Profiling Java applications using IBM Rational Application Developer

An introduction to using the Rational Application Developer profiling functionality to profile local and remote Java applications

Charu Jain
Jonathan West

April 01, 2010

This tutorial provides you guidance on using IBM® Rational® Application Developer to profile your Java applications. To that end, it will first provide relevant background on the Eclipse Test and Performance Tools Project (TPTP) Java™ Virtual Machine Tool Interface (JVMTI) profiling agent functionality.

Overview

The ongoing advancement of technology, in both processing power and storage technologies, has brought with it a number of new and interesting technologies. These technologies trade pure application performance for secondary concerns, such as programmer efficiency or system flexibility. Among these are technologies like garbage-collected just-in-time-compiled languages such as Java™, and a greater prevalence of whole-system virtualization.

As computer processing power and speed rapidly grows ever greater, and the cost per unit of processing power continues to decline, it might seem that the need for individual application efficiency may seem to be lessening. However, even the smallest applications may suffer crushing performance issues when applied across a sufficiently large number of users. Also, the largest applications may fall prey to nasty performance bottlenecks and memory leaks that hurt application availability, impair usability due to page load time issues, and potentially require costly upgrades where code fixes would suffice.

The IBM® Rational® Application Developer and IBM® Rational® Software Architect profiling tooling provides sophisticated tools that developers can use to identify and alleviate these performance issues. Both products package the profiling tools that are described in this tutorial; however, while the functionality and features are available in either, this tutorial focuses on Rational Application Developer. The profiling functionality is based on the open source Eclipse Test and Performance Tools Project (TPTP) Java™ Virtual Machine Tool Interface (JVMTI) profiling agent functionality.
and Performance Tools Project (TTPP) Java™ Virtual Machine Tool Interface (JVMTI) Profiling agent, for which more information is available in the Related topics section.

The Rational Application Developer profiling platform provides three different analyses of application behavior:

- Memory-usage analysis
- Method-level execution analysis
- Thread analysis

Built-in integration with the existing Rational Application Developer launch types makes profiling your application as easy as selecting the stop-watch profiling icon, and then selecting your existing Run/Debug launch configurations from the launch list. However, when your profiled application has launched, and the data has begun to be collected, a familiarity with the terms and concepts of application profiling will be helpful in getting the most out of the profiling functionality.

This tutorial provides you guidance on using Rational Application Developer to profile your Java applications. To that end, it will first provide relevant background on the Performance Tools functionality.

**JVMTI interface and agent architecture**

The Rational Application Developer Java profiler consists of a set of native agents that are implemented through the Java Virtual Machine's JVMTI interface. The JVMTI interface is a standard interface that allows native libraries (known as agents) to control a running virtual machine (VM) and obtain information about the executing Java application. A JVMTI agent subscribes to JVM events (and, where needed, instrument class byte code), to ensure that it is notified of all profiling events as the Java application executes on that VM.

The profiler registers its native functions through the JVMTI interface: these functions are invoked by the JVM when registered events occur. With the Rational Application Developer Java profiling agent, profiling data is generated in real-time. That is, the agent does not wait until the application has terminated to provide and present that data. The product supports agents to collect information on the three dimensions listed previously.

One important note: The Rational Application Developer JVMTI Java Profiler is not a sample-based profiler, and as such, all JVM events that are not filtered by the profiling filters will be transmitted back to the workbench. This method of profiling ensures a high level of accuracy, but at the expense of a greater profiling overhead versus sample-based profiling.

**Profiling agents**

*Profiling agents* are executed alongside the JVM (and inside the JVM process) when that JVM is run with special JVMTI-specific VM arguments. When the profiling agents run, they collect data from the JVM in the form of execution, heap, or thread events. Within the Rational Application Developer profiling sphere, these agents are referred to as the Execution Analysis, Heap Analysis, and Thread Analysis agents.
• The **Execution Analysis agent** is used to collect execution statistics, such as time spent in each method, the number of method calls, and the overall call graph.

• The **Heap Analysis agent** is used to generate memory usage statistics. It collects object allocation details, such as live instances and active size in memory.

• The **Thread Analysis agent** is used to obtain details about the threads spawned by the Java application, and to track object monitor use and contention throughout the target application.

To facilitate agent launching and client-agent communication, the profiling tooling uses a second component called the *agent controller* in order to allow agents to communicate with the workbench. This component is pre-bundled with Rational Application Developer.

Using the agent controller, a target Java application may reside on a different machine than that of the developer's workbench. The agent controller functions as an intermediary, sending commands and data between the workbench and the profiling agent. The following figure shows a remote profiling scenario. Communication between the developer's workbench and the agent controller is through sockets, while communication between the agent controller and agents is through named pipes and shared memory.

