Trace-Based and Sample-Based Profiling in Rational Application Developer

This document is aimed at highlighting the importance of profiling in software development and talks about the profiling tools offered in Rational Application Developer (hereafter termed/referred to as 'RAD').

I. Why do we need to profile our programs?

Software application performance is a key aspect of delivering software. Poor performance and scalability results in high costs to the software company and disappoints clients, thereby damaging the company's reputation. Customers require applications that are responsive, reliable and saves time. This can be taken care of by analyzing code and looking for areas of code that run at peak performance. A common property of computer programs is described by the 90/10 rule: a program spends approximately 90% of its execution time in approximately 10% of the (executed) static program. Static program could be found in the form of loops or recursive functions. Optimizing these 'hot spots' results in reduced overall application execution time or reduced use of memory resources.

Using a profiling tool helps in locating such bottlenecks within the code and thus diagnoses application performance. The IBM Rational Application Developer provides sophisticated profiling tools that can be used in the development phase itself. This gives developers time to make necessary changes to avoid performance issues from becoming critical problems later on.

II. Profiling in RAD

RAD profiling offers two main profiling modes: trace-based profiling and sample-based profiling. Understanding the difference between each of these modes helps in making the right decisions while testing the performance of applications. A high-level description of the mechanism with which both profiling modes operate and the pros and cons of each profiling method will be discussed. Simultaneously, a walk-through of the different profiling options available in RAD will be provided.

In RAD, on selecting the “Profile Java application” option, a profiling configuration window appears as shown in Figure 1. The Java Profiling option use a trace-based profiler, while the Health Center Data Collector option uses a sample-based profiler.
III. Trace-based profiling

In trace-based profiling, the profiling agent running alongside the JVM actively instruments methods in the JVM to find how long each method takes to run. The JVM notifies the profiler when a method is entered and exited and therefore provides an accurate measure of the number of calls and time spent in executing a particular method. The RAD JVMTI profiler is a trace-based profiler. It is used to detect:

(i) performance bottlenecks, in terms of execution time (execution analysis),

(ii) memory usage problems (memory analysis), and

(iii) Contention analysis and deadlock detection (thread analysis).
The drawback of this method appears while profiling a large code base. Transmission of data from the JVM to the profiling agent, processing of the data by the workbench at regular intervals and storage of all data in memory for statistics generation leads to slower speed and large overhead during profiling. A workaround to this problem is to set filters during profiling. To set filters, click on the Edit Filters option (see Figure 2), which opens a window as shown below. The packages or classes to be profiled can be added in here. This helps you to collect data of interest and focus on analysis of your code.

Figure 2: Filter Set window
(i) **Execution Analysis view**

In the Package view, you can expand the packages and view the time taken to execute each class. The following statistics are displayed for each package, class or method:

(i) The **Base Time** refers to the time that is taken to execute all method invocations only. This does not include the time spent in each of the called methods.

(ii) The **Average Base Time** is the base time divided by the number of calls.

(iii) The **Cumulative time** is the time that is taken to execute the invocation, including the time that is spent in the called methods.

(iv) **Calls** refers to the number of calls that are made to the package, class or method.

Execution Analysis also provides information regarding call tree, details about method invocation and number of times a method has been invoked.
Call Tree

The Call Tree view displays the function execution paths that were traversed in the profiled application. The root of the tree is the entry point into the application. Each function node lists all the functions it called and performance data about those function calls.

Method Invocation Details

Method Invocation Details include information about a specific method. It tells two things:
(i) Which methods invoked the selected method
(ii) Which methods are invoked by the selected method

For example in Figure 5 the selected method is "getProducts()"، it is invoked by "testProducts()" but it doesn't invoke any other methods in the execution path that was taken.
Method Invocation Details

Selected method

<table>
<thead>
<tr>
<th>Calls</th>
<th>Method</th>
<th>Class</th>
<th>Package</th>
<th>&lt;Base Time (s)</th>
<th>Average Base</th>
<th>Cumulative Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>getProduct()</td>
<td>com.ibm.storeapp</td>
<td>Store</td>
<td>0.000005</td>
<td>0.000005</td>
<td>0.000005</td>
</tr>
</tbody>
</table>

Selected method is invoked by:

<table>
<thead>
<tr>
<th>Invoked by</th>
<th>Method</th>
<th>Class</th>
<th>Package</th>
<th>&lt;Base Time (s)</th>
<th>Average Base</th>
<th>Cumulative Time</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>testProducts()</td>
<td>void</td>
<td>Store</td>
<td>0.001451</td>
<td>0.001451</td>
<td>0.017781</td>
<td>1</td>
</tr>
</tbody>
</table>

Selected method invokes:

<table>
<thead>
<tr>
<th>Invokes</th>
<th>Method</th>
<th>Class</th>
<th>Package</th>
<th>&lt;Base Time (s)</th>
<th>Average Base</th>
<th>Cumulative Time</th>
<th>Calls</th>
</tr>
</thead>
</table>

Figure 5: Method Invocation Details

- **Method Invocation**

Method Invocation provides information about number of times a specific method has been invoked and each invocation performance data is displayed as well.

