BEA JRockit Mission Control

JRockit Runtime Analyzer User Guide

1.0
July 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Using the BEA JRockit Runtime Analyzer

Looking at a Recording

Help Us Improve BEA JRockit and the JRA Tool

CHAPTER 1

Using the BEA JRockit Runtime Analyzer

The BEA JRockit Runtime Analyzer (JRA) tool provides a wealth of information on internals in BEA JRockit that are interesting to the development team of BEA JRockit. Some of these metrics are also interesting to Java developers using BEA JRockit as their runtime VM.

Note: The JRA tool itself requires J2SE version 1.4 or higher.

The BEA JRockit Runtime Analyzer consists of two parts. One is running inside the JVM and recording information about the JVM and the Java application currently running. This information is saved to a file which is then opened in the other part: the analyzer tool. This is a regular Java application used to visualize the information contained in the JRA recording file.

This section describes how to get started with JRA, i.e. creating a recording and open the tool. The following subjects will be covered:

What is New in the JRA

- The JRockit Runtime Analyzer is now part of the JRockit Mission Control version 1.0. That means that you now need a license to run the it.

For a complete list of the enhancements for this release, please see the Release Notes.

JRockit Mission Control License Information

The following license types are currently available:

- **Developer License**—The Developer License is free and allows JRockit to run the tools for one hour.
- **Enterprise License**—The Enterprise License allows unlimited use of the tools. It is bound to an IP address.

To download the correct license, please visit:

http://dev2dev.bea.com/jrockit/tools.html

Creating a Recording

There are several ways to create and start a JRA recording:

- Start Recording from Management Console
- Start Recording with jrcmd
- Start a Recording from the JRockit Command Line

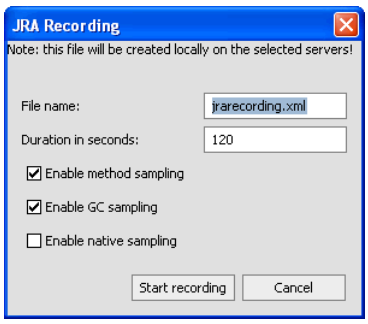
Start Recording from Management Console

1. Start your Java application with BEA JRockit, and add the `-Xmanagement` option to the command line.
2. Start the Management Console and connect to the JRockit instance you just started.
   For JRockit 5.0, see this BEA JRockit Management Console user guide
   For JRockit 1.4.2, see this BEA JRockit Management Console user guide.
3. 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.
4. In the BEA JRockit Management Console, click **Plugins > Start JRA recording**.

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

Figure 1-1 JRA Recording Dialog Box



5. Type a descriptive name for the recording in the **File name** field.

   This is the name of the file on the host machine where the recording is made. The file is created in the current directory of the BEA JRockit process, unless you specify a different path. It will be overwritten if it already exists.

6. Set a time for the length of the recording (in seconds).

   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.

7. Select none, one, or all of the following sampling options:
   - **Method sampling**—records samples of methods
   - **GC sampling**—records garbage collection events
   - **Native sampling**—records samples of native code

8. Click **Start recording**.

   The JRA Recording Progress box appears (Figure 1-2).

Figure 1-2 JRA Recording Progress Box

This box indicates that the recording has started. You will see a confirmation message, in the JRockit command line window, halfway through the recording and when the recording is finished. After the final message is printed you can shut down your application if you want.

Start 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 BEA 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.

Start a Recording from the JRockit Command Line

You can also start a JRA recording from the command line by using some additional options to the `java` command when you start up the application you want to record. Table 1-1 lists the different options depending on which version of JRockit you are running.

If you are running BEA JRockit version 7.0sp7/1.4.2_04 or newer use the command `-XXjra` together with the parameters listed in the BEA JRockit 7.0sp7/1.4.2_04 or newer.

If you are running BEA JRockit version 1.4.2_03 or older, you need to set each parameter with its own startup option (listed in BEA JRockit 1.4.2_03 or older in Table 1-1).

Table 1-1 Command Line Startup Parameters

BEA JRockit 7.0sp7/1.4.2_04 or newer	BEA JRockit 1.4.2_03 or older	Description
`delay`	`-XXjradelay`	Amount of time, in seconds, to wait before recording starts.
`recordingtime`	`-XXjrarecordingtime`	Duration, in seconds, for the recording. This is an optional parameter. If you don't use it, the default is 60 seconds)
`filename`	`-XXjrafilename`	The name of recording file. This is an optional parameter. If you don't use it, the default is `jrarecording.xml`.
`sampletime`	`-XXjrasampletime`	The time, in milliseconds, between samples. Do not use this parameter unless you are familiar with how it works. This is an optional parameter.
`nativesamples`	`-XXjranativesamples`	Displays method samples in native code; that is, you will see the names of functions written in C-code. This is an optional parameter.

Table 1-1 Command Line Startup Parameters

BEA JRockit 7.0sp7/1.4.2_04 or newer	BEA JRockit 1.4.2_03 or older	Description
`methodtraces`	Not applicable	You can set this to false to disable the stack trace collection that otherwise happens for each sample. The default value is true.
`tracedepth`	Not applicable	Sets the number of frames that will be captured when collecting stack traces. Possible value are 0 through 16. The default value is 16.

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.

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`).