**Figure 1. Remote profiling**

In additional to agent support and communication, the agent controller provides additional services such as process launch, termination, and monitoring, and file transfer. It is always necessary for the agent controller process to be running on the target machine in order to utilize the profiling functionality on that machine. Fortunately, there are a variety of ways to start this process.

When profiling locally-launched applications directly in your local workbench, an *Integrated Agent Controller (IAC)* is started automatically, without user intervention, whenever any functionality that requires it is activated. The IAC will remain running as long as the workbench is open, and will shut down when the workbench is closed. The default IAC settings can be adjusted through the configuration view available under **Preferences > Agent Controller**.

In addition to local launch through the IAC, the agent controller can be used to launch applications on any supported remote host. A *standalone agent controller*, packaged as the IBM® Rational® Agent Controller daemon process, is available for download for 32- and 64-bit Linux®, Microsoft® Windows®, IBM® AIX®, Solaris, IBM® z/OS®, and Linux on IBM® System z® systems. These provide the ability for applications to be directly executed on a remote host, with the data generated by the remote agents being transmitted back to the local workbench through a socket connection to the remote machine.
See the Related topics section at the end of this tutorial for the Rational Agent Controller download locations.

What you need to go through the examples in this tutorial

All examples in this tutorial were profiled using Rational Application Developer Version 7.5.4. The profiling tools are available through either Rational Application Developer or Rational Software Architect, or a supported Eclipse-based product with the Eclipse TPTP plug-ins. See the Related topics section for more information.

Introduction to the Profiling dialog

To begin, switch to a Java-based perspective and look for the green play button profiling dialog icon (play). The Profile Configurations profiling dialog is the central launch point for all of the profiling functionality in Rational Application Developer. This may also be accessed from most perspectives by selecting Run > Profile Configurations from the menu.

The Rational Application Developer profiling functionality supports almost all of the standard launch types:

- Eclipse Application
- Java Applet
- Java Application
- Test
- Java Unit (JUnit)
- IBM® WebSphere® Application Server Versions 6.0, 6.1, and 7.0 application client
- And others

In addition, support is extended by two profiler-specific types, Attach to Agent and External Java Application:

- **Attach to Agent** allows any Java virtual machine to be profiled, independent of the use of that JVM by the application, as long as the appropriate JVMTI profiling agent VM arguments are used. Because this entry works with any application, it can be used for otherwise unsupported application configurations, either local or remote, whether they are on an application server or are a full-fledged standalone application. This requires that you configure the appropriate classpath, environment variables, and application parameters from the command line, along with the required JVMTI profiling agent arguments (described later in the tutorial).

- **External Java Application**, like Attach to Agent, supports profiling any JVM independent of its use. Unlike Attach to Agent, External Java Application requires that you specify the classpath, environment variables, application parameters, and profiling type in the workbench (in the launcher), rather than from the command line. One import point to note: all required class files, Java archive (JAR) files, and additional dependencies must already be present.
on the target host, because the agent controller does not support the remote file transfer of classes or files to meet the application requirements.

Collecting method-level execution statistics

Of the three profiling agents discussed in this tutorial, likely the most commonly-used agent is the method-level execution analysis agent, which provides a variety of statistics on method execution. The following example should provide a straightforward explanation of the available execution statistics' functionality.

1. First, select a Java application to profile.
2. Right click the Java project and select Profile as.
3. If this is the first time that you are profiling this project, the Profiling dialog is displayed. Otherwise it will automatically revert to the last saved profile configuration for the selected project.
4. To modify the previous profile configuration, use the Profile configurations dialog, as described previously.

Profiling options include Execution Time Analysis, Memory Analysis, and Thread Analysis, as shown in Figure 2.

Figure 2. Options in the Edit configuration and launch dialog
In the profile dialog you will see several profiling options: the previously-mentioned profiling options under **Java Profiling** are the subject of this tutorial. The fourth item, Probe Insertion, describes an additional functionality using the **Probekit** tool to instrument applications with custom probes, but is outside the scope of this tutorial.

5. From here, click the **Edit Options** button.

**Execution Time Analysis** has three options:

- **Execution Flow** provides a greater variety of profiling data views, but increases profiling overhead, profiling data volume, and workbench memory usage. This may be unsuitable for applications with large profiling data sets.
- **Execution Statistics** has a significantly reduced overhead, and workbench memory usage, but at the expensive of some profiling functionality. Only the Execution Statistics and Method Invocation Details will be available.
- **Collect method CPU time information**: In either of the above modes, the profiler can collect the time the CPU spends executing profiled methods. This will differ from the above, because I/O and wait times will not be included in the CPU time.

The best course of action is to begin with Execution Flow, and then switch to Execution Statistics (or the adjust filter set) if the data volume becomes too great.

6. For this example, select Execution Flow, because this provides an example of all of the available profiling views.

**Filters**

Before selecting the profile button, it is important that the proper filters are in place. A large amount of profiling data is generated as part of the profiling process, because every single method call, object allocation, or thread event requires an event to be generated, transmitted, and processed.

