



Performance Tuning

VTune™ Performance Analyzer

Paul Petersen, Intel

Sept 8, 2006

Remember when
the sky was the limit?



Copyright © 2006 Intel Corporation

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

All dates provided are subject to change without notice.

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

Copyright © 2006, Intel Corporation.



Outline

Performance Tuning

- Overview
- Methodology
- Benchmarking
- Timing

VTune™

- Counter Monitor
- Call Graph
- Sampling

Summary



Resources

Getting Started with the VTune™ Performance Analyzer

- Online Help, Intel Software College

The Software Optimization Cookbook

- High-performance Recipes for the Intel® Architecture
- By Richard Gerber, Intel Press

VTune™ Performance Analyzer Essentials

- By James Reinders, Intel Press

IA-32 Intel® Architecture Software Developer's Manual

- Volume 3: System Program Guide
- Chapter 14: Debugging and Performance Monitoring



Performance Tuning

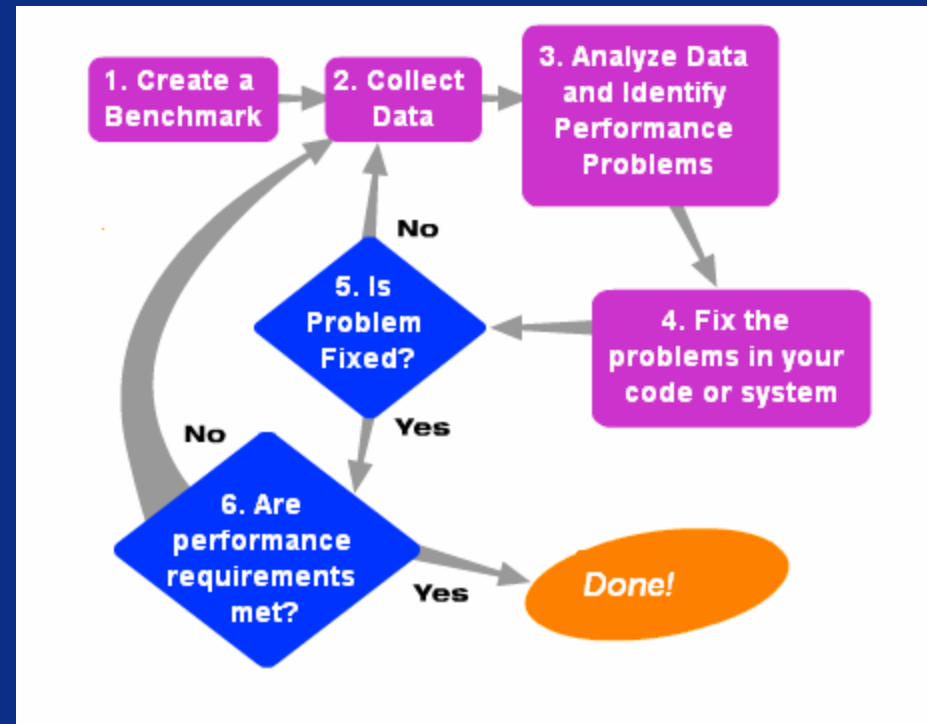
Performance tuning helps to optimize your code and make the maximum use of the latest architectures

On the right is a performance tuning methodology for analyzing and tuning application and system performance

This is the methodology generally recommended by performance analysts at Intel

The extent to which you tune your application, however, depends on your performance requirements.

You can choose to tune your application to get rid of only the major bottlenecks or you may wish to keep tuning your application until you meet a much higher level of optimization of your code.



Methodology

When you tune software it is important to follow a performance tuning methodology. It is also important to understand how the VTune analyzer fits in this methodology.

First, before you even run the VTune analyzer, you need to create a benchmark for your application. A benchmark gives you a way to measure the performance of the code you are trying to optimize.

Many new users of the VTune analyzer mistakenly use the tool before creating a good benchmark. If you don't have a good benchmark, the data you collect with the VTune analyzer will not be very useful for performance tuning.

Methodology (cont.)

Next, use this benchmark to measure the performance of your software.

- This will be your program's baseline performance.

Now you can use the VTune analyzer to identify performance bottlenecks.

