NetBeans Profiler is an optional feature of the NetBeans IDE. It is a powerful tool that provides important information about the runtime behavior of an application. Imposing relatively little overhead, the NetBeans Profiler tracks thread state, CPU performance, and memory usage. It uses innovative technology to allow you to tightly control exactly which parts of an application are profiled, resulting in reduced overhead and easier to interpret results. The profiled application can run locally or on a remote system. And by being tightly integrated into the IDE workflow the NetBeans Profiler makes it easy to identify performance problems and memory leaks.

Installing the NetBeans Profiler

The NetBeans Profiler can profile applications running on the following JVMs:
- A customized JFluid VM, based on the standard JDK 1.4.2 VM. This customized JVM is available as a separate NetBeans module download.
- A standard JDK 5.0_04 VM or newer.
- A standard JDK 6 (also known as Mustang) Early Access VM starting from build 26.

Here are the steps for installation:
1. Download the profiler pack installer for your platform from the NetBeans Profiler homepage.
2. If you have a previous version of
the NetBeans Profiler installed, uninstall it first – see the instructions on the download page.

3. Launch the downloaded installer.

4. Proceed through the steps of the installer wizard to install the module.

5. (Re)Start the NetBeans IDE.

Once installed, the module adds the Profile menu item to the menu bar. The Profiler menu allows you to start profiling and work with the results. The module also adds two toolbar buttons that are shortcuts to the Profile command ( ) and the Attach and Profile command ( ).

Getting Started

The NetBeans Profiler provides a number of internal settings that let you tune profiling to your needs. For example, you may decrease the profiling overhead at the cost of some reduction in the amount of generated information. However, it may take some time to understand the meaning and use of the numerous settings available.

For most applications, certain default settings are sufficient. For this reason, the NetBeans Profiler offers two major profiling options. You can start profiling by choosing a simple predefined profiling task, which has most of the settings preset to optimal values and therefore requires little or no tuning. Alternatively, you can create your own custom profiling configuration, where you are free to modify any of the available settings.

To start profiling an application:

1. Open your project in the NetBeans IDE.
2. Right-click your project’s entry in the Projects window and then select Set Main Project.
3. Choose Profile|Profile Main Project from the main menu.
4. Choose a profiling command from the list in the Select Profiling Task dialog box.
5. Click Run.

When you click Run, the target application launches and the selected profiling command starts. The NetBeans Profiler Control Panel opens in the IDE.

To see the results of the profiling command, click the Live Results button ( ) in the Profiler Control Panel. This opens the Profiling Results tab. To stop the profiling command, choose Profile|Stop from the menu or click the Stop button ( ). If you start the application with the NetBeans Profiler, when you stop the profiling command the application also stops.

Profiling Tasks

The Select Profiling Task dialog box (Figure 1) is the main interface for selecting and running profiles. The dialog box gives you five different ways of profiling the target application. The first four are predefined profiling commands. The last command, Run Custom Profiling, allows you to create your own custom profiling configuration.

When you click on a profiling command the box expands, displaying a brief explanation. For some tasks it also allows you to set some profiling options. Clicking Run at the bottom of the dialog box launches the target application and starts the selected profiling command.

Only one task, i.e. one kind of profiling, can be active for the profiled application at any given time (monitoring is always active however, even when another task is chosen, since its overhead is very low). Note that while profiling you can switch between...
profiling tasks without stopping and restarting your application.
You can choose from the following profiling tasks:

**Monitor Application**
This command displays high-level information about several important properties of the target JVM, including thread activity and memory usage.

**Analyze Performance**
Profiles method-level CPU performance (execution time). You can choose to profile the entire application or a part of the application. Detailed filters can be set to control exactly which methods get profiled, allowing the rest of your application to run at full speed.

- **Entire Application**: In this mode, the NetBeans Profiler instruments all of the methods of the profiled application. Threads emit the “method entry” event when entering a method and generate the corresponding “method exit” event when exiting the method. Both of these events contain timestamps. This data is processed in real time.

- **Part of Application**: In this mode, you can instrument and profile a limited subset of the application’s code. When partial application profiling is used, profiling data is not collected until one of the application’s threads enters a user-selected root method. Profiling a limited subset of the application’s code may greatly reduce the profiling overhead. Furthermore, for some programs this option may be the only way to obtain any detailed and/or realistic performance data at all – because the amount of generated data when the entire application is profiled can be so high as to render the application unusable or even cause it to crash (for example, due to unexpected timeouts).

**Application Startup**
Use this mode when you want to analyze application startup time.

**Analyze Code Fragment Performance**
This command measures the time it takes to execute an arbitrary piece of code within one method, or the whole method. By analyzing a code fragment rather than the entire application, the profiling overhead is greatly reduced. The absolute results that you obtain in this mode are closest to the real runtime performance, since the instrumentation is the most lightweight.

**Analyze Memory Usage**
When you analyze memory usage, the profiling results displayed depend on which of the following options you choose:

- **Record both object creation and garbage collection (Object Liveness)**: This option provides information about how many objects of each type are still alive, as well as data on live objects.