The sheer volume of profiling data can sometimes be overwhelming, both to the application under execution and to the workbench itself. However, only a fraction of it will be useful for analysis purposes. Fortunately, the profiling tool provides a way to filter out irrelevant information so that you can reduce the agent data volume for the profiled application.

For instance, in profiling a Java application, you are likely only concerned for the methods of your application, and not in the execution time of the standard Java language packages like java.*, sun.*, and so on. It is important to use filters that are specifically targeted to the classes of your application, so as to reduce profiling overhead from external classes as much as possible.

1. To set filters, double click the **Java profiling – JRE 1.5 or newer** entry. You will see a window similar to the one shown in Figure 3. Filters specify which packages and classes to profile; filters are themselves contained in configurable filter sets.
In this dialog, you define new filters or modify existing filters. A few filters that are useful in the most common profiling scenarios are provided, as shown in Figure 3. Filter sets are listed in the top pane, and the filters themselves are listed on the bottom. Filter precedence is from top to bottom, meaning that filters higher on the list will override any conflicting filters lower on the list. Filters and filter sets can be added and removed using the buttons on the right of the dialog box.

2. For this example, you are content with the default filter and select Finish to return to the profiling dialog box.
3. From here, select Execution Analysis, and click the Profile button. The workbench will confirm a switch to the Profiling and Logging perspective, and the profiled application will begin running in the Profiling Monitor view.
4. Click Execution Time Analysis to open the Execution Statistics view shown in Figure 4.
The **Session summary** tab provides an overview of the top ten methods with the highest base time. However, it’s the **Execution Statistics** tab from which all of the data is made fully available. Here you will find details on methods invocations and their statistics: the number of times called, average time spent, cumulative time, and so forth.

A few key definitions should help clarify the columns in this view:

- **Base Time**: The time to execute the contents of the method itself, *excluding* calls to other methods. (In the table, the Base Time field sums together all of the calls of that method.)
- **Average Base Time**: The average time that a particular method took to complete, *excluding* the time of method calls to other methods. (In the table, this is the Base Time divided by the number of Calls)
- **Cumulative Time**: The time to execute the method contents itself, *including* calls to other methods.

These statistics can help locate the performance bottlenecks in a program.

Based on these definitions, there are three important considerations:

- **Average Base Time**: This is the average time that a method took to complete. So, on average, this is how long a single invocation of that method took to finish (as noted above, this excludes the time taken by child methods called by this method or, more specifically, excluding the time of unfiltered child methods)
- **Base Time**: This is the total amount of time that a method took to complete. This is an amalgamation of all of the time that was spent in this method (excluding calls to other unfiltered methods.)
- **Cumulative CPU Time**: Cumulative CPU Time represents the amount of CPU time spent executing a specified method. However, the granularity of the data provided by the JVM in this regard is coarser then might be desirable. Consequently, CPU time maybe reported as zero if the time is less than a single platform-specific unit as reported by the JVM. Also, CPU time will not take into account other types of performance bottlenecks, such as those involving
communication type and I/O access time. As a result, base time is often favored as a metric for performance bottleneck reduction.

At first glance, average base time would seem to be a key data point in determining which methods are slowing down the system. However, while average base time will pinpoint methods that take a long time to execute, it does not take into account the number of times a method is called. You can ask yourself which is worse: a method that runs once and takes 5.0 seconds, or a method that runs 1000 times and takes 0.5 seconds? Average base time for the first would be 5.0 seconds, while average base time for the second would be 0.5 seconds.

However, the base time for the first method would be 5.0 seconds, while the base time for the second would be 500 seconds. Reducing 500 seconds from an application run time is tremendously more impactful than merely reducing 5. Thus you can say unequivocally that the second method is a greater concern than the first, due to base time difference. Base time is therefore a primary concern for reducing the performance bottlenecks in an application. Because base time represents an amalgamation of the entire application run, generally speaking reducing base time is equivalent to reducing run time.

While base time only represents time spent executing the method itself (excluding calls to other methods), cumulative time represents the total time spent executing a method, including child calls. So, for instance, in a single-threaded application, the cumulative time of the \texttt{main(...)} method is equivalent to the sum of all of the base times for all other methods in the application, because the \texttt{main(...)} method is the starting point of the application, and thus is the starting point for all methods calls. Therefore, it is equivalent to the total application run time.

With an understanding of the details of execution statistics in mind, let's look at ways to further drill down to analyze performance problems. To get a better idea of what specific methods are invoking, and from where specific methods are being called, double-click the desired method in the \textbf{Execution Statistics} tab. This will open the \textbf{Method Invocation Details} tab, as shown in Figure 5.
Figure 5. Drill down into finer detail of specific methods

The **Method Invocation Details** tab is a fairly straightforward presentation of those execution statistics that are directly related to the selected method. The second table in the tab is **Selected method is invoked by**, which will list all of the methods that called the selected method during the application run.

