence of a memory leak in your application.

Start of r	ecording - m	ost comme	n types in the h	ean			
	ccupy more that	n 0.5% of th	e used heap space				
lter column	Class Name	*					
Class Name	e			% of Used Heap	#Instan	ces T	otal Size
char[]				25.92%	1,052,8	371 73	.000 MB
java.lang.Sl	tring			8.01%	985,8	323 22	.564 MB
java.lang.S	tring[]			2.83%	148,2	297 7	.974 MB
spec.jbb.St	ock			2.01%	148,5	588 5	.668 MB
spec.jbb.Or	rderline			1.65%	76,3	316 4	.658 MB
java.math.E	BigDecimal			1.30%	120,2	260 3	.670 MB
java.lang.Ir	nteger			0.92%	b 169,4	410 2	.585 MB
java.util.Ha	shMap.Entry			0.67%	82,5	527 1	.889 MB
	chMan Entry []			0.42%	5 1	180 1	.176 MB
java.util.Ha	isningb.cncry[]						
java.util.Ha java.math.B	BigInteger			0.27%	20,0	045 783	3.008 kB
java.util.Ha java.math.8 spec.jbb.Or	BigInteger rderline[]			0.27%	20,0 8,2	045 78 216 770	3.008 kB).906 kB
java.util.Ha java.math.E spec.jbb.Or End of ree	cording - mo:	st common n 0.5% of th	types in the hea e used heap space	0.27% 0.27% ap	5 20,0 5 8,2	045 78: 216 770	3.008 kB 0.906 kB
java.util.Ha java.math.E spec.jbb.Or End of ree yr 2 at or iter coumn	cording - mos ccupy more that Class Name	st common n 0.5% of th v	types in the hea	0.27% 0.27% ap	5 20,0 5 8,2	945 78 216 770	3.008 kB 0.906 kB
java.util.Ha java.math.E spec.jbb.Or End of red y at or ilter sumn Class Name	cording - mos ccupy more that Class Name	st common n 0.5% of th v	types in the hear e used heap space % of Used Heap	0.27% 0.27% ap #Instance	s 20,0 s 8,2 s Total S	045 78: 216 770	a.008 kB 0.906 kB
java.util.Ha java.math.E spec.jbb.Or End of rec y at or ilter dumn Class Name char[]	cording - mo: ccupy more than Class Name	st common n 0.5% of th v	types in the hea e used heap space % of Used Heap 25.92%	0.27% 0.27% ap #Instance: 1,052,87	s 20,0 5 8,2 5 Total 2 1 73.000	5/216 770 5/216 770 5/216 Differe MB -64552	3.008 kB 0.906 kB (C) (C) (C) (C) (C) (C) (C) (C) (C) (C)
java.util.Ha java.math.E spec.jbb.Or End of ree y at or ilter column Class Name char[] java.lang.S	cording - mo: ccupy more that class Name	st common n 0.5% of th v	types in the heat e used heap space % of Used Heap 25.92% 8.01%	0.27% 0.27% ap #Instance: 1,052,87 985,82	s Total 5 1 73.000 3 22.554	5125 Differe MB -64555 MB +191	3.008 kB 0.906 kB () () () () () () () () () () () () ()
java.util.Ha java.math.E spec.jbb.Or End of ree y 2 at or iter Sumn Class Name char[] java.lang.Si java.lang.Si	cording - me: coupy more that [Class Name] b tring	st common n 0.5% of th v	types in the here e used heap space % of Used Heap 25.92% 8.01% 2.83%	0.27% 0.27% ap #Instance: 1,052,877 965,827 148,29	s Total 5 1 73.000 3 22.564 7 7.974	5/22 Differe MB -64557 MB +19 MB +4	3.008 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB
java.util.Ha java.math.E spec.jbb.Or End of ree y at or ite aumn Class Name char[] java.lang.Si spec.jbb.St	cording - mo: cording - mo: cocupy more that [Class Name e tring tring[] ock	st common n 0.5% of th	types in the hea e used heap space % of Used Heap 25.92% 8.01% 2.83% 2.01%	0.27% 0.27% ap	s Total 5 1 73.000 3 22.564 7 .7.974 3 5.668	5/216 77(5/216 77(5/216)	3.008 kB 0.906 kB
Find of rev prove the second	cording - mo: cording - mo: ccupy more that [class Name e tring tring[] ock crderline	st common n 0.5% of th v	types in the her e used heap space % of Used Heap 25.92% 8.01% 2.83% 2.01% 1.65%	0.27% 0.27% ap #Instance: 1,052,87 965,82 148,29 148,29 148,29 148,29 148,29	s Total 5 1 73.000 3 22.564 7 7.974 5 4.658 5 4.658	5/22 Differe MB -64555 MB +199 MB +17 MB +12 MB +22 MB +22	3.008 kB 0.906 kB 0.906 kB 0.906 kB 0.006 kB 0.006 kB 0.664 kB 0.664 kB 0.668 kB
End of ree y at an ath.f. spec.jbb.Or End of ree y at or iter cours class Name char[] java.lang.Si java.lang.Si spec.jbb.St spec.jbb.St	cording - mo: coupy more that class Name e tring tring cock define BigDecimal	st common h 0.5% of th v	types in the hea e used heap space % of Used Heap 25.92% 8.01% 2.83% 2.01% 1.65% 1.30%	0.27% 0.27% ap	s Total 2 1 73.000 3 22.564 7 7.974 3 5.668 5 4.658 0 3.670	Size Differe MB -6455; MB +19 MB +17* MB +8 MB +88	3.008 kB 0.906 kB 0.906 kB 0.906 kB 0.000 kB 0.900 kB 0.9
iava.util.iHa java.math.f spec.jbb.Or End of red yr 2 st or iker Summ Class Name Char[] java.lang.S java.lang.S java.lang.S java.lang.s	cording - mos coupy more that [Class Name] class Name]	st common n 0.5% of th V	types in the heat % of Used Heap 25.92% 8.01% 2.01% 1.65% 0.92%	0,27% 0.27% ap	5 Total 5 4 73.000 3 22.564 5 4.658 5 4.658 0 3.670 0 2.595 4 658	5ize Differe MB -6455; MB +199 MB +17 MB +25 MB +25 MB +25 MB +25 MB +25 MB +25 MB +99 MB +99	3.008 kB 0.906 kB 0.906 kB 0.906 kB 0.000 kB 0.0000 kB 0.00000 kB 0.00000 kB 0.0000 kB 0.0000 kB 0.0000 kB 0.00
End of rec yr 2 to man and the spec signal and	cording - me: coupy more that [Class Name] colored fring] tring tring[] ock define BigDecimal integer shMap.Entry	st common n 0.5% of th v	types in the hea e used heap space % of Used Heap 25.92% 8.01% 2.63% 2.63% 0.92% 0.65%	0.27% 0.27% ap 	s Total 2 1 73.000 3 22.564 7 7.974 3 5.668 3 4.658 0 3.679 7 1.899 1 899 1 8	Size Differe MB -6455; MB +19; MB +19; MB +25; MB +8; MB +90; MB -90;	3.008 kB 0.906 kB 0.906 kB 0.906 kB 0.228 bytes 1.484 kB 5.664 kB 4.219 kB 0.281 kB 0.281 kB 0.281 kB 0.281 kB
iava.util.iha java.math.£ spec.jbb.Or End of ree y at or ite aumn Class Name char[] java.lang.S java.lang.S java.lang.s java.lang.r java.math.£ java.lang.r	cording - mos coupy more that [Class Name] e tring tring bigDecimal nteger shMap.Entry[]	st common n 0.5% of th V	types in the hea e used heap space 9% of Used Heap 25.92% 8.01% 2.83% 2.01% 1.65% 0.92% 0.62% 0.42%	0.27% 0.27% ap #Instance 1,052,67 148,253	s Total 5 s Total 5 t 73.000 3 22.564 7 7.974 5.668 5 4.658 0 3.670 0 2.565 7 1.889 0 1.176	Size Differe MB -6455; MB +199; MB +179; MB +129; MB +22; MB +38; MB +99; MB +90; MB +40;	3.008 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB 0.906 kB 0.281 kB 0.688 kB 0.688 kB 0.688 kB 0.688 kB 0.688 kB 0.9688 kB 0.96888 kB 0.96888 kB 0.9688 kB 0.9688

