Objectives

At the completion of this module, you will be able to:

- Understand the intended purpose and usage models supported by the VTune™ Performance Analyzer.
- Identify hotspots by drilling down through various sample views.
- Understand how sampling works.
- Use callgraph profiling to find hotspots.

Agenda

- What is the VTune™ Performance Analyzer?
- Performance tuning concepts
- Using the sampling collector
- How sampling works
- Sampling Over Time
- Call Graph

VTune™ Performance Analyzer

Helps you identify and characterize performance issues by:

- Collecting performance data from the system running your application.
- Organizing and displaying the data in a variety of interactive views, from system-wide down to source code or processor instruction perspective.
- Identifying potential performance issues and suggesting improvements.
Supported Environments

Local and remote data collection
Profile applications that are running on the system that has the analyzer installed on it, or
Run profiling experiments on other systems that are running VTune analyzer remote agents on them

Local Performance Analysis

Intel® IA-32 Processors
• Microsoft Windows® operating systems
• Red Hat Linux®
• SuSE Linux
Itanium® Family Processors
• Microsoft Windows operating systems
• Red Hat Linux
• SuSE Linux
For specific operating systems versions, see the release notes

Host/Target Environment

VTune™ Performance Analyzer supports remote data collection
VTune™ Performance Analyzer installed on host system
Remote agent installed on target system

Host System
• Windows® operating system
• Controls target
• View results of data collection

Target System
• IA-32 or Itanium® processor family
• Windows or Linux®
• Intel® PXA2xx processors running Windows CE®

Feature Overview

Sampling
Call graph
**What Is a Hotspot?**

Where in an application or system there is a significant amount of activity

- **Where** = address in memory => OS process => OS thread => executable file or module => user function (requires symbols) => line of source code (requires symbols with line numbers) or processor (assembly) instruction
- **Significant** = activity that occurs infrequently probably does not have much impact on system performance
- **Activity** = time spent or other internal processor event
- Examples of other events: Cache misses, branch mispredictions, floating-point instructions retired, partial register stalls, and so on.

**Sampling: The Statistical Method of Finding Hotspots**

The sampling collector

- Periodically interrupts the processor
  - Time-based
  - Event-based: Triggered by the occurrence of a certain number of microarchitectural events
- Collects the execution context
  - Execution address in memory (CS:IP)
  - Operating system process and thread ID
  - Executable module loaded at that address
  - If you have symbols for the module, post-processing can identify the function or method at the memory address.
  - Line numbers from the symbol file can direct you to the relevant line of source code.

**Sampling Collector**

Periodically interrupt the processor to obtain the execution context

- Time-based sampling (TBS) is triggered by:
  - Operating system timer services
  - Every n processor clockticks
- Event-based sampling (EBS) is triggered by processor event counter overflow
- These events are processor-specific, like L2 cache misses, branch mispredictions, floating-point instructions retired, and so on.
Activity 1: Find the Hotspot

Learn how to identify hotspots with the VTune™ analyzer.

Red time intervals have more samples in them.

Click here for disassembly view.

Activity at Instruction Locations

Select Event

Zoom In

Zoom Out
Three Key Benefits of Sampling

You do not have to modify your code.
• But DO compile/link with symbols and line numbers.
• But DO make release builds with optimizations.
Sampling is system-wide.
• Not just YOUR application.
• You can see activity in operating system code, including drivers.
Sampling overhead is very low.
• Validity is highest when perturbation is low.
• Overhead can be reduced further by turning off progress meters in the user interface.

How else can you reduce sampling overhead?

How Event-based Sampling (EBS) Works Conceptual Diagram

Select Event Signal

Count Down

“Sample After” Number

Interrupt CPU to Take Sample

Underflow to Zero

Internal Interrupt Controller

How do you choose a “Sample After” number?

How Many Samples Are Enough?

One million samples for a five-second run?
• Do you have enough samples for it to be statistically significant?
• How much overhead are you causing?
What if you only get 100 samples?
• Is your sample after number 1?
• Are you getting a good profile?