The third table, **Selected method invokes**, will list all of the methods that the selected method calls. The same statistics from the **Execution Statistics** tab are reproduced here, and selecting a method from any of these tables will update the selected method at the top of the view.

The next tab of note here is the **Call Tree** tab, which is only available when you are profiling with the Execution Flow profiling option. The **Call Tree** tab breaks down method calls for all of the methods that are called for a specific thread. The first-level items in the table are all of the threads that have spawned during the application run. Below this is an amalgamation of each of the method calls that are made, with methods on a subsequent level being called by methods from the previous level of the tree.
Figure 6. The Call Tree tab presents method invocations from a more thread-centric perspective

At the top-most level, the **Cumulative Time** for each thread represents the total time that the thread spent running in the application. Those threads with a higher cumulative time are candidates for analysis and optimization.

The **Percent Per Thread** field represents the total time spent executing a method as expressed as a percentage of the total time spent executing the thread. It is represented as a total of the cumulative time of the top-most method call for that thread (which is the total time spent executing the thread). Additional statistics are provided, such as the maximum, minimum, and average time that it took to complete a method execution on the thread, as well as the total number of calls.

Below the top call tree table is a table of method call stacks, which display the contents of the stack of every method call for the presently selected item in the call tree table. The number of stacks available will be equal to the number of calls listed. This can prove useful for analyzing particular method invocation instances.

Finally, you can right-click on the method entry and select Open Source from any of the views. If the related Java file is present in the workspace, the workbench file will be opened and the method definition will be located. When performance bottlenecks are identified, you can modify them and then profile again to see the difference.

**Heap analysis: locating memory leaks**

A primary objective of application developers looking to profile memory use is either to:

- Analyze the contents of the heap as the application is running (allowing class-by-class statistics to be generated)
• Identify memory leaks by utilizing either heap analysis on-the-fly or heap analysis after a user-requested garbage collection (to identify objects that are or are not being not garbage collected).

To begin to gather heap information from your application, launch your application from the Profiling dialog, using the Memory Analysis profiling type. The only profiler option available in memory analysis is whether or not to **Track Object Allocation Sites**. Allocation sites are the locations in the code where objects are being instantiated (implicitly or explicitly). With this option selected, you are able to select classes in the Object Allocations view, and discern from which methods those objects were created.

The only disadvantage to having the **Track Object Allocation Sites** option selected is that it will significantly increase the volume of profiling data being generated. If profiling performance is impacted, or workbench responsiveness is affected, consider either clearing this option or reducing the amount of data by using an updated filter set.

In addition to object allocation, ensure that you have set the correct filters for your application. Heap profiling has a greater overall instrumentation overhead associated with it than execution or thread analysis, which itself already imposes a potentially hefty penalty to application performance. Finally, when selecting filters: if you want to see, for instance, space occupied by Java types such as `String` or `Integer`, you will need to add these to the filter. By default, these are filtered out by the `java* *EXCLUDE` filter.

### Object Allocation view

When data becomes available from the profiled application, it is displayed in the Object Allocation view. The Object Allocation view is the main view in which all of the information gathered by the heap profiling agent is presented.

Object Allocation columns:

- **Live Instances**: The current number of objects in the heap, of the specified class, that are presently being utilized (have not been garbage collected).
- **Total Instances**: The total number of objects in the heap that have been created during the JVM's lifetime (including those that have been garbage collected).
- **Active Size (bytes)**: The total size of all of the object instances, of the particular class, that are presently used by the JVM (in other words, that have not been garbage collected). Note that object size is JVM-implementation dependent.
- **Total Size (bytes)**: The total size of all object instances of the class, including those that have been garbage collected earlier in the application lifecycle.
- **Average Age**: The average age of an object before it is garbage collected, as measured by the number of garbage collections that this object has survived. Objects that have survived a large number of garbage collections are considered to be a memory leak if their usage is no longer required by the application.

The data in the memory statistics table can be switched between package level and class level using the view toolbar. This can be especially helpful when you are dealing with a large number
of classes. The data can be represented as a percentage or as a delta of existing data from the previous refresh. Figure 7 shows the icons for, from left to right, report generation, filtering, package/class view options, and the percentage and delta options.

**Figure 7. Icons in the Object Allocations view toolbar**

If you have the Object Allocation profiling option selected, then double-clicking any of the entries in the Memory Statistics table will switch to the Allocation Details tab. This tab presents a table of all of the locations in the program where objects of that type were allocated. When a particular type has been identified as being over-represented in the heap, it can be particularly helpful for you to use the data in the view to determine where those objects are being allocated, allowing the identification and elimination of excessive allocation.