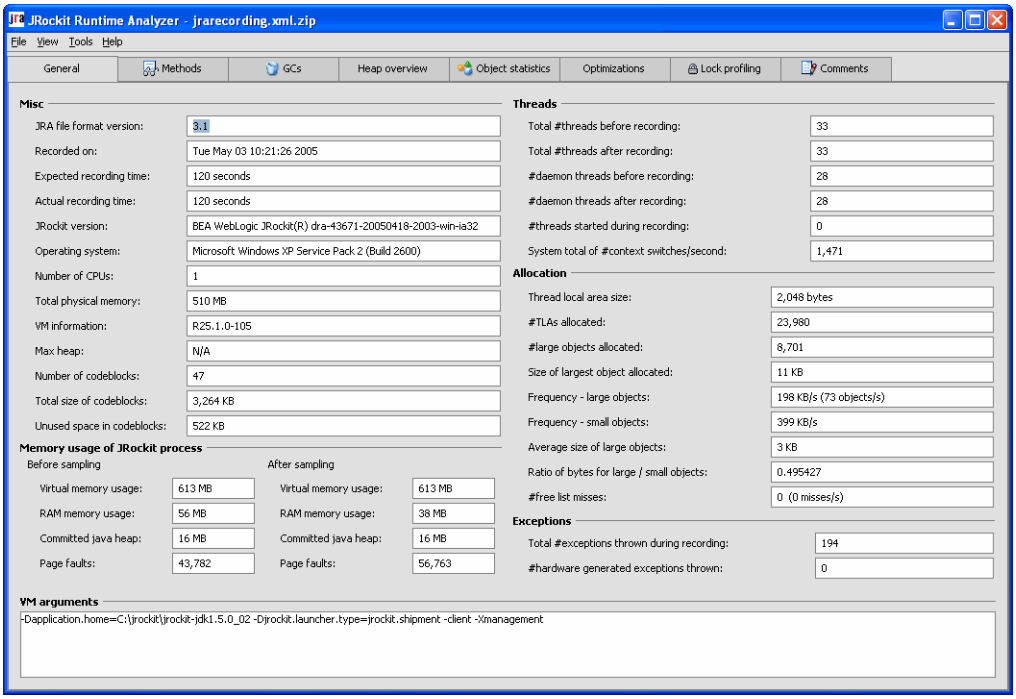
To replicate this data with the JRA version released with BEA JRockit 1.4.2_03 or older, you would need to enter the following four separate commands:

- `-XXjradelay=10`
- `-XXjrarecordingtime=100`
- `-XXjrafilename=jrarecording2.xml`

Starting the JRA Tool

Use the JRA tool to see the recording you just created. Start the tool with `java -jar RuntimeAnalyzer.jar`. This will open the JRA application (see Figure 1-3):

Figure 1-3 JRA Application with an Open Recording



Opening a Recorded File

1. Click **File > Open file**.
2. Locate and select the recorded file and click **Open**.

   The Improve JRockit window opens. In this window you find information on how you can help the JRockit engineering team improving JRockit and the JRA.
3. Click **OK**.

   The JRA General window is now filled with data (see Figure 1-3).

   Note: If you are running an older version of the JRA, some fields may not have any relevant data, since that data was impossible to obtain.

CHAPTER 2

Looking at a Recording

This section describes the information displayed on the different tabs on the JRA tool. A lot of the information requires a very deep understanding of the inner workings of the BEA JRockit virtual machine to be useful. It is out of the scope of this document to explain the meaning of all that data in detail. The following topics are covered in this section:

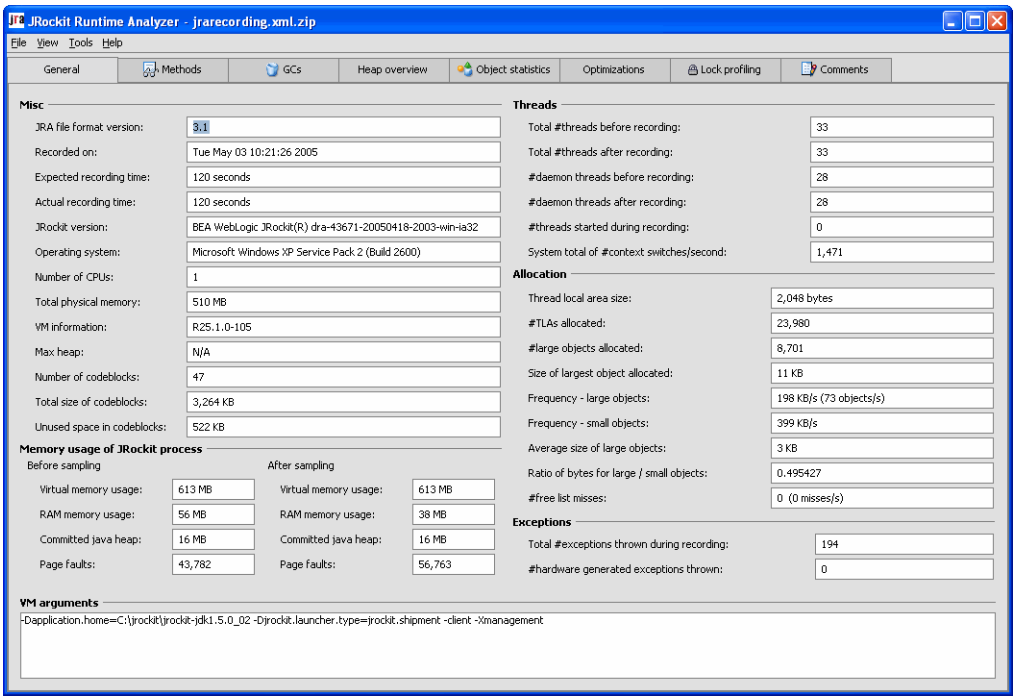
Viewing General Data about the Application

The **General tab** (see Figure 2-1) displays data about the recorded application behavior, such as thread, allocation, exception and memory usage statistics. It also displays the command line options that were used to start BEA JRockit (**VM arguments** section).

Note: If you are running an older version of JRockit (older than 5.0), data will not be collected for all fields, i.e. the field is left blank or marked as “unknown.”

