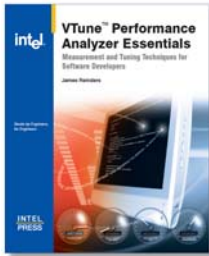


# VTune™ Performance Analyzer Essentials

The complete book is available from [shopintel.com](http://shopintel.com) at a special discount for VTune™ analyzer users. Click the book cover and enter the promotional code "vta2006" at the [shopintel.com](http://shopintel.com) checkout.



## VTune™ Performance Analyzer Essentials Measurement and Tuning Techniques for Software Developers

by James Reinders, Published March 2005  
Part Number: ISBN 0-9743649-5-9  
Recommended Price: **\$59.95**

A real challenge in modern software environments is the ability to properly identify performance bottlenecks. The Intel® VTune™ Performance Analyzer helps locate and remove software performance bottlenecks by collecting, analyzing, and displaying performance data from the system-wide level down to the source level. VTune Performance Analyzer Essentials is written for software application developers, software architects, quality assurance testers, and system integrators who wish to take the guesswork out of software tuning. Much like diagnostic computers for tuning engines, or flashlights for seeing plumbing in the dark reaches of your basement, the tools within the VTune analyzer "illuminate" your system and everything running on it. This book is a guide to "turning on the lights" and understanding what you see.

Included are a wide range of examples and step-by-step techniques that illustrate the VTune analyzer in action.

### *Highlights Include:*

- Hotspot hunting and automatic analysis
- Software tuning guidelines for different languages, such as C++, Fortran, Java, Microsoft Visual Basic, and Microsoft C#
- Automation of analysis tasks
- Remote analysis techniques for "headless" servers, PDAs, and cell phones
- How to analyze multithreaded programs

A special companion Web site to this book contains all code examples and bonus material, plus trial versions of Intel software development products including the VTune Performance Analyzer.

---

**James Reinders** is a senior engineer who has spent the past 16 years at Intel Corporation working on projects such as the world's first TeraFLOP supercomputer (ASCI Red) and on the compilers and architectures for the Pentium® Pro, Pentium® II, Itanium®, Pentium® 4, and iWarp processors. James is currently the director of business development and marketing for Intel's Software Products Division and serves as the division's chief product evangelist.



Intel Press books are essential for computer product Developers and IT professionals because they are timed with industry roadmaps. Our books are a simple way to learn from the experts about the latest technologies from Intel.

Visit our website at [www.intel.com/intelpress](http://www.intel.com/intelpress)

# VTune<sup>™</sup> Performance Analyzer Essentials

Measurement and Tuning Techniques  
for Software Developers

James Reinders

Copyright © 2005 Intel Corporation. All rights reserved.

ISBN 0-9743649-5-9

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Publisher, Intel Press, Intel Corporation, 2111 NE 25<sup>th</sup> Avenue, JF3-330, Hillsboro, OR 97124-5961. E-Mail: [intelpress@intel.com](mailto:intelpress@intel.com).

Celeron, Intel, Intel Centrino, Intel logo, Intel NetBurst, Intel Xeon, Intel XScale, Intel SpeedStep, Itanium, MMX, Pentium, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

† Other names and brands may be claimed as the property of others.

**INTEL**  
**PRESS**

These pages were excerpted from Chapter 4 of *VTune™ Performance Analyzer Essentials* by James Reinders.

Visit Intel Press on the web at [www.intel.com/intelpress](http://www.intel.com/intelpress) to learn more about this book.

In this section of the book, the author walks through an example on how to use the graphical user interface to find the hotspots in your application down to the lines of source code. More examples and detail on how sampling is implemented are contained later in the same chapter as this excerpt.

---

#### Example 4.1: Time-Based Sampling Using the Graphical Interface

Follow these steps to perform time-based sampling from the graphical interface.

##### Step-by-Step Procedure

1. Double-click the **VTune Performance Analyzer** icon to start the analyzer.
2. Click the **Create New Project** icon.
3. Click **Sampling Wizard** and select **OK**.
4. Select **Windows/Windows CE/Linux Profiling** and click **Next**.