**Using this functionality to identify heap problems**

When the workbench has begun collecting profiling data from the target application, you can consult the Object Allocation view at any time in order to determine the present content of the heap. The table will reflect, in real time, all object allocation and de-allocation events as they occur. This provides a moment-to-moment view of the memory contents, expressed either as a percentage of the total, or in absolute terms (bytes).

By sorting the data table and selecting the classes with the largest active size, developers can target problem areas in need of improvement. The allocation details for those classes then provide a list of the object creation sources, which you can then, one-by-one, rule out or investigate as being a cause of the heap size issues.

This example represents a simple chat room Web application that allows users to log in and communicate with each other. Room chatter is transmitted to all participants, and then all conversations are written to a log file on the server. However, using the Heap profiling capabilities of Rational Application Developer, you have identified a serious memory leak (likely involving one specific class).

In the Memory Statistics view with the total percentage listed, you can see that the ChatLineMessage class represents nearly 98% of the objects presently allocated in the heap, composing 61% of the total heap contents by size. This should be a serious warning sign to the application developer that one or more classes are over-represented in the heap contents, and are contributing to a memory leak in the application.
Figure 8. Viewing by package with the delta option selected

The Profiling Monitor view also allows you to request garbage collection of the JVM. When used in combination with the delta table, this is especially helpful in determining how much of the heap is contained in irretrievable heap objects, which are another type of memory leak.

Before selecting garbage collection, select the Show Delta Columns icon from the Object Allocations toolbar. This introduces four new delta columns, which are delta versions of the existing columns and reflect changes in these values from when you previously selected the Refresh button (see Figure 9, following). When you are in the delta table, click the Run Garbage Collection (green play arrow with a garbage can) icon from the Profiling Monitor toolbar to trigger a JVM garbage collection. The results of the garbage collection will be reflected in the Memory Statistics table as soon as you select the refresh button, as shown in Figure 9 following.

Those classes that contributed the most to the garbage collected heap contents can be identified by sorting by the Delta: Active Size table column. These are the classes that lost the most size during the collection, and which contributed the most to the total unused object pool.

When the JVM performs a garbage collection, it looks for allocated objects in the heap that are orphaned (that is, they are not referenced by any other object in the heap, which itself is not referenced by any heap object.) Using the same chat application example as discussed previously, you have now instructed the JVM to perform a garbage collection through the workbench. You can see that the initial result is a major decrease in both the number of allocated ChatlineMessage objects, as well as the total size of the heap.
Figure 9. The delta columns provide moment-to-moment statistics on object collection

This screen capture shows the contents of the Memory Statistics view a moment after you have requested a garbage collection. Already, the contents of the heap have been reduced dramatically, with 22,079 fewer live instances and a corresponding percentage drop. Within about five or six seconds, the Live Instances percentage will drop to nearly zero. In this application, you discover that ChatlineMessage objects are being allocated, used briefly, and then discarded.

Using heap analysis, delta columns, and garbage collection, you can identify objects that are being allocated but not garbage collected, or that are not referenced by any other objects in the application lifecycle and are contributing to a large collection of orphaned objects. Application developers can analyze and reduce the overall footprint of their application, and can potentially reduce system swapping penalties, improve response time, and reduce application system requirements and cost.

Thread analysis: tracking thread behavior

The audience for this section is users who wish to identify and correct any number of thread-related issues in their application. For specific threads of interest, you can use these profiling tools to examine how often these particular threads are blocked (and by whom), and to visualize your application's thread characteristics.

For global concerns, such as for instance performance concerns where thread behavior is a potential suspect, the goal is to identify general thread issues that might impact application performance. A primary objective of an application developer who wants to profile thread behavior is to find threads that would otherwise run sooner, or more rapidly, were the resource and thread characteristics of the application altered.

To begin to gather thread information from the target application, launch the application from the Profiling dialog using the Thread Analysis profiling type. The Thread Analysis view of Rational Application Developer is the central view for all of the information gathered by the thread profiling agent.

The Thread Analysis view is split into three tabs:
• **Thread Statistics**: A table of statistics for every thread launched by the application, both past and present. Listed information includes thread state; total running, waiting, and blocked times; as well as the number of blocks and deadlocks per thread.

• **Monitor Statistics**: Provides detailed information on monitor class statistics, including block and wait statistics for individual monitor classes.

• **Threads Visualizer**: Provides a visual representation of all threads profiled in the target application, by status.

Threads in all of the views are organized by thread group. The data in the thread analysis view will *only* update whenever additional data is received from the profiled application. This point is an important one: the tables and graph will *only* be updated when a thread-related event is received from the target application profiler. If it appears as if the data is not being updated, this is because no new thread events have been received, and the data is still in its previous state.