- **Record object creation (Object Allocation)**: Gives you information about the number, type, and location of objects that have been allocated. This profiling mode is a functional subset of object liveness profiling. The reason for having both modes is that pure object allocation profiling has a smaller performance and memory overhead.

By default only ten percent of all objects for each class are tracked by the NetBeans Profiler. This statistical approach has been shown to deliver results that are as accurate as when all objects are tracked, but with the benefit of greatly reduced profiling overhead.

**Run Custom Profiling**
This command is for running custom profiling configurations. You can create, edit, and save these custom configurations, allowing you to control all the internal profile settings.

**Control Panel**
The profiling control panel is displayed in the left pane of the IDE when you run the NetBeans Profiler (see Figure 2). You can open the control panel by choosing Window |Profiling>Profiler Control Panel. It contains controls that do the following:
Exploring the NetBeans Profiler

To control the profiling task, display the status of the current profiling task, display profiling results, manage profiling results snapshots, and display basic telemetry statistics.

Displays

The NetBeans Profiler provides several displays of information. The VM Telemetry Overview is always displayed when the Monitor Application command is chosen. To display it at any other time, select Profile|View>Telemetry Overview.

See an example in Figure 3. In the graph on the left the red shading indicates the allocated size of the JVM heap. The purple overlay indicates the amount of heap space actually in use. In the example the allocated heap size at the last update was over 300 Mb. Of that about 20 Mb is actually being used to hold Java objects.

- The graph on the right shows the count of active threads in the JVM.
- The graph in the center shows two important heap statistics.
- The blue line is the percentage of execution time spent by the JVM doing garbage collection and is graphed against the y-axis on the right edge of the graph. Time spent by the JVM doing garbage collection is time that is not available for it to run your application. So if the blue line indicates a large percentage you may want to consider tuning the JVM by configuring a larger heap size (refer to the \-Xmx parameter documentation) or perhaps switching to a different garbage collection algorithm.
- The red line is surviving generations and is graphed against the y-axis scale on the left edge of the graph. The count of surviving generations is the number of different ages of all the Java objects on the JVM’s heap, where ‘age’ is defined as the number of garbage collections that an object has survived. When the value for surviving generations is low it indicates that most of the objects on the heap have been around about the same amount of time. If, however, the value for surviving generations is increasing at a high rate over time then it indicates your application is allocating new objects while maintaining references to many of the older objects it already allocated. If those older objects are in fact no longer needed then your application is wasting (or ‘leaking’) memory.

Figure 2
The NetBeans Profiler Control Panel

Figure 3
VM Telemetry Overview
Thread State

Thread state is optionally displayed when the Monitor Application command is chosen. It contains the following tabs:

- **Threads (Timeline):** Shows current and historical thread state, updated as the application runs.
- **Threads (Details):** Shows a summary of thread state information for a single thread.

A sample timeline graph is shown in Figure 4. Color coding is used to display thread state:

- **Green:** the thread is either running or is ready to run.
- **Purple:** the thread is sleeping; for example it called `Thread.sleep()`.
- **Yellow:** the thread is waiting in a call to `Object.wait()`.
- **Red:** the thread is blocked while trying to enter a synchronized block or method.

Live Results

Clicking the Live Results button ( ) in the control panel will open the Profiling Results tab. Depending on the profiling command that is running, this tab will display either performance or object allocation and liveness statistics.

The NetBeans Profiler will update the displayed profiling results automatically at short intervals (about 2 seconds) if the Update Results Automatically button ( ) in the toolbar is clicked.

CPU Snapshot

The CPU Snapshot captures data on method call chains, times, and invocations when profiling CPU performance, either from the Analyze Performance predefined task or when running a custom CPU profile. The CPU Snapshot is displayed when you click the Take Snapshot button ( ) in the control panel. See an example in Figure 5.

The CPU Snapshot contains the following tabs:

- **Call Tree** – The Call Tree tab displays the Calling Context Tree (CCT) showing the method call chain and the time/number of invocations for executing threads and methods in each context. (A context is a unique chain of method calls leading to the method's invocation.)
- **Hot Spots** – The Hot Spots tab shows the total execution time and number of invocations for each method, irrespective of the context.
- **Combined** – The Combined tab displays the CCT information in the upper half of the window and the Hot Spot data in the lower half.
- **Info** – The Info tab displays data on when the snapshot was taken, where it is saved, and the profile settings used.

Memory Snapshot

The Memory Snapshot captures data on object allocation and liveness when profiling memory usage with the Analyze Memory Usage command or when doing memory profiling in a custom profile. Like the CPU Snapshot, the Memory Snapshot is displayed when you click the Take Snapshot button ( ) in the control panel.

The Memory Snapshot contains the following tabs:
Memory Results

This tab displays a list of classes (including array classes), with statistics on the total size and number of instances allocated as of the time you took the snapshot (see an example in Figure 6). The specific information provided includes:

- **Allocated Objects** is the number of objects that the NetBeans Profiler is actually monitoring. By default this number will be approximately ten percent of the value of total allocated objects. By monitoring only a subset of the created objects the NetBeans Profiler is able to dramatically reduce the overhead it places on the JVM, which then allows your application to run at close to full speed.