Most of the information that is displayed in the **General** tab is pretty straight forward and self-explanatory; however, some labels require further explanation. Below follows explanations on selected label of the **General tab**.

Figure 2-1 General Tab



Miscellaneous Information

- 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 **Max heap**, maximum heap size, that is set with a BEA JRockit startup option.
- The value **Number of codeblocks** is a JVM internal value. All generated code is divided into (non-heap) memory blocks called codeblocks.

Memory Usage of JRockit Process Information

This section shows a snapshot of the memory usage before and after the recording of JRockit's memory usage.

- The value **Committed java heap** is the current total heap size. It is less than or equal to the maximum heap size.

Threads Information

This section shows information on the number of Java threads that existed before and after the recording.

- 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.

Allocation Values Information

- 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."

Exceptions Information

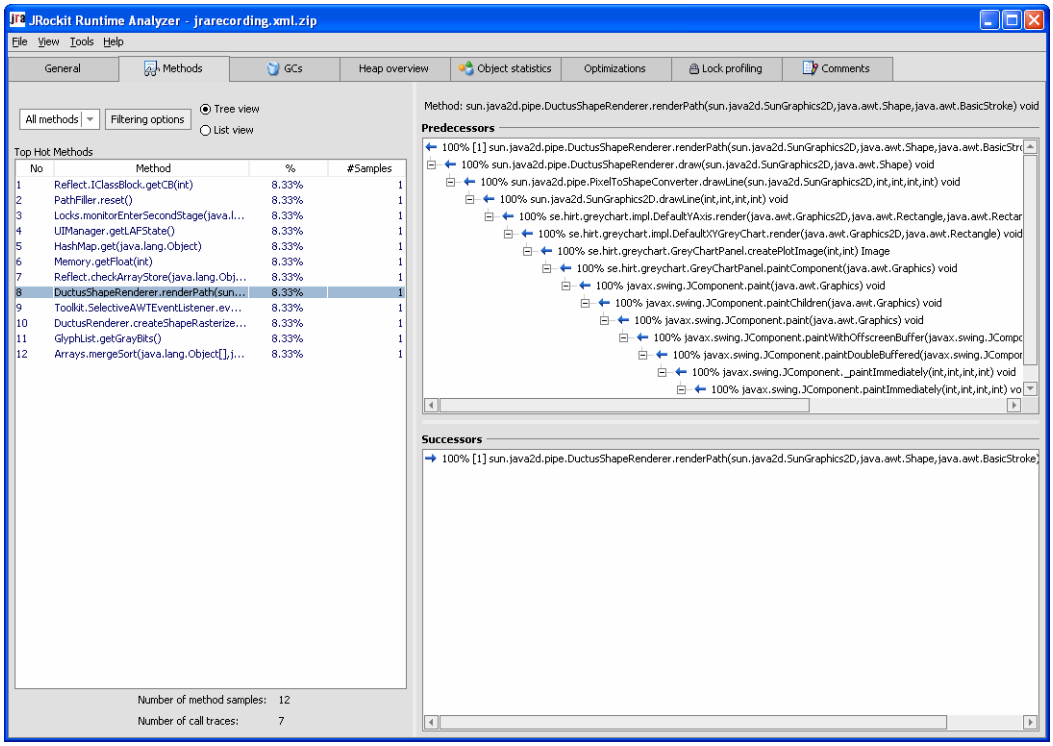
This section displays information on the total number of Java exceptions that are thrown during a recording. This includes both caught and uncaught exceptions. Excessive exception throwing can be a performance problem. Hardware generated exceptions are originating from a "trap" in the hardware and are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

Viewing Hot Methods

The **Methods tab** (see Figure 2-2) lists the top hot methods during the recording. 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 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 2-2 The Methods Tab



The hot methods data is collected periodically by JRockit sampling the running threads and looking at which method the threads are executing. If your recording has native sampling enabled, you can also see methods prefixed by `jvm#`, which are native methods in the JVM. You can limit the methods shown in the top list by clicking **Filtering options** and select the least number of samples that you want displayed.

By selecting a method in the list, you can see its sampled **Predecessors** and **Successors** in the tree view to the right. These are the methods that call the method and the methods that the selected method calls. The number in brackets is the number of sampled call traces of which the method is part. The percentages show how common a particular path is in the method tree.

If you prefer to see the successors and predecessors in a list view, you can change the view by selecting **List view** instead of **Tree view**.

Viewing Garbage Collection Data

The **GCs tab** (see Figure 2-3) shows detailed information about each garbage collection (GC) event that has occurred. Graphs show the heap usage before and after each garbage collection as well as pause times and number of `java.lang.ref.*` objects discovered. You select a specific garbage collection event in the list, **GCs during recording**, to view details about it in the lower right pane.

Figure 2-3 The GCs Tab

GCs During Recording Information

This section lists all garbage collection (GC) 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). Click on a garbage collection in the list to see it in the **GC Charts** tab and the **Details on selected GC** part.

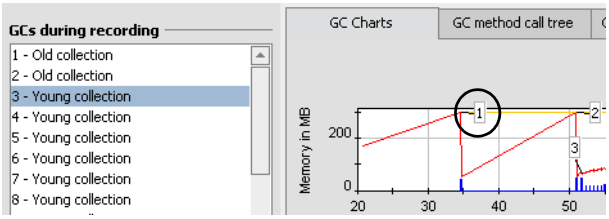
GC Charts Tab Information

The information that is displayed on the **GC Charts tab** contains two different charts and two buttons:

- The upper chart, **Heap usage chart**, shows how the heap usage varies with the garbage collections (in red) and how long the garbage collection pause times are (in blue).
- The lower chart, **References chart**, shows different types of reference counts after each old collection. For more information, see the javadocs for `java.lang.ref` package.