5. In the **Application To Launch** text box, type the path:

C:\examples\ch4\windows\prog0401.exe  
(Linux users: examples/ch4/linux/prog0401)

or whatever executable file you want to sample, possibly:

C:\WINNT\system32\notepad.exe or /bin/ls

6. In the **Command Line Arguments** text box, type:

input0402.txt 1000

for prog0401. For other applications, either type the options you want or leave it blank if you're not sure.

7. Select the **Modify default configuration when done with wizard** checkbox and click **Finish**.

The Advanced Activity Configuration dialog box appears.

8. Click **Configure...** in the middle of the screen for the Data Collector called Sampling, which is already selected.

The Configure Sampling window appears.

9. Select **Time-based sampling (TBS)** and click **OK** to return to the Advanced Activity Configuration dialog box.

10. Click **OK**.

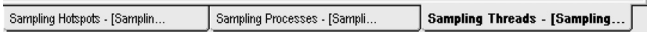


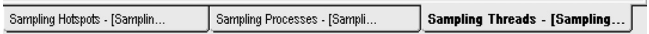




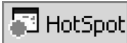
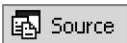
Once prog0401 finishes executing, the VTune analyzer displays a screen showing the applications detected as running while Sampling took place. If you are running your own application or Notepad, you are probably running with a time limit, because this is the default setting. If so, either close that program or click the **Stop Activity** icon on the toolbar (Figure 4.3) in order to see the Sampling results screen.



**Figure 4.3** The Stop Activity Icon on the Toolbar

Table 4.2 shows some important toolbar icons for navigating among Sampling data views.

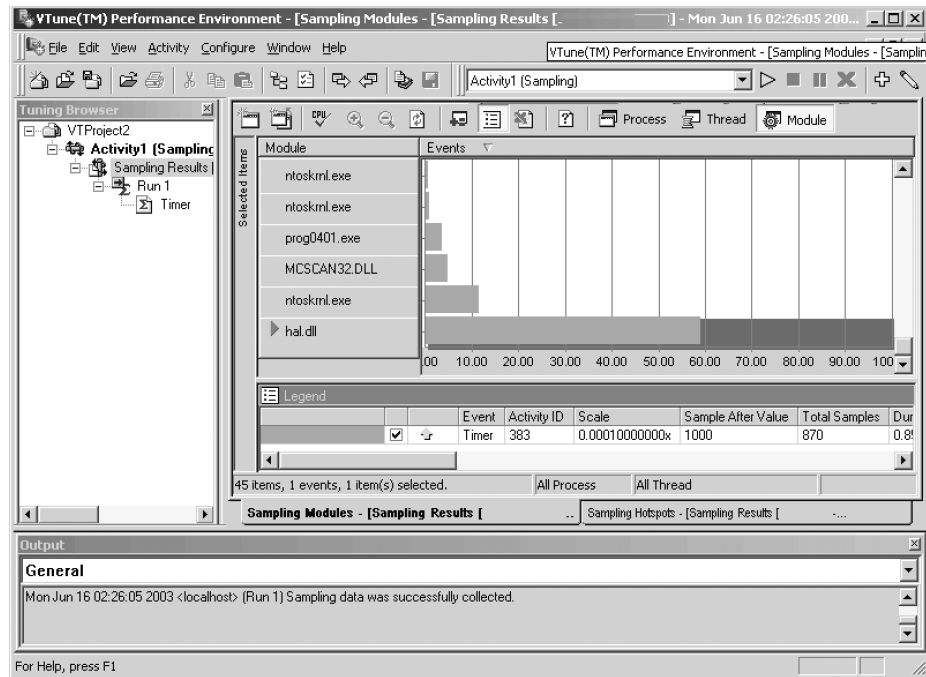
**Table 4.2** Important Icons for Navigating Sampling Views (Windows)