About 1,000 samples per second is a good balance between significance and overhead

Objective: 1,000 Samples Per Second

What is the sample after value for clockticks?
• Dependent upon CPU clock speed
• ANSWER: CPU clock speed in KHz
• IF CPU clock speed = 1,400,000,000 Hz
• Sample after 1,400,000 clockticks
What is the sample after value for L2 cache read misses?
• It depends on how often you miss the L2 cache!
• Circular definition? Is not that what you are trying to determine?
• Make an intelligent guess! Estimate!
• More or less often than the clockticks?
• 10 times? 100 times? 1000 times?
**Calibration**

Sets the sample after value to get a reasonable number of samples.
- ~1000 samples per second per logical CPU

Requires the workload to be run twice

**Manual Calibration:**
- Uncheck **Calibrate Sample After** value
  - Found on Advanced Activity Configuration dialog
  - Start with default value or an estimate
  - Run a test
  - Modify the sample after value and re-test
  - Try to get about a 1000 samples per second per logical CPU

**Sampling Over Time**

Shows how sample distributions change over time by process, thread, or module

**Zoom in on time regions**

Useful for:
- Identifying time-variant performance characteristics
- Understanding thread behavior

**Sampling Over Time Usage Model**

Collect sampling data

Select items of interest from either the process, thread, or modules view

Click ☐

Highlight region of interest

Click ☐

Click ☐ to see process/thread/address histogram for time region

**Activity 2: Sampling Over Time**

Learn how to use the Sampling Over Time view
**Call Graph Profiling**

Tracks the function entry and exit points of your code at run time
Uses binary instrumentation
Uses this data to determine program flow, critical functions and call sequences
Not system-wide: Only profiles code in applications call path in Ring 3

**What Can You Profile?**

Win32 applications
Stand-alone Win32* DLLs
Stand-alone COM+ DLLs
Java applications
.NET* applications
ASP.NET applications
Linux32* applications

---

**Call Graph View**

The red lines show the critical path. The critical path is the most time-consuming call path. It is based on self time.

Bright orange nodes indicate functions with the highest self time.

**Call Graph Navigation Window**

Use the graph navigation window for an overview of the entire call graph.
Basics of VTune™ Performance Analyzer

Switch between call list and call graph views here.

Call Graph Call List View

Call Graph Metrics

<table>
<thead>
<tr>
<th>Performance Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self Time</td>
<td>Total time in a function, excluding time spent in its children (includes wait time)</td>
</tr>
<tr>
<td>Total Time</td>
<td>Time measured from a function entry to exit point</td>
</tr>
<tr>
<td>Total Wall Time</td>
<td>Time spent in a function and its children when the thread is blocked</td>
</tr>
<tr>
<td>Wall Time</td>
<td>Time spent in a function when the thread is blocked (excludes blocked time in its children)</td>
</tr>
<tr>
<td>Calls</td>
<td>Number of times the function is called</td>
</tr>
</tbody>
</table>

Activity 3: Call Graph

Find the hotspot in gzip using call graph.

Sampling Versus Call Graph

<table>
<thead>
<tr>
<th>Sampling</th>
<th>Call graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead</td>
<td>Higher overhead</td>
</tr>
<tr>
<td>System-wide</td>
<td>Ring 3 only on your application call tree</td>
</tr>
<tr>
<td>System-wide address histogram</td>
<td>Show function level hierarchy with call counts, times, and the critical path</td>
</tr>
<tr>
<td>For function level drill-down, must have debug information</td>
<td>Need re-link with /fixed:no automatically instruments</td>
</tr>
<tr>
<td>Can sample based on time and other processor events</td>
<td>Results are based on time</td>
</tr>
</tbody>
</table>

Copyright © 2006, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. *Other brands and names are the property of their respective owners.
Java* and .NET* Applications

Provides performance data for both managed code and unmanaged code
Gives insight into how managed code calls translate into Win32* calls
Uses managed code profiling API and binary instrumentation

Basics of VTune™ Performance Analyzer

What’s Been Covered
You can use the different profilers in the VTune™ analyzer to understand the different aspects of the performance of your application.