The thread name listed is the name passed into the `Thread(String name)` constructor, or set with the `Thread.setName` method. It may be beneficial to call this method in the target application, in order to allow for easier thread identification. Thread statistics are gathered on all Java threads, which include VM threads, and may also include threads used by the application container or application framework. Fortunately, you can filter uninteresting threads out of the view by selecting the thread filter icon (three arrows with middle yellow one going through vertical green line 🔄), and clearing unwanted threads.

The seven thread states are:

• **Running**
• **Sleeping**: one in which the `sleep` method has been explicitly called
• **Waiting**: one on which the `wait` method has been explicitly called and is waiting for a `notify` or `notifyAll` call on its monitor object
• **Blocked**: refers to cases where a thread is blocked by an object monitor that is in use by another thread (for instance, a thread holding an object monitor in a synchronized statement)
• **Deadlocked**: statistics are gathered on a per-thread level, for each thread in the target application. A deadlock is considered to have occurred when two or more threads hold resources for which the resource dependency graph contains a cycle (that is, all deadlocked threads require additional resources that they cannot obtain without another deadlocked thread releasing required resources).
• **Stopped**
• **Unknown**

In Java, deadlocks can occur in a variety of situations, for example:

• When, in two threads, synchronized methods in two classes are trying to call the synchronized methods of each other.
• When one thread synchronizes on resource A and attempts to synchronize on a second resource B, while a second thread synchronizes on resource B and attempts to synchronize on resource A.
Any other situation where it is not possible for two or more deadlocked threads to unblock and complete.

**Thread statistics**

This tab lists all of the threads that are currently running, or have run, throughout the lifetime of the application. Threads remain in the table even after termination. This view lists running time, waiting time, and blocked time. **Running time** is defined as the total running time of the thread minus the time that it was waiting or blocked. **Waiting time** is defined as the amount of time that the thread spent waiting for a monitor, and **Blocked time** is the amount of time that the thread spent blocked by the ownership of monitors by other threads. In addition, there are several recorded counts: **Block count** and **Deadlock count** are the number of times a thread has blocked or deadlocked, respectively, throughout the life of the thread.

In this example, you are profiling an Eclipse plug-in. Figure 10 shows the present states of all running threads, their running time, waiting time, blocked time, and deadlocked time, and the blocked count. Application developers may utilize this view to either view a general picture of the entire thread landscape of their application, or to drill down to the moment-by-moment statistics of particular threads.

**Figure 10. List of threads that are running in a profiled Eclipse plug-in**

![List of threads](image)

The waiting time, blocked time, and deadlocked time are important statistics to consider in the context of application performance. These values should be closely scrutinized to ensure that they are appropriate, especially for time-dependent threads.

**Monitor statistics**

The second tab in the Thread Analysis view is **Monitor Statistics**, shown in Figure 11. All of the objects in Java have a corresponding monitor, which is the basis for all concurrency operations in Java. Monitors are invoked when inside a synchronization block, or when `wait` or `notify` methods are called to wait for a dependency or signal an availability, respectively. This tab provides monitor statistics on a thread-by-thread basis.
In Figure 11, the **Thread Statistics** table contains a list of threads in the profiled application, along with various statistics of the thread from the first tab. Select one of the threads in the **Thread Statistics** table to display the associated monitors referenced by that Thread, including various statistics for those monitors. You can then select the monitor to open up information about its class, including block and wait statistics for callers of the monitor, as well as timing and object information. This allows the identification of the particular objects that are in contention, by whom, and how often.

**Threads visualizer**

In the **Threads Visualizer** tab shown in Figure 12, each of the seven thread states is denoted by bars of varying backgrounds and line patterns. Threads are sorted by thread group and thread name. The x-axis of the graph represents time, the range of which can be adjusted using the zoom-in and zoom-out icons.

Each row of the table contains a bar representing thread execution. Inside each bar is a continuous list of events, which represent changes in the thread state. You can double-click an event to display its call stack in the Call Stack view, and you can move from event to event using the **Select Next Event** and **Select Previous Event** buttons in the top right-hand corner of the **Threads Visualizer** tab.

In the graph, Waiting and Blocked states are denoted by dotted lines, and Deadlocked and Stopped states are denoted by a solid line. Of most importance to the application developers looking to identify performance issues are Deadlocked (red), Waiting (orange), and Blocked (yellow).

One important UI note: Perhaps contrary to what might be expected behavior, when a thread has terminated it will continue to maintain a dark grey representation on the chart (rather than disappearing from the chart entirely).
Figure 12. The Threads Visualizer represents the thread status of all threads in a graph, plotted against the application timeline

The buttons on the thread analysis toolbar are used to move or change focus to or from particular threads. From left to right, as shown in Figure 13, the buttons are: Legend, Show Call Stack, Reset Timescale, Zoom In/Out, Select Next/Previous Event, Select Next/Previous Thread, Group Threads, Filter Threads, and Visualize Thread Interactions. Most of these are self-explanatory: for example, the Next/Previous Thread button changes the currently selected thread, and the Select Next/Previous Event button moves the event cursor to the next or previous event of the currently selected thread. Additionally, you can group and filter threads as required.