By clicking a garbage collection sample in the left-hand list of GCs, a small flag, annotation, is lit to indicate that specific garbage collection on the chart. See Figure 2-4.

Figure 2-4 The Annotation that Indicates Statistics for the Selected Garbage Collection



- Click the **Show GC strategy changes** button to view garbage collection strategy changes in the heap usage chart (if any).
- Click the **Clear annotations** button to remove all annotations from the charts.
- The **GC method call tree tab** shows an aggregation of the call traces of the threads triggering a garbage collection.
- The **GC strategy changes tab** shows the list of garbage collection strategy changes that occurred during the recording. These changes can only happen if you are running BEA JRockit with the default `-Xgcprio:throughput` option (`-Xgcprio` for versions earlier than JRockit 5.0).

General Information

This section displays general garbage collection statistics for the duration of the recording, for example, the average old collection pause time (**Avg OC pause time**) or the average young collection pause time (**Avg YC pause time**).

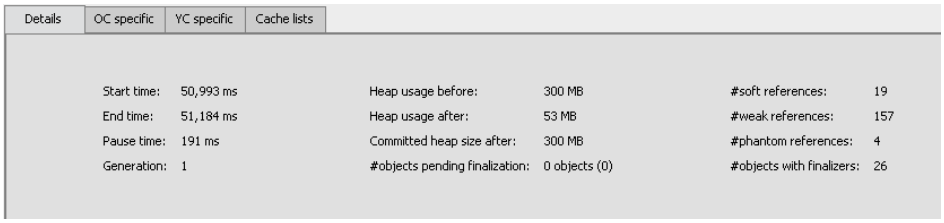
Selected GC Information

The **Details on selected GC** section contains four tabs with in-depth information relevant to the garbage collection round you have selected.

Details Tab

The following information is displayed in the Details tab (see Figure 2-5).

Figure 2-5 Details Tab

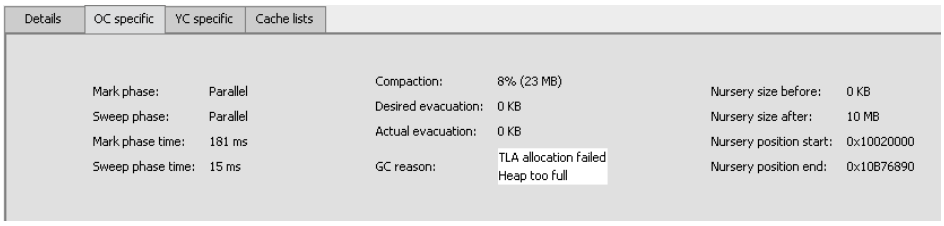


- **Start** and **End time**(s)—the time(s) when the garbage collection started/ended, counted in milliseconds from when JRockit started.
- **Pause time**—the time in milliseconds that the garbage collector stops all threads in JRockit. This is not the same as end time-start time in the case of a concurrent garbage collector.
- **Generation**—1 indicates a garbage collection in old space, 0 a garbage collection in young space. In the case of a parallel garbage collector, only generation 1 exists.
- **Heap usage before/after**—the used heap size before/after the garbage collection.
- **Committed heap size after**—the total size of the heap (used + unused memory) after the garbage collection.
- **# objects pending finalization**—if a number in parentheses is shown, this is the value before the recording.

OC Specific Tab

The following information is displayed in the **Old Collection (OC) specific tab** (see Figure 2-6).

Figure 2-6 Old Collection (OC) Specific Tab



- **Mark/Sweep phase**—indicates if the phase is concurrent or parallel.
- **Mark/Sweep phase time**—indicates the time spent in this phase measured in milliseconds.
- **Compaction**—ratio and size of the heap that was compacted in this old space garbage collection.
- **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.
- **Nursery size**—indicates the size of the young space of the heap before and after garbage collection (in some cases the nursery size can increase).
- **Nursery position**—memory address of nursery (sum internal).

YC Specific Tab

The following information is displayed in the **Young Collection (YC) specific tab** (see Figure 2-7).

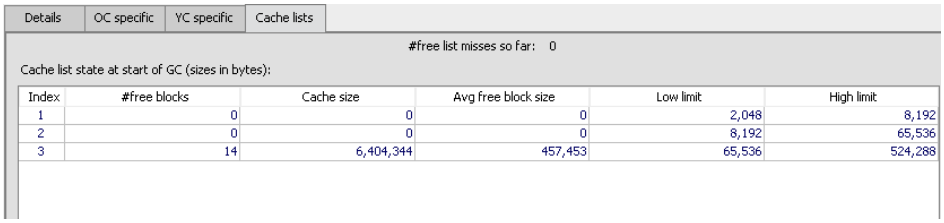
Figure 2-7 Young Collection (YC) Specifics Tab

- **Nursery usage before/after**—the amount of used memory in the nursery before and after the garbage collection.

Cache Lists Tab

Here you can view the specification for the different cache lists (see Figure 2-8). Each cache list contains settings for upper and lower cache size.