- **Live Objects** is the number of the Allocated Objects that are still on the JVM's heap and are therefore taking up memory.

- The two **Live Bytes** columns show the amount of heap memory being used by the Live Objects. One column displays a graph; the other displays text.

- **The Avg. Age value is calculated using the Live Objects. The age of each object is the number of garbage collections that it has survived. The sum of the ages divided by the number of Live Objects is the Avg. Age.**

- **The Generations value is calculated using the Live Objects. As with Avg. Age, the age of an object is the number of garbage collections it has survived. The Generations value is the number of different ages for the Live Objects. An increasing value for Generations indicates a possible memory leak.**

In the Memory Results tab, you can right-click any class and select Show Allocation Stack Traces. The stack traces for the selected class are displayed in the Allocation Stack Traces tab.

- **Allocation Stack Traces**

  This tab displays a reverse call graph with all call paths leading to object allocations for the given class.

- **Info**

  The Info tab displays data on when the snapshot was taken, where it is saved, and the profile settings used.

---

Figure 5
CPU Snapshot showing Calling Context Tree information and Hot Spot data

Figure 6
The Memory Results tab shows statistics on the total size and number of instances allocated
Profiling example

The following example demonstrates one important feature of the NetBeans Profiler: the ability to quickly identify CPU performance problems. The example is a web application that calculates prime numbers. When attempting to find performance bottlenecks, you typically know which features are running slowly. That allows you to narrow down the search for the bottleneck to a top-level method for that feature. The NetBeans Profiler supports this by allowing you to specify a root method for profiling.

1. A profiling session begins by selecting a project as the IDE's Main Project. Then Profile | Profile Main Project is chosen from the IDE menu.

2. The Select Profiling Task dialog is displayed.

3. The Analyze Performance button is clicked.

4. The Part of Application radio button is clicked. Then the Select button is used to select the class that contains the root method. In this case the class is demo.Performance and the method is processRequest() – see Figure 7. This means that the demo.Performance.processRequest() method and all methods that it calls, and all methods that they in turn call (and so on) will be profiled. Starting from demo.Performance.processRequest(), the Profiler does analysis of the method call graph to determine which methods need profiling. Only those methods are profiled – the rest of the application will continue to run at full speed with no profiling overhead.

5. Particularly when profiling web or enterprise applications, there are usually large blocks of code that you do not want to profile. In this example, the web server is Tomcat and there is no need to do profiling of Tomcat's code. So in the Analyze
Performance window, the Quick Filter is used to specify methods that should not be profiled. The string “org.apache” is specified so that all methods in the org.apache package (and child packages) will not be profiled – even if they are called from the root method that was selected (see Figure 8). This reduces profiling overhead and filters out information that is not relevant.

6. Clicking the Run button in the Select Profiling Task window starts the profiling session. The IDE will start Tomcat and display the web application’s index.jsp page in a web browser window. At the same time, the Profiler will run in the background.

7. The portion of the web application that causes the root method to run is then invoked by interacting with the application’s user interface.

8. After the application responds, the Profile|Take Snapshot of Collected Results command is selected in the IDE. The Profiler displays the performance results, as illustrated in Figure 9.

9. The top window shows the complete method call graph beginning with the root method. The bottom window is a flatter depiction; it shows the Hot Spots in the application – those methods that took the most time to execute.

10. To examine and interpret the results, notice that the processRequest() method ran for a total of 4308 milliseconds (ms). Note, however, that very little time was spent running the instructions of the processRequest() method itself – the “self time” for processRequest() is only 10.1 ms. The vast majority of the time was spent in methods called by processRequest(). The Hot Spots displayed in the bottom window are sorted by “self time.” By looking at that list you can see that the calculate() method took up 97.8% of the execution time. This is not surprising given the amount of work the calculate() method has been given to do and the inefficient way it goes about doing that work.

11. To help you decide how your application can be optimized, the NetBeans Profiler helps you identify bottlenecks in your code that were not expected or that will prevent your application from scaling well. From here, it is possible to right-click the calculate() entry and choose Go To Source in order to examine the source code. As a comparison to calculate()’s runtime, the Profiler output of an optimized algorithm in a method called calculate2(), is shown in Figure 10. Notice that the processRequest() method ran for only 107ms and the calculate2() method took up less than 10% of the execution time!

Conclusions

The NetBeans Profiler is a powerful tool that provides important information about the runtime behavior of an application. It can be used to identify thread state problems, CPU performance bottlenecks, and memory usage bugs.

Figure 10
Profiler results after method optimization

Gregg Sporar (gregg.sporar@sun.com) has been a software developer for over twenty years, working on projects ranging from control software for a burglar alarm to 3D graphical user interfaces. He has been using Java since 1998 and his interests include user interfaces, development tools, and performance profiling. He works for Sun Microsystems as a Technical Evangelist on the NetBeans project.