Once you have identified a bottleneck, make a code change, then re-run the benchmark to see if the performance of your software has improved.

Keep repeating this process until you reach your performance goals.

The Benchmark

The benchmark is the program or process used to:

- Objectively evaluate the performance of an application
- Provide a repeatable application behavior for use with performance analysis tools
- Define a measurement metric recorded as the baseline

Attributes of a good benchmark

Repeatable

- A benchmark that produces different results each time it is run is not very useful
- If using statistical sampling, the benchmark has to run long enough to provide a statistically significant set of samples.

Representative

- The benchmark needs to cause the execution of a typical code path in the application, so that common situations are analyzed and therefore optimized

Easy to Run

- The easier it is to run, the more times it will be run by more people providing more chances to detect performance issues sooner

Verifiable

- It is frustrating to spend time optimizing a portion of the benchmark only later to find out the benchmark was defective

Timers

Use Wall Clock or CPU time?

What is the resolution of the timer?

What is the range of the timer?

What is the overhead of the timer?

Are times consistent across threads?

High-Level Timing Functions

Timer	Range & Accuracy	Code Sample
C runtime function	73 years +/- 1 sec	<pre>Time_t StartTime, ElapsedTime; StartTime = time(NULL); <... your code ...> ElapsedTime = time(NULL) - StartTime;</pre>
clock()	~49 days +/- 27ms	<pre>clock_t StartTime, ElapsedTime; StartTime = clock(); <... your code ...> ElapsedTime = clock() - StartTime;</pre>
Windows multimedia timer	~49 days +/- 1 ms	<pre>DWORD StartTime, ElapsedTime; StartTime = timeGetTime(); <... your code ...> ElapsedTime = timeGetTime() - StartTime;</pre>
OpenMP timer	~18 days omp_get_wtick()	<pre>double StartTime, ElapsedTime; StartTime = omp_get_wtime(); <... your code ...> ElapsedTime = omp_get_wtime() - StartTime;</pre>

Low-Level Timing Functions

Timer	Range & Accuracy	Code Sample
CPU clocks 32 bits	4.29 sec +/- 0.001 usec (1 Ghz processor)	<pre> DWORD StartTime, ElapsedTime; __asm { RDTSC mov StartTime, eax } <... your code ...> __asm { RDTSC sub eax, StartTime mov ElapsedTime, eax } </pre>
CPU clocks 64 bits	~580 years +/- 0.001 usec (1 Ghz processor)	<pre> __int64 StartTime, ElapsedTime, EndTime; __asm { RDTSC mov DWORD PTR StartTime, eax mov DWORD PTR StartTime+4, edx } <... your code ...> __asm { RDTSC mov DWORD PTR EndTime, eax mov DWORD PTR EndTime+4, edx } ElapsedTime = EndTime-StartTime; </pre>

Hotspots and Bottlenecks

Hotspots are locations in software that have a significant amount of activity

A Bottleneck is the location of a performance constraint in your system

The VTune Performance Analyzer helps you identify and eliminate Bottlenecks

Finding software Hotspots is one way the analyzer can help you identify Bottlenecks

VTune

The VTune™ Performance Analyzer can help you analyze the performance of your application by locating hotspots.

- Hotspots are areas in your code that take a long time to execute.

However, simply knowing where the hotspots are, is not always enough.

You must find out what is causing these hotspots and decide what type of improvements to make.

Conduct further analysis of the hotspots using the VTune analyzer.

- You can track critical function calls and monitor specific processor events, such as cache misses, triggered by sections in your code.
- You can also calculate event ratios to determine if processor events are causing the hotspots.



Data Collectors Overview

To optimize the performance of your application or system, you can do one or more of the following to find the performance bottlenecks:

- Determine how your system resources, such as memory and processor, are being utilized to identify system-level bottlenecks.
- Measure the execution time for each module and function in your application.
- Determine how the various modules running on your system affect the performance of each other.
- Identify the most time-consuming function calls and call sequences within your application.
- Determine how your application is executing at the processor level to identify microarchitecture-level performance problems.