Figure 2-8 Cache Lists Tab



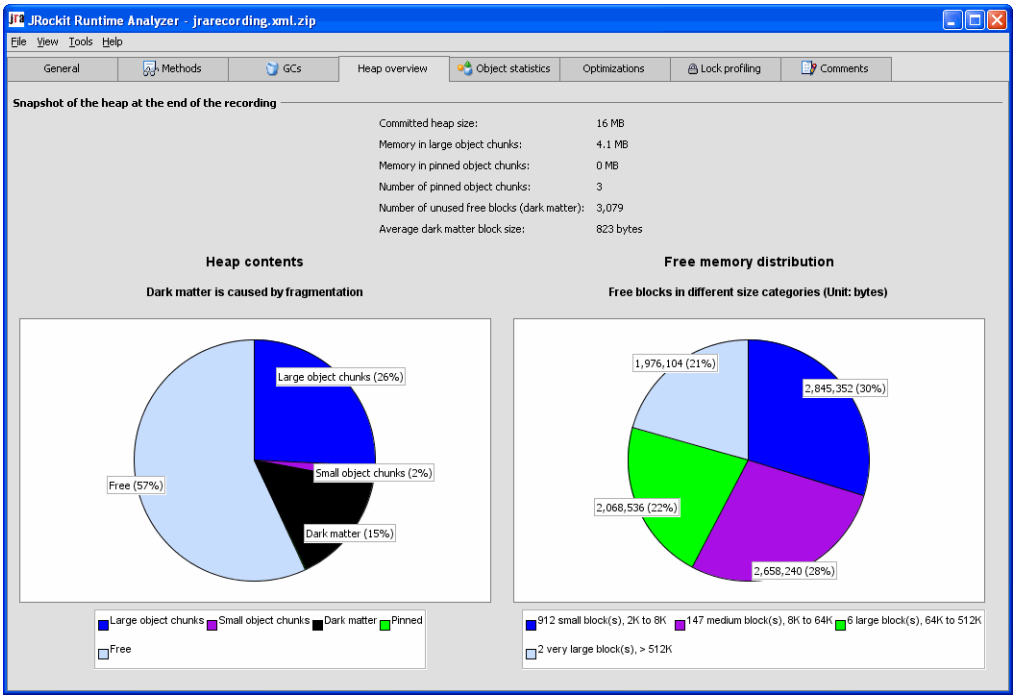
- **Index**—this is the identification number for the cache list.
- **#free blocks**—the number of free blocks in the cache list.
- **Cache size**—the total size of this cache list.
- **Avg free block size**—the average size of each free memory block in the cache list.
- **Low limit**—the lower limit 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.

Viewing Java Heap Content

The **Heap overview tab** (see Figure 2-9) gives a quick overview of what the memory in the Java heap consists of at the time of the recording. The tab consists of two pie charts that display the proportions of the: **Heap contents** (to the right) and **Free memory distribution** (to the left).

The information that is listed at the top of the Heap overview tab shows various statistics about the memory use on the Java heap.

Figure 2-9 The Heap Overview Tab



Heap Contents Pie Chart

The **Heap contents** pie chart shows how much of the heap that consists of large and small object chunks, dark matter, and free space. 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 in the heap.

Free Memory Distribution Pie Chart

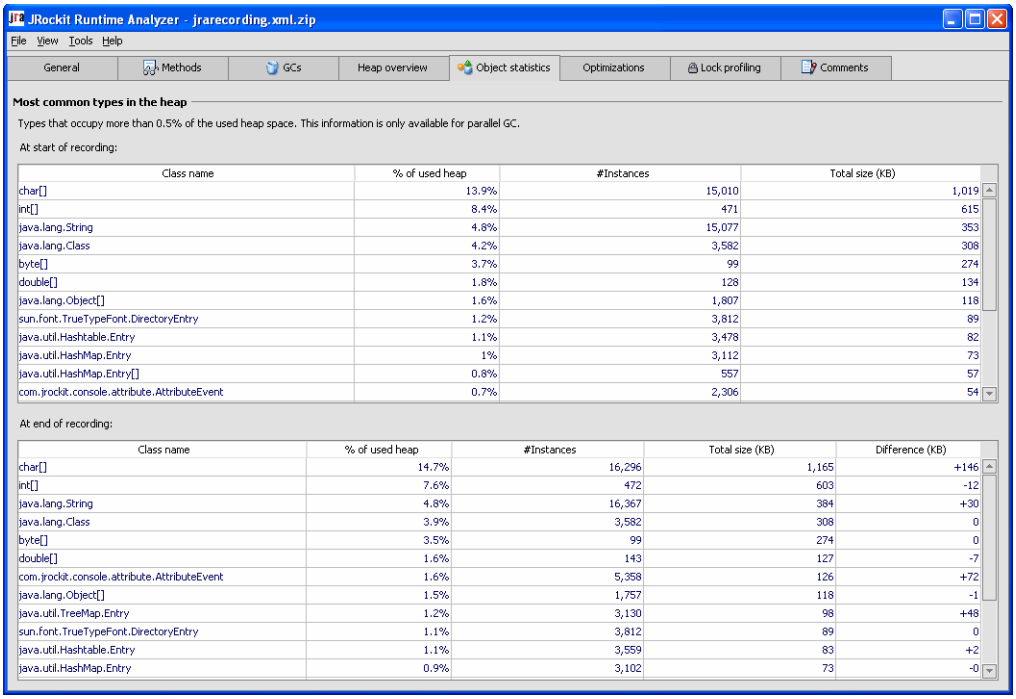
The **Free memory distribution** pie chart shows how the free memory is distributed in free blocks of different sizes on the Java heap.

Viewing Object Statistics

At the beginning and end of a recording session, snapshots are taken of the most common types/classes of objects that occupy the Java heap, that is, the types which instances in total

occupy most memory. The results are shown on the **Object statistics tab** (see Figure 2-10). Abnormal results in the object statistics might help you detect the existence of a memory leak in your application.