Name	Icon	Description
Tabs	See description.	Click tabs (usually positioned at the bottom of the screen) to switch between open data views: 
Previous (Standard Toolbar)		Displays the previous open data view window. For example, if you drilled down from Module to Hotspot view, clicking <b>Previous</b> would take you back to Module view, much like the <b>Back</b> button in an Internet browser.
Next (Standard Toolbar)		Displays the next open data view window. For example, if three views are open:  clicking <b>Next</b> would take you to the next view in the series, much like the <b>Forward</b> button in an Internet browser.
Drill Down		Drills down to a lower level in the data view hierarchy by opening the selected item (such as a process, thread, module, or function) in a new window.
Processes	 Process	Displays all the processes that ran on your system when Sampling data collection took place.
Threads	 Thread	Displays the threads for the selected process(es). <sup>1</sup>
Modules	 Module	Displays the modules for the selected thread(s). <sup>1</sup>
Hotspots	 HotSpot	Displays the hotspots for the selected module(s) grouped by function, relative virtual address, source file, or class. <sup>1</sup>
Source	 Source	Displays the source code for the selected hotspot(s). <sup>1</sup>

### Which Application Took the Most Time?

Looking at Figure 4.4, which application took the most time? Notice that it was not prog0401, at least not on the system used to prepare this example.

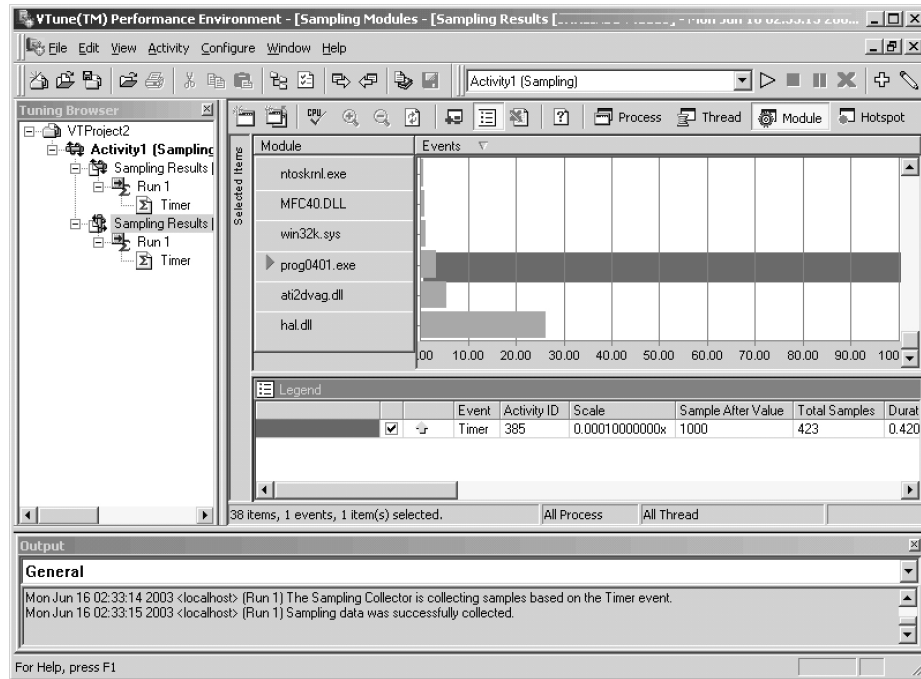
<sup>1</sup> Hold down the Ctrl or Shift key to make multiple selections.



**Figure 4.4** Sampling Results for prog0401 with Virus Protection Software

Instead, the OS (hal.dll and ntoskrnl) plus the virus protection software (MCSCAN32.DLL) appear to be the culprit. Linux users, you will see shortly, have a similar issue with Java on their system. This may seem frustrating for the purposes of this example, but it illustrates the power of the VTune analyzer. The analyzer is telling you that, if you must make this particular system run faster, the biggest problems with this particular example are the OS and the virus protection software (or perhaps Java on Linux) that analyzes prog0401.exe before it ever runs.