Extra Slides
Intel® Tuning Assistant

Identifies bottlenecks in:
- Pentium® 4, Pentium M®, Itanium® 2, and Pentium® III processors.
- Uses EBS and Counter Monitor data.
- Shows scaling differences between different runs.
- Code Coach is still available but is not enabled by default.
Basics of VTune™ Performance Analyzer

Intel® Tuning Assistant

VTune™ analyzer automatically selects events in the Sampling wizard.

Lab Activity 3: Getting Tuning Advice

Learn how to get processor-specific tuning advice

Windows* Command Line Interface

Collect sampling data from the command line.
Useful for integrating performance data collection into your automated regression testing.
View the data in the VTune™ Performance Analyzer or export as ASCII text.
Invoke by typing "vtl" at the command line.
Windows® Command Line Interface

- Creates hidden project structure
- To create an activity: `vtl create [activity name] + options`
- To run an activity: `vtl run [activity name]`
- To view activities type: `vtl show`
- To view results of a particular activity type: `vtl view [activityname::result] [options]`
- To delete the entire project: `vtl delete –all`
- To delete a specific activity: `vtl delete <activity name>`

Windows® Command Line Interface Examples

- Sample on clockticks and instructions retired and launch app matrix.exe:
  ```
  vtl activity a1 –c sampling –app matrix.exe run
  vtl view a1::r1 –hf –mn matrix.exe
  vtl a1::r1 view –modules
  ```

Windows® Command Line Interface Help

- For general command line arguments: `vtl –help`
- For sampling command line arguments and events:
  ```
  vtl –help –c sampling
  ```
- For in depth help and examples go to: Start->Programs->Intel® VTune™ Performance Analyzer->Help for the Command Line

Lab Activity 4: Using the Windows® Command Line Interface

- Learn how to collect sampling data from the command line
Call Graph Advanced Configuration

Set instrumentation levels.
- Helps control overhead
Select which functions are instrumented.
- Helps control overhead

Call Graph Advanced Options

This is the instrumented module status grid.
Click here to set module instrumentation levels.

Instrumentation Levels

<table>
<thead>
<tr>
<th>Instrumentation Level</th>
<th>Description</th>
<th>Debug Info Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Functions</td>
<td>Every function in the module is instrumented.</td>
<td>Yes</td>
</tr>
<tr>
<td>Custom</td>
<td>You can specify which functions are instrumented</td>
<td>Yes</td>
</tr>
<tr>
<td>Export</td>
<td>Every function in the module’s export table is instrumented.</td>
<td>No</td>
</tr>
<tr>
<td>Minimal</td>
<td>The module is instrumented but no data is collected for it.</td>
<td>No</td>
</tr>
</tbody>
</table>

More Advanced Call Graph Options

Cache directory location:
This is useful for long runs and very large applications. If you do not set this, the machine might run low on memory.

Allow call graph to instrument COM interfaces:
Function Selection

Click here to enable or disable instrumentation for a particular function.

Use Sampling and Call Graph Together

Use sampling to find which functions have hotspots. Use call graph to find out who is calling these functions.

Lab Activity 6: Using Sampling and Call Graph Together

Optimize an application (linpack) using sampling and call graph

Sampling and Call Graph Have Different Hotspots?

Self time includes blocked time.

Event-based sampling (EBS) and time-based sampling (TBS) do not include blocked time in functions (this usually appears in processor.sys).

Hotspots should be the same for self time – wait time (this is non-blocked self time).
What Counter Monitor Does

Collects hardware and software performance counter data
- Windows® Perfmon® counters
- Performance DLL SDK
Correlate counter data with sampling data

Performance DLL SDK

SDK for creating custom performance counters that can be
used by counter monitor
Available on the Intel® web site
Example: performance counter that measures the transactions
per second for a server application

Monitor Window

Click to highlight different
counter data in the graph.
To Correlate Sampling Data

Click the highlight icon and highlight a time slice by dragging over the graph from left to right.
Click on the drill icon.
You should now see the sampling data for that time slice.

Lab Activity 7: Counter Monitor

Use counter monitor to analyze gzip

Trigger API

Allows you to create your own mechanism to programmatically trigger performance counter data collection
Example: collect counter monitor data every time a frame is rendered