Figure 2-10 The Object Statistics Tab

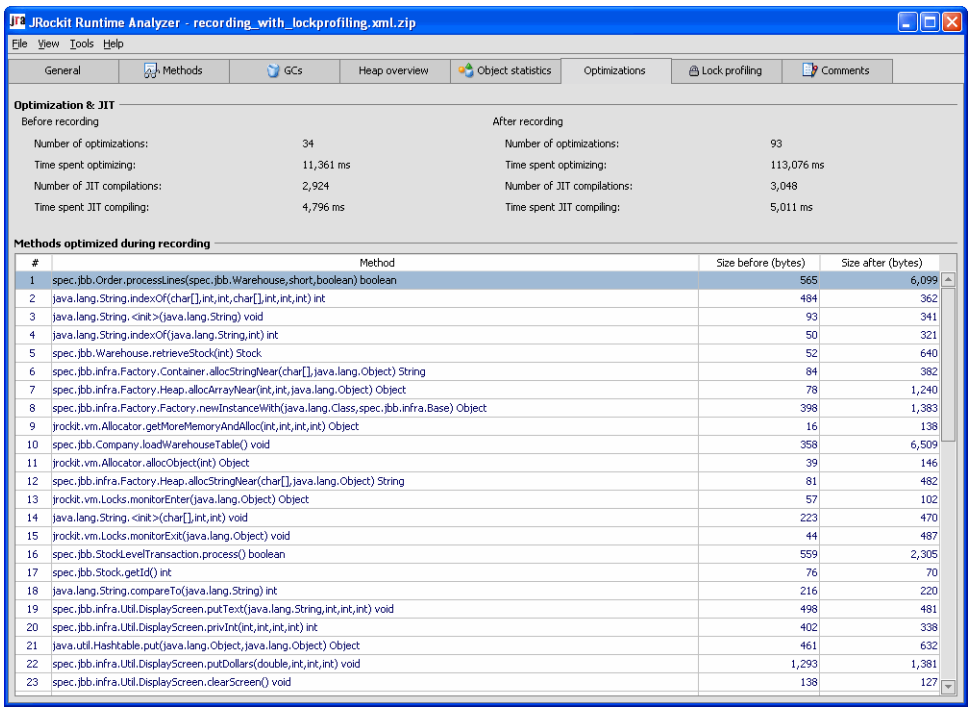


Viewing Method Optimizations

The **Optimizations tab** (see Figure 2-11) displays the methods that were optimized by the adaptive optimization system in BEA JRockit during the recording.

The optimized methods are displayed in chronological order. The sizes in the **Methods optimized during recording** table are the method size in bytes before and after optimization. Some optimizations, like inlining, causes the method size to increase. The information that is available under **Optimization & JIT** displays how JRockit has performed in optimizing the code of your application.

Figure 2-11 The Optimizations Tab



Viewing Lock Activities in Your Application and JRockit

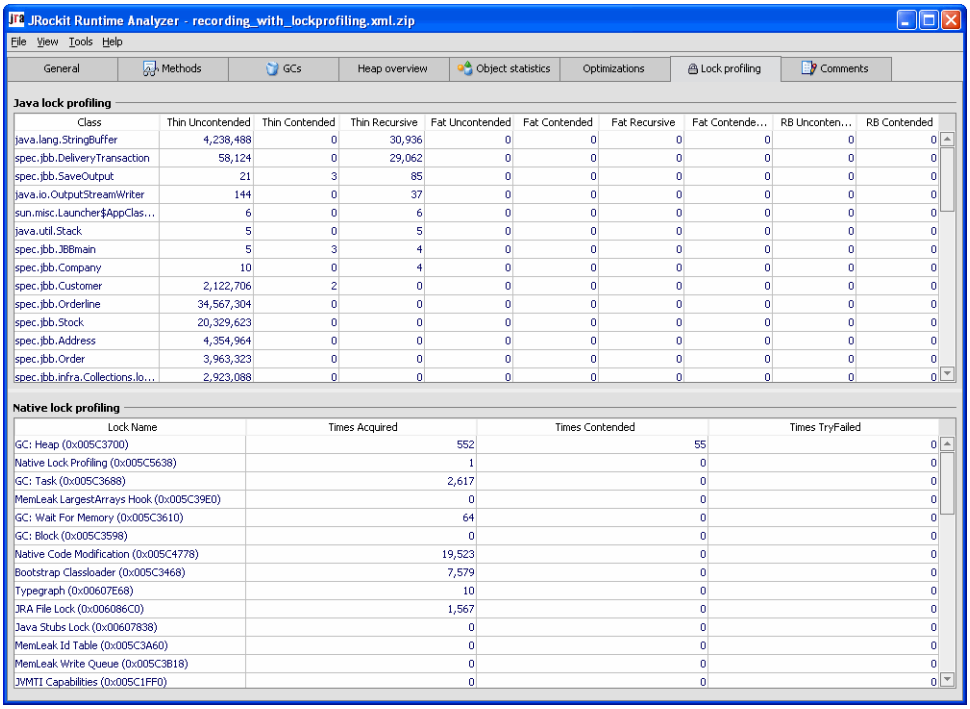
The **Lock Profiling tab** (see Figure 2-12) shows comprehensive information about lock activity for the application JRA is monitoring. A lock profile can only be generated when the `-Djrockit.lockprofiling` command is issued at the JRockit command line.

For example:

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

For more information on locks, please refer to the appendix About Thin, Fat, Recursive, and Contended Locks in BEA JRockit.

Figure 2-12 Lock Profiling Tab



Java Lock Profiling

The information that is displayed under the **Java lock profiling** chart shows the number of locks of the threads in your application. You see information on the number of thin uncontended, thin contended, and thin recursive locks; the number of fat uncontended, fat contended, and fat recursive locks; and the number of reservation bit (RB) uncontended and contended.

Click on the column header to sort the information for that column content.