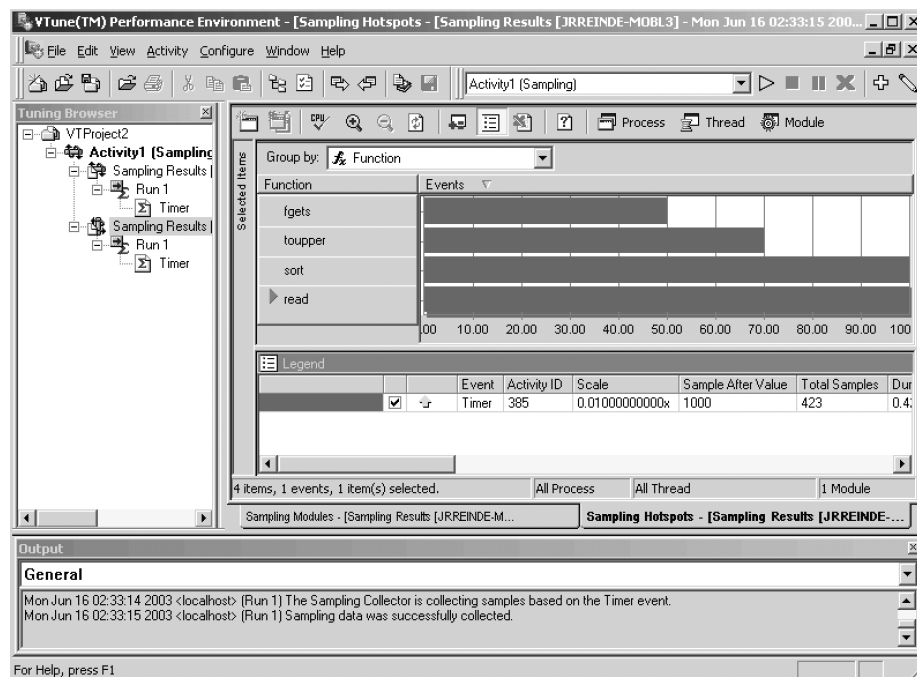
If you can eliminate that software, you can speed up system execution more than if you worked on prog0401 all day long. Take a look at the results of running the prog0401 program again after disabling the virus protection software; in other words, rerun Example 4.1. The screen shown in Figure 4.5 tells the tale; MCSCAN32.DLL has disappeared, hal.dll is now taking less than half the execution time, and ntoskrnl is taking almost no time.



**Figure 1.5** Sampling Results for prog0401 Without Virus Protection Software

You can drill down to the function level next to see which functions in prog0401 took the longest to run. To do this, simply double-click on the dark horizontal bar to the right of the filename prog0401.exe. Before you click, while your mouse is pointing at the dark bar, you might notice that the analyzer displays a small box with the information Timer 32,000. The VTune analyzer is showing you the number of timer interrupts that occurred while prog0401 was the active process.

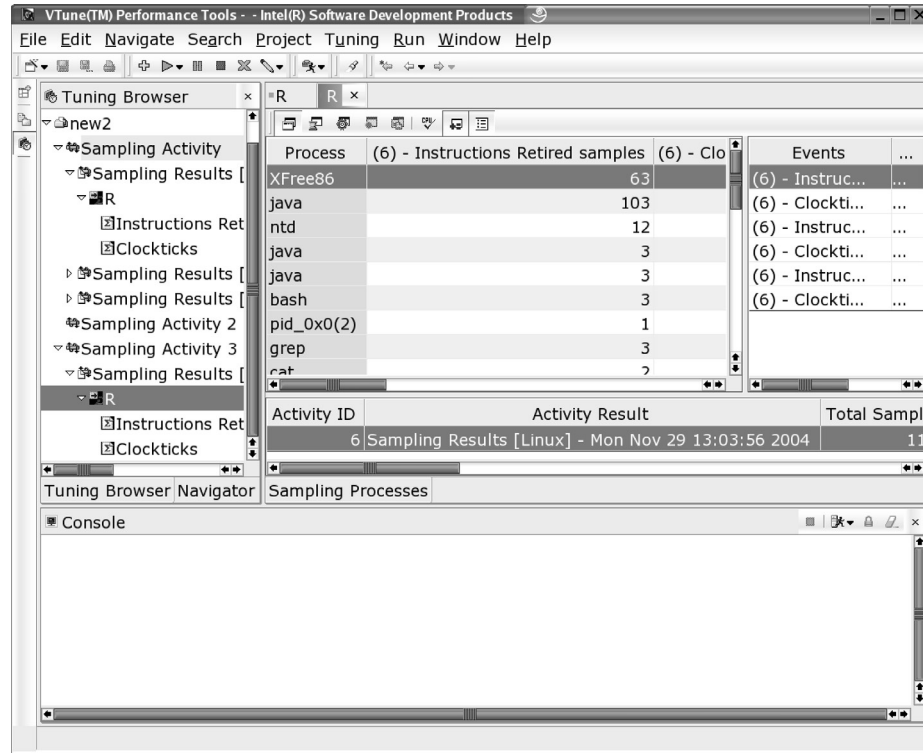
After drilling down you see the view shown in Figure 4.6, which clearly shows that the top three functions in terms of time spent are fgets and toupper (parts of the C library), plus sort and read, which are in the prog0401 code. By mousing over the dark bar for the read function, you can see the message Timer 10,000. This means that prog0401 spent about 31 percent of its run time in the read function (10,000/32,000).