For example in Figure 6 the selected method is "getProducts()", it is invoked one time during the execution path taken.
The Object Allocations view shows statistics about the application heap. It provides information on the number of classes loaded, the number of instances that are alive and the memory size that is allocated by every class.

(i) The **Live Instances** column shows the number of instances of the selected package, class or method where no garbage collection has taken place.

(ii) The **Active Size** column shows the total size in bytes of all live instances.

(iii) The **Total Instances** column shows the total number of instances that have been created of the selected package, class, or method during the JVM's lifetime.
(iv) The **Total size** column shows the total size in bytes of the selected package, class, or method, of all instances that were created for it during the JVM's lifetime.

(v) The **Average Age** column shows the average number of garbage collections that objects of the class have survived.

Memory Analysis View also consists of a tab called "Allocation Details"

- **Allocation Details**

![Allocation Details](image)

Allocation Details shows information about which methods are allocating memory for which classes along with information on the number of classes loaded, the number of instances that are alive and the memory size that is allocated for each class.

For example in Figure 8, a method called "Store" is allocating memory for class "Customers".
(iii) *Thread Analysis View*

**Figure 9: Thread Analysis view**

**Figure 10: Threads Visualizer view**

The *Thread Statistics* view shows the state, running time, and waiting time of the threads. The *Monitor Statistics* tab shows the details of a selected thread including the Java classes that are involved. The *Thread Visualizer* tab shows the timeline of the threads with their states.
IV. Sample-based profiling

Sample-based profiling tools collect snapshots of the current call stacks from the JVM at regular intervals. A set of samples of methods running on the JVM is generated. This method is not as accurate as trace-based profiling and reflects a statistical approximation of a method's execution time. Moreover, we do not obtain information of the number of method calls. However, this kind of profiling is faster and has very low overhead as compared to trace-based profiling and is therefore good to use when profiling applications that are CPU bound.

Sample-based profiling using the Health Center Agent has been newly added to RAD v9.0. This is the same Health Center agent that is available on IBM Support Assistant (ISA)

Figure 11: Health Center Agent Profiling Sample Data view

The following figure shows the Health Center Agent view in RAD:

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Self</th>
<th>Tree %</th>
<th>Tree</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12.9</td>
<td>19.4</td>
<td></td>
<td></td>
<td>com.ibm.otl.vm.BootstrapClassLoader.loadClass(java.lang.Class)</td>
</tr>
<tr>
<td>2</td>
<td>6.5</td>
<td>6.5</td>
<td></td>
<td></td>
<td>com.ibm.otl.vm.VM.findClassOrNull(java.lang.String)</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>java.io.ObjectInputStream$BlockInputStream$ReplaceTable$lookupObject</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td></td>
<td>com.ibm.java.diagnostics.healthcenter.agent.dataproc...</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>com.ibm.java.diagnostics.healthcenter.agent.data...</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td></td>
<td>java.net.ServerSocket.getInetAddress()</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>java.lang.Class.forName$CacheKey.equals(java.lang.Object)</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>com.ibm.lang.management.ThreadMXBeanImpl.get...</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>sun.nio.cs.StreamDecoder$Init$java.io.InputStream$Reader</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>sun.nio.cs.StreamEncoder$implWrite(char[], int, int)</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td></td>
<td>java.util.concurrent.ConcurrentQueue$Transferable$transfer$...</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td>java.io.ObjectInputStream$BlockDataInputStream$readChar</td>
</tr>
<tr>
<td>1</td>
<td>3.2</td>
<td>3.2</td>
<td></td>
<td></td>
<td>sun.nio.CharToByte.UTFB.convert(char[], int, int, byte[], int)</td>
</tr>
</tbody>
</table>

The Health Center Profiling Sample Data view shows which methods are using the most processing resources. The objects shown higher on the table need to be optimized in order to improve the application's overall execution time. The following data is displayed:

(i) The **Method** column shows the fully qualified method name.

(ii) The **Samples** column shows the number of samples taken while a particular method was being run at the top of the stack.

(iii) The **Self %** column shows the percentage of the number of samples taken while a particular method was being run at the top of the stack.

(iv) The **Tree %** column shows the percentage of the samples taken while a particular method was anywhere in the call stack.

(v) The **Self** and **Tree** columns are bar-representations of the **Self %** and **Tree %** columns respectively.
V. Conclusion

By providing support for both trace-based and sample-based profiling, Rational Application Developer has provided users with a comprehensive profiling tool that can be used to diagnose a wide array of performance issues. In today's world, high performance software applications are in demand, and RAD's profiling tool can be of great help to software developers in improving the performance of their software applications.