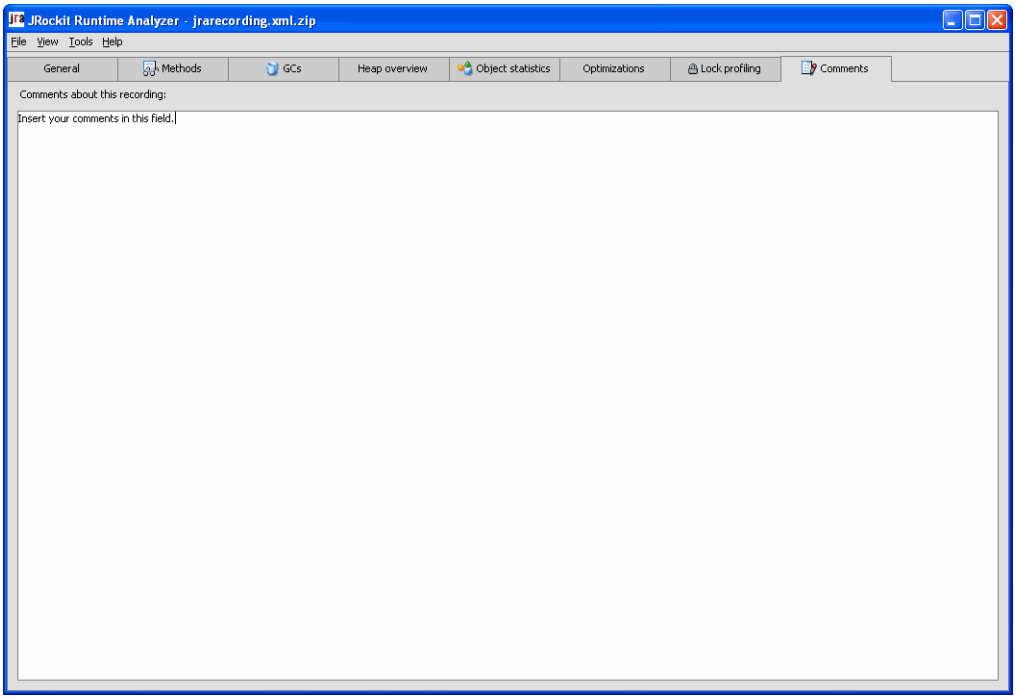
Native Lock Profiling

If you are looking at a recording of JRockit J2SE 5.0 or later, the recording includes information about native locks. Native locks are locks in the JRockit internal code and is nothing your application can control. 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 newsgroup at the dev2dev web site.

Adding Comments to the Recording

On the **Comments tab** (see Figure 2-13), you have a simple text editor where you can add comments about the recording. This can be a useful place to add comments when sending the JRA recording to BEA. To save you comments, click **File** > **Save comments**.

Figure 2-13 The Comments Tab



CHAPTER 3

Help Us Improve BEA JRockit and the JRA Tool

The JRA tool provides an easy way to capture information about BEA JRockit when it is running in a deployment environment. JRA is designed to induce a minimal overhead when enabled, and no overhead at all when not enabled.

Because of this, the JRA can be used to capture a very “true” picture of the running system. The data that is recorded is actually what is happening and not a side effect of having the tool enabled. JRA is able to do this by being very tightly integrated into BEA JRockit.

The results of the recordings sent from customers are often used by the engineering team to guide for future improvements of BEA JRockit. We therefore invite you to help us with this effort by sending us the JRA recordings of your J2SE or J2EE application running with BEA JRockit. Once you’ve made a recording, just follow these steps:

1. Attach the recording file to an email
2. Describe your application in a few sentences.
3. Send the email to **jrockit-improve@bea.com**

How will BEA Use These Recordings

The recordings will be analyzed and used by the development team pretty much in the same way you have analyzed your recording. The information is then used to find new ways to improve BEA JRockit. This can help you getting your application to run faster and better in the future.

JRockit Support for JRA

Only the most recent versions of BEA JRockit supports the possibility to create JRA recordings, i.e., versions 7.0sp4 (and later), 8.1sp1 (and later), 1.4.2_x, and 5.0 (listed in chronological order).

The different BEA JRockit versions do not provide the exact same data to the recordings. Recordings of BEA JRockit 5.0 support more types of data than previous versions. You can use the latest JRA tool to view recordings from different BEA JRockit versions, but when you see a field in the tool that says “unknown”, this means that this particular data was not recorded in that version of BEA JRockit. If a field says “N/A” (Not Applicable) it means that the data is supported but not applicable in that particular case (for example, no data about garbage collections in young space exists if you run with the parallel garbage collector).

Frequently Asked Questions

The following are some questions that have frequently been asked about the JRA:

Is the Performance Overhead of the JRA Recording in BEA JRockit Significant?

Performance overhead required for making a JRA recording is very low. For the applications that have been measured for BEA JRockit 8.1sp1 and 2 it has been only 1-2%. For BEA JRockit 5.0 it is slightly higher since more data is recorded. The overhead can momentarily be larger when the recording starts and when the recording is zipped and the file written to disk. There is absolutely no overhead once the recording has stopped.

When JRA reports method time, is it CPU time or elapsed time?

It is the CPU time that is reported and the time reported is only for the method itself not its children.

Is there any way to select CPU time v.s elapsed time or self v.s including children?

No, you can only see elapsed time in the CPU and elapsed time of the method, not its children.

Is There a Forum Where I can Discuss the JRA?

If you have any questions you are welcome to share them in the BEA JRockit general interest newsgroup, which is monitored by our engineering team. To access the newsgroup, go to:

http://newsgroups.bea.com/cgi-bin/dnewsweb?cmd=xover&group=jrockit.developer.interest.general&utag=

APPENDIX A

About Thin, Fat, Recursive, and Contended Locks in BEA JRockit

This is a description of the different kinds of locks in BEA JRockit.