**Figure 4.6** Functions View of prog0401

Linux users may not have virus protection software running, but they will see that Java is consuming a lot of resources. The Eclipse user interface uses Java, so it is running on the system even though the example program itself has no Java code. The resulting analysis shown in Figure 4.7 has the example program not even ranking in the first screenfull as it was too trivial a task on the system. The VTune analyzer is offering its analysis as to what is really happening on the computer; it is up to you to use or ignore the wealth of information as it suits your purposes.



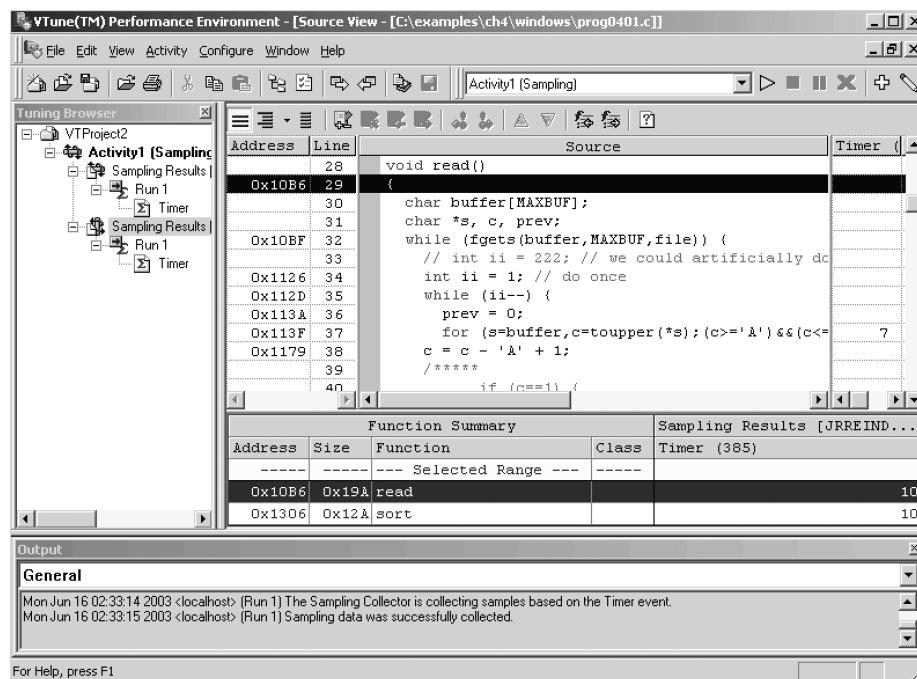


**Figure 4.7** Sampling Results for prog0401 on Linux with Java Dominating

### Drilling Down to the Source

You can take your analysis of the prog0401 program one step further by drilling down to the source code to look for hotspots at the individual-line level. Simply double-click on any horizontal bar next to the read function and the analyzer displays the source code. If a dialog box pops up telling you that source code is not available, you have the option of looking at assembly language, telling VTune analyzer the directory with the source code in case it is located in an unusual place, or revisiting the instructions in the earlier section entitled “Before You Begin Sampling,” that explain how to compile your program so you can view the source code.