Figure 13. Interact with the Threads Visualizer graph using the buttons at the top of the view

When you use the Rational Application Developer thread profiling functionality, you should take into account a number of considerations:

- For threads that are on a wake-sleep cycle, how long does it take to complete the wake phase of the cycle, and how long does the thread sleep?
- For threads that are dependent on external resources becoming available, how long are they blocked waiting for a resource to become available?
- In input-processing-output oriented applications, such as a Web application, how long do the various threads take to respond to user input, process the data, and produce the corresponding output?
- Some threads periodically wake to check a condition or perform a function, then return to sleep. Thread analysis allows you to observe these relationships using the Threads Visualizer.
- You can use the profiling functionality to monitor producer-consumer and reader-writer relationships.
In aggregate, the thread profiling functionality provides a variety of views to analyze application thread performance and behavior. These views allow you to gather information and analyze various aspects of program execution, in order to gain insight into potential bottlenecks or failure conditions.

Profiling applications on a remote machine

So far, this tutorial has discussed profiling Java applications as they are running on a local machine. Rational Application Developer profiling also provides the capacity to launch and profile applications that are running on a machine separate from the workbench. To enable this functionality, you can download the Rational Agent Controller component and install it separately on Windows, Linux x86/x64, Linux for System Z, IBM AIX, IBM z/OS, Solaris SPARC, and Solaris x86.

Instructions to install and start the Rational Agent Controller are available on the IBM download site; consult the Related topics section for more details. When it is installed, run the SetConfig setup script, and then start the agent controller using ACServer.exe (Windows) or ACStart.sh (UNIX®) on the remote machine. An example Linux on System Z configuration is shown in Figure 14.

Figure 14. Starting the Rational Agent Controller process from the command line and setting up the Java Profiler environment variables

The profiler requires you to set additional environment variables. On Linux, for instance, agent-specific additions to the LD_LIBRARY_PATH and PATH variables are required. Other variables shown in Figure 14 previously are set just for the convenience of using their values more than once without having to type the long path. You can add these to your global environment variables, specify them in the terminal session, or add these to the launch script of your Java application. Additionally, some platforms allow you to profile without setting these environment variables (instead specifying the path on the command line). Consult the Getting Started document of your Rational Agent Controller installation for more information.
In addition to setting these environment variables, you'll also need to determine the type of profiling data that is required, and set the JVM arguments to reflect this type when launching your application.

Windows:
JVM Arguments on Windows:

```
-agentlib:JPIBootLoader=JPIAgent:server=<agent-behaviour>;<profile-option>
```

All UNIX varieties:
JVM Arguments on Linux:

```
'-agentlib:JPIBootLoader=JPIAgent:server=<agent-behaviour>;<profile-option>'
```

(Note the single quotes around the entire string; these are required so that the semi-colon is not interpreted as a new line character by the shell)

Note the use of CGProf (<profile-option> in the generic command line) in the command line JVM arguments in the example above: this is one of the options that corresponds to the data collection profiling types:

- **CGProf**: This is equivalent to Execution Time Analysis in the workbench Profiling UI. As mentioned previously, this option is used to identify performance bottlenecks, by breaking down execution time on a method-by-method basis.
- **HeapProf**: This is equivalent to Memory Analysis in the workbench Profiling UI. As mentioned, this option tracks the contents of the heap by tracing object allocation and deallocation, as well as garbage collection events.
- **ThreadProf**: This is equivalent to Thread Analysis in the workbench Profiling UI. This option traces thread and monitor usage during application execution.

You need to select one of the data collection types, and place that value in the profile-option JVM argument value above. You may only specify one at a time.

In addition, you need to select agent behavior (for instance, the example in Figure 14 uses **controlled**):

- **controlled**: This agent behavior prevents the JVM from initializing until the agent is attached to (from the workbench) and given instructions to start monitoring. As soon as the agent connection is established, the JVM will start. Because the JVM waits until the workbench has connected, the profiling agent will generate data for the entire lifecycle of the application.
- **enabled**: With this agent behavior, the profiling agent is launched at JVM startup. However, the JVM is initialized immediately, and begins running without waiting for the workbench to connect. The profiling agent does not begin to generate data until after the workbench has connected to the agent and started monitoring. No profiling data is produced until the workbench attaches. Any application execution that takes place before the workbench has connected will not be recorded.
An additional agent behavior is standalone, which is outside the purview of this tutorial. It allows profiling without an agent controller by writing data to a trace file on the local file system, which can then be directly imported into Rational Application Developer. Similarly, additional command line options are available to fine tune profiler data. For more information, consult the Rational Agent Controller Getting Started document.