Let's start with the easiest part: recursive locks. A recursive lock occurs in the following scenario:

```
synchronized(foo) {  // first time thread takes lock
  // ...
  synchronized(foo) {  // this time, the lock is taken recursively
    // ...
  }
}
```

The recursive lock taking may also occur in a method call several levels down—it doesn't matter. Recursive locks are not necessarily any sign of bad programming, at least not if the recursive lock taking is done by a separate method.

The good news is that recursive lock taking in JRockit is extremely fast. In fact, the cost to take a lock recursively is almost negligible. This is regardless if the lock was originally taken as a thin or a fat lock (explained in detail below).

Now let's talk a bit about contention. Contention occurs whenever a thread tries to take a lock, and that lock is not available (that is, it is held by another thread). Let me be clear: contention **always** costs in terms of performance. The exact cost depends on many factors. I'll get to some more details on the costs later on.

So if performance is an issue, you should strive to avoid contention. Unfortunately, in many cases it is not possible to avoid contention—if your application requires several threads to access a single, shared resource at the same time, contention is unavoidable. Some designs are better than others, though. Be careful that you don't overuse synchronized-blocks. Minimize the code that has to be run while holding a highly-contended lock. Don't use a single lock to protect unrelated resources, if that lock proves to be easily contended.

In principle, that is all you can do as an application developer: design your program to avoid contention, if possible. There are some experimental flags to change some of the JRockit locking behavior, but I strongly discourage anyone from using these. The default values is carefully trimmed, and changing this is likely to result in worse, rather than better, performance.

Still, I understand if you're curious to what JRockit is doing with your application. I'll give some more details about the locking strategies in JRockit.

All objects in Java are potential locks (monitors). This potential is realized as an actual lock as soon as any thread enters a synchronized block on that object. When a lock is *born* in this way, it is a kind of lock that is known as a "thin lock." A thin lock has the following characteristics:

- It requires no extra memory—all information about the lock is stored in the object itself.
- It is fast to take.
- Other threads that try to take the lock cannot register themselves as contending.

The most costly part of taking a thin lock is a CAS (compare-and-swap) operation. It's an atomic instruction, which means as far as CPU instructions goes, it is slow. Compared to other parts of locking (contention in general, and taking fat locks in specific), it is still very fast.

For locks that are mostly uncontended, thin locks are great. There is little overhead compared to no locking, which is good since a lot of Java code (especially in the class library) use lot of synchronization.

However, as soon as a lock becomes contended, the situation is not longer as obvious as to what is most efficient. If a lock is held for just a very short moment of time, and JRockit is running on a multi-CPU (SMP) machine, the best strategy is to "spin-lock." This means, that the thread that wants the lock continuously checks if the lock is still taken, "spinning" in a tight loop. This of course means some performance loss: no actual user code is running, and the CPU is "wasting" time that could have been spent on other threads. Still, if the lock is released by the other threads after just a few cycles in the spin loop, this method is preferable. This is what's meant by a "contended thin lock."

If the lock is not going to be released very fast, using this method on contention would lead to bad performance. In that case, the lock is “inflated” to a “fat lock.” A fat lock has the following characteristics:

- It requires a little extra memory, in terms of a separate list of threads wanting to acquire the lock.
- It is relatively slow to take.
- One (or more) threads can register as queueing for (blocking on) that lock.

A thread that encounters contention on a fat lock register itself as blocking on that lock, and goes to sleep. This means giving up the rest of its time quantum given to it by the OS. While this means that the CPU will be used for running real user code on another thread, the extra context switch is still expensive, compared to spin locking. When a thread does this, we have a “contended fat lock.”

When the last contending thread releases a fat lock, the lock normally remains fat. Taking a fat lock, even without contention, is more expensive than taking a thin lock (but less expensive than converting between fat or thin locks). If JRockit believes that the lock would benefit from being thin (basically, if the contention was pure “bad luck” and the lock normally is uncontended), it might “deflate” it to a thin lock again.

A special note regarding locks: if `wait/notify/notifyAll` is called on a lock, it will automatically inflate to a fat lock. A good advice (not only for this reason) is therefore not to use locking for notification with any additional locking schemes on a single object.

JRockit uses a complex set of heuristics to determine amongst other things:

- When to spin-lock on a thin lock (and how long), and when to inflate it to a fat lock on contention.
- If and when to deflate a fat lock back to a thin lock.
- If and when to skip on the fairness on a contended fat lock to improve performance.

These heuristics are dynamically adaptive, which means that they will automatically change to what’s best suited for the actual application that is being run.

Since the switch between thin and fat locks are done automatically by JRockit to the kind of lock that maximizes performance of the application, the relative difference in performance between thin and fat locks shouldn't really be of any concern to the user. It is impossible to give a general answer to this question anyhow, since it differs from system to system, depending on how many CPUs you have, what kind of CPUs, the performance on other parts of the system (memory,

cache, etc.) and similar factors. In addition to this, it is also very hard to give a good answer to the question even for a specific system. Especially tricky is it to determine with any accuracy the time spent spinning on contended thin locks, since JRockit loops just a few machine instructions a few times before giving up, and profiling of this is likely to heavily influence the time, giving a skewed image of the performance.

To summarize: If you're concerned about performance, and can change your application to avoid contention on a lock—then do so. If you can't avoid contention, try to keep the code needed to run contended to a minimum. JRockit will then do whatever is in its power to run your application as fast as possible. Use the lock information provided by JRA as a hint: fat locks are likely to have been contended much or for a long time. Put your coding efforts into minimizing contention on them.

Index

H

I

J

L

M

N

O

P

R

S

T

U

X

Y