Figure 4.8 shows the Source Code view of the prog0401 program. Mouse over the column heading for *Timer*. Notice that the analyzer shows information about this column. Right-click on the column heading and take note of the many options. Selecting the **What's This Column?** option is a very powerful way to see the VTune analyzer's built-in documentation on this event. Another important option is **View Events As**, which controls the numbers shown in the column; the default view is **Sample Counts**. Since VTune analyzer Sampling works by interrupting your program occasionally, this option shows you how many actual interrupts occurred. In the prog0401 example, the Timer occurred 10 times (7+3) in the for loop in the read routine. Seven of these happened to land on line 37 of the program, and three on line 45 (not shown). You can choose to change the column to display **% of Module**, which is often more meaningful for users who prefer to look at performance data in terms of percentages, rather than sample counts.



**Figure 4.8** Source Code View of prog0401

These pages were excerpted from Chapter 5 of *VTune™ Performance Analyzer Essentials* by James Reinders.

Visit Intel Press on the web at [www.intel.com/intelpress](http://www.intel.com/intelpress) to learn more about this book.

In this section of the book, the author explains key concepts of the Call Graph features in the VTune analyzer: wait time, advice in using call graph vs. sampling to find hotspots, and an explanation of how Intel's tool actually implements Call Graph analysis.

## Blocked Time on a Thread/Function Basis

Of special interest is time spent waiting because a thread is blocked from execution. Call Graph can show this information, which can be quite valuable in fine-tuning an application through better synchronization.

### Does My System Show Wait Time?

Certain fields related to Wait Times may always display as zero on your computer, depending on your operating system. Collection of exact Wait Time information is not supported when running on a single processor. This feature is exclusively available on multiprocessors or processors with Hyper-Threading Technology. If your computer does show nonzero Wait Times, be aware that:

- *Self-Time*, the time spent inside a function excluding its children, does include Wait Time during which the thread is blocked (suspended).
- *Self Wait Time* shows time spent inside a function while its thread is blocked.
- *Total Wait Time* shows time spent in a function including its children during which the thread is blocked.

### Why Do Sampling and Call Graph Have Different Hotspots?

If you collect the time for a hotspot function using Sampling and then collect the time for that same function using Call Graph, the two times may not match. Sampling, whether event-based or time-based, does not include Wait Time during which the thread is suspended waiting for another thread to complete, whereas function Self-Time in Call Graph does include Wait Time. If you calculate non-blocked Self-Time by subtracting Self Wait Time from Self-Time, the hotspots for both Sampling and Call Graph data collectors should be the same.

### Finding High-level Inefficiencies Related to Hotspots

Call Graph can help you understand where and why hotspot functions are being called. Look for redundant (wasted) work. For example, you may find that a hotspot function is being called multiple times to perform the same calculation. In this case, you could modify the code so that the hotspot routine is only called once and the results are stored.

Use Call Graph's list of frequently called functions, and note the number of times functions are called and the amount of time spent in each function. Look for measurements that do not make sense given your understanding of how the application works. When you see these anomalies, track them down to understand what is happening. Often anomalies are caused by undetected bugs that may be decreasing performance.

### Call Graph Usage Tips

Call Graph analysis is used to get an algorithmic or high level look at the program, whereas sampling is best used when you want to analyze a particularly dominant hotspot in more detail, or when you want to tune the application more precisely for the CPU architecture. Often Call Graph analysis can be useful when the Sampling profile is “flat,” when no single hotspot is dominant. Sampling analysis was covered in Chapter 4.

When initially using the VTune analyzer, you should run Sampling first and see whether a hotspot is dominant. If so, drill down to source to analyze it in more detail. If not, it is time for Call Graph analysis. Often, data-processing applications are better suited for Call Graph analysis than programs that employ loop-based calculation, since it is less likely that the data processing applications are going to have only a small number of significant hotspots. In such cases, after first using VTune analyzer to exhaust tuning based on Sampling analysis, you would generally rely on Call Graph analysis for your primary tuning methodology with occasional use of sampling to make sure that new “low-hanging fruit” has not emerged.