Figure 13-1 The Thread Latency Overview tab

The Thread Latency Overview tab is divided into the following sections:

- 1. **Start of Recording**—this section lists the most common types on the heap at the beginning of the recording.
- 2. End of Recording—this section lists the most common types on the heap at the end of the recording.

Event Graph Window Workflow From Recording to Viewing **Drilling Down to Your Problem** Expanding and Collapsing Thread Nodes **Customizing Your View Using the Filter Functions** Using the Time Scale on Top of Window Getting Familiar with the Events Table Tab About Selecting Events for "intressanta mängden" Selecting Events for "intressanta mängden" Deleting Events from "intressnta mängden"

Thread Latency Viewer Overview



Getting Familiar with the Events Table Tab

About Selecting Events for "intressanta mängden" Selecting Events for "intressanta mängden" Deleting Events from "intressnta mängden" Filtering Columns Getting Familiar with the Events Table Tab


Getting Familiar with the Stack Trace Tree Overview Tab

About the Stack Trace Tree Overview Tab Adding and Removing Content in the Tree Table Deleting Events from "intressanta mängden" Filtering Columns Getting Familiar with the Stack Trace Tree Overview Tab



Adding Comments to a Recording

The JRA tool is equipped with a small text editor where you can add comments about the recording and your application. These comments will help the BEA JRockit engineering team to understand what has happened to JRockit and your application during the recording (see Figure 16-1).





Adding Comments to a Recording

To add a comment

- 1. Enter a description of you application in the text field.
- 2. Close the JRA recording.
- 3. Click Yes, when asked if you want to save the recording.