**Example 1:**

```
-agentlib:JPIBootLoader=JPIAgent:server=enabled;HeapProf
```

(Heap profiling on Windows, mentioned enabled mode)

**Example 2:**

```
-agentlib:JPIBootLoader=JPIAgent:server=controlled;CGProf
```

(Execution time profiling, on Linux mentioned controlled mode)

When the target application JVM has been run with appropriate JVM arguments, you are ready to connect from the workbench. To connect from the workbench, bring up the Profile Configurations dialog, as shown in Figure 15.

**Figure 15. Selecting the Profiling Configurations dialog from the workbench UI**

A dialog box that shows the available profiling options is displayed, as shown in Figure 16.
Create a new Attach to Agent launch configuration (described previously) by double-clicking that option. You can also customize the new configuration by adding the remote machine as a new host. The agent controller on the remote machine is available at port 10002 (the default port number), as shown in Figure 17.

After it is added as a host, the agent running on the remote machine (in this example the execution statistics agent) should be available on the Agents tab. If not, it could help to verify the Agent Controller setup and status. When you are ready, select the agent and click Profile. The workbench will attach to the agent and switch to the profiling dialog. You can now collect and analyze the data as required.

One other way of profiling an application on a remote machine is to use the previously described External Java Application option (as shown in Figure 18) instead of Attach to Agent.
In a new configuration of External Java Application, you specify the location and name of the Java main class on the remote machine, as shown in Figure 18. The Monitor tab helps specify the kind of profiling agent to use (Execution profiler, Memory profiler, or Thread profiler). When you click the Profile button, the application executes on the remote host, but the input and output are directed to the console window in the local workbench.

**Profiling Eclipse Rich Client Platform plug-ins**

Rational Application Developer supports profiling Eclipse Rich Client Platform (RCP) plug-ins. You can perform this profiling through the Eclipse Application option in the Profiling Configurations dialog box. There is an option to profile a new Eclipse instance using the plug-ins that are under development in the workbench. When you profile Eclipse plug-ins, it is especially wise to use a filter set that limits profile data directly to the packages that relate to your particular plug-in. As with other launch configurations, the profiling launch UI is built on the existing launch UI, which means that the workbench maintains a consistent profiling UI across varied application types. Profiling a plug-in is as easy as profiling a local Java application or other launch type.

**Eclipse Web Tools Platform integration and profiling with WebSphere**

Additionally, Rational Application Developer supports profiling servers like WebSphere Application Server or Tomcat, either running on the local machine, or connected to a remote machine. Rational Application Developer’s profiling functionality closely integrates with existing server configurations. When you develop Web applications or Web services that run on WebSphere Application Server,
or other supported application servers, you can launch a profiled application by selecting the server to profile in the Server view and then selecting the Profile icon. From here, the Profile on Server dialog box is displayed, and you can select the profiling type, as well as additional choices such as filters and profiling options.

A note on server profiling: the JVMTI profiling agent collects data at the JVM level rather than collecting data on a per-application basis. This means that all Java code that runs on the JVM will generate event data (including the server itself). You must ensure that you have correct filters set up to correctly target your application.

What you have learned

This tutorial explored the multi-faceted profiling functionality provided by Rational Application Developer. Rational Application Developer provides a user-friendly and intuitive interface to examine those details that are helpful for tuning a Java application, all the while seamlessly integrating with existing application configurations. The profiler is available for a wide variety of platforms, and supports any and all JVM configurations quickly and easily. With the proper application of profiling tools, and careful analysis of application performance and characteristics, you can discover and deal with performance issues before they become a problem, and before more costly solutions are required.
Related topics

- Find out more about Rational Application Developer:
  - Try Rational Application Developer.
  - Explore the Rational Application Developer Information Center.
- IBM Rational Agent Controller download pages: The IBM Rational Agent Controller releases are available for download from this page. Additional installation instructions are available there, as well.
- Eclipse Test and Performance Tools Platform: Find more information about the open-source Eclipse project on which the Rational Application Developer profiling tools are based.
- Eclipse TPTP documentation: This is a full listing of the documentation produced for Eclipse TPTP projects, including tutorials, screencasts, and conference presentations.
- DeveloperWorks article: Read Introduction: Eclipse Test and Performance Tools Platform by Martin Streicher. This is a tutorial-style introduction to the profiling components of Eclipse TPTP. Note: The profiling agent described in this tutorial is only applicable for JVM V1.4.2 and 1.5.0. Likewise, major revisions to the UI and available functionality were made in subsequent versions.
- Thread Synchronization and the Java Monitor: Read a thorough description of the Java thread constructs, including monitors, thread concurrency, and synchronization.
- Subscribe to the IBM developerWorks newsletter, a weekly update on the best of developerWorks tutorials, articles, downloads, community activities, webcasts and events.