The general usage model for Call Graph analysis starts by gathering the data and sorting the resulting results by Self Time to find the function that took the longest time. Double-click on it to make it the central node of the graph. Looking at the immediate call environment around that function is generally the next useful thing to do, so you should click on the Call List tab in the lower right corner and get the many details on all the callers and callees of that function. You can then double-click on any of the callers and callees in the Call List display and make them the central node (or focus function, as VTune analyzer calls it) and you can navigate up and down the call path for the details of the functions in the immediate vicinity of the function hotspot.

Highlighting is a feature worth exploring. On the right side of the middle icon row, you can choose to highlight certain types of functions. Highlighting Max Path to Function can help in call graphs that are complex. Sometimes it is also interesting to highlight functions with source or recursive functions. It is interesting to highlight .NET or Java methods so you can see which functions are native and which are not. This is often useful even when tuning pure managed code because there is usually unmanaged code in the run time or other library functions.

Windowing in on portions of your program is another feature worth exploring. If you right-click in the display portion of the graph and select Overview, you see the complete call graph with the currently displayed portion highlighted. You can scroll either one to navigate through the graph.

The overhead of using Call Graph analysis can be a concern for users, especially if a lot of instrumentation is needed for the whole program but you know you do not need a full program analysis. You can go to **Configure** → **Modify** and select which DLLs/SharedObjects you do or do not want to be instrumented to make the instrumented program run faster. Defaults can be set by clicking on **Advanced**, and you can even select individual functions to be instrumented or not. In this manner, you can control the amount of instrumentation and data results that are generated. This method can be particularly useful for larger applications.

Always start with the methodology discussed in the section “Learning to Fish: A Tuning Methodology” in Chapter 1. If you do not start with the big picture, you may spend unproductive time tuning to address small issues while missing the big picture.

## How Call Graph Analysis Works

Call Graph analysis works by gathering performance data for your application using a technique known as *binary instrumentation*, which is the process of injecting code into a copy of each binary module. On Microsoft Windows operating systems, these modules usually have the .exe or .dll filename extension. Instrumentation modifies a compiled program by adding data collection routines. When the modified program executes, the VTune analyzer calls these collection routines at specific execution points to dynamically record run-time performance information such as function timing and function entry and exit points. It uses this data, based on time rather than events, to determine program flow, critical functions, and call sequences.

At run time when performing Call Graph data collection, the VTune analyzer automatically instruments all Ring 3 application-level modules used by your application. As noted earlier, it cannot instrument Ring 0 kernel and driver modules. If your application dynamically loads libraries; that is, if it calls `LoadLibrary` instead of linking to stub libraries, Call Graph intercepts the module loads and automatically instrument the modules.

Call Graph places all these instrumented modules in a cache directory. When the application runs, its modules load from this directory; however, the application runs in the working directory specified by the user. Call Graph does not modify any of the original modules unless you specifically request this in the advanced configuration options.

For Java and .NET applications, Call Graph profiling uses the Java Virtual Machine Profiling Interface (JVMPID) and the .NET Profiling API, respectively, to collect performance data for managed code. By using instrumentation and the profiling APIs together, Call Graph can provide mixed-mode performance data for both Java and .NET. Mixed-mode profiling allows you to see how your managed code calls result in unmanaged code calls; however, if you are only interested in Java and .NET method calls, you can use pure mode profiling.

Call Graph uses debug and base relocation information to instrument your application's modules. It uses the debug information to locate functions in the modules. Without this information, only the exported symbols are visible and only those visible functions can be instrumented, thus limiting the Call Graph analysis. Call graph uses base relocation information to help understand the relationships between different pieces of code in a module. Under Microsoft Windows, .dll files contain base

relocation information by default, but .exe files do not unless you specifically link them with the `fixed:no` option. Under Linux, all executables contain this information unless you explicitly remove it with an option such as `--strip-unneeded`, resulting in what is commonly called a *stripped binary* or image.

Since instrumentation happens automatically, you do not need to recompile before using the Call Graph feature, although you may need to relink as noted earlier. Instrumentation does not change program functionality; however, it does slow down performance since it adds overhead. You can take steps during configuration to minimize this overhead, as you will see.