Data Collectors - Characteristics

CHARACTERISTICS	SAMPLING	CALL GRAPH	COUNTER MONITOR
Intrusiveness	Non-intrusive	Intrusive	Non-intrusive
System-wide performance data	Provides system-wide software performance data	Only application-specific data	Provides system-wide hardware and software performance counter data
Parent and child function relationship	Does not determine parent and child function relationships	Determines parent and child function relationships and critical calls and call sequences.	Does not associate counter data with a specific application or code
Program flow	Does not determine program flow but provides a statistical analysis of the data collected on an application.	Determines program flow of an application and displays the data function calls and critical call sequences in graphs and charts	Does not determine program flow since data is not associated with any application.
System and micro-architecture-level performance data	Monitors processor events, such as Cache Misses, to help identify microarchitecture-level performance problems associated with specific sections of your code.	Does not monitor microarchitecture-level performance problems.	Monitors hardware and software performance counters over a specified duration to help identify system and microarchitecture-level performance data.

System-Level Tuning

Counter Monitor

- Identify OS issues via Counter Monitor
 - Similar to Perfmon (Windows)
 - Administrative Tools\Performance

Can use time correlation with Sampling

- Multiple runs of the same benchmark

Application-Level Tuning

Call Graph

- Examine the flow of control through the application using Call Graph
 - Which functions took the longest
 - Which functions were blocked the longest
 - Calling sequence critical path

Algorithmic changes are best driven from Call Graph analysis.

- This is typical the source of the big improvements

Architecture-Level Tuning

Sampling is a technique for statistically profiling software.

- Sampling periodically interrupts the system under observation and records its execution context.

The VTune analyzer uses two different methods to generate sampling interrupts:

- time based sampling (TBS)
- event based sampling (EBS).

The sampling period for TBS is based on a user defined time interval

- usually 1 ms.

The sampling period for EBS is determined by the occurrence of micro-architectural events.

- Number of events needed to trigger an interrupt is called the Sample After Value.
- This value is user definable or can be automatically determined by the VTune analyzer by using calibration.
- For example, if you select the event, 2nd Level Cache Load Misses Retired, with a Sample After Value of 5000, the VTune analyzer will sample the execution context of your system every 5000 loads that miss the L2 cache occur.

Collecting Samples

Every time a sampling interrupt occurs, the VTune analyzer collects:

- program counter,
- process ID,
- thread ID,
- CPU number,
- processor time stamp.

Using this information, the VTune analyzer can bin the samples by process, thread, module, function, or line of code. Each of these items can also be broken down by CPU.

Collecting Samples (cont.)

Before sampling begins, the VTune analyzer creates a memory map for every process on the system.

- Notified every time a new process is created or a process loads a new module while it is collecting sampling data.
- Allows the VTune analyzer to associate samples with modules.

Once the module in which the sample occurred is determined, the VTune analyzer uses the module's debug information to identify the function or line of source where the sample occurred.

- This happens after data collection to minimize the performance overhead of the profiler.

Performance Counters

The Performance Monitoring Unit (PMU) on Intel® processors are architected with counters that can be configured and monitored to measure processor events.

- The performance counter value is set up to interrupt the processor after a predetermined number of events have been observed.
- The Pentium® 4 processor is equipped with 18 such counters.

However, this does not mean that 18 events can be monitored at the same time.

- There are hardware restrictions based on MSR resources that allow only a specific number of events to be monitored at a time.
- Use the VTune™ Performance Analyzer to tweak Processor Event Address Registers (EARs) to enable a counter to record the performance of a specific processor event.

Analysis of the performance counters provides insight into the details that can help in improving software performance.

- Based on the event data, and advice from the Tuning Assistant, you can identify specific areas in your code that need to be changed in order to fix the problem.

Clockticks (Non-sleep)

The clockticks event, also referred to as non-sleep clocktick, counts the basic unit of time recognized by each physical processor package in the Intel NetBurst® microarchitecture family of processors.

- Non-sleep Clockticks are used to indicate the time spent by each physical processor package to execute a stream of instructions.

The raw data collected by the VTune™ Performance Analyzer can be used to compute various indicators.

- For example, ratios of the Non-sleep Clockticks to Instructions Retired can give you a good indication that the code in the application may be stalled in a physical processor package and may be suitable for re-coding.

Non-sleep clockticks can be used to calculate the non-sleep CPI (Cycles Per Instruction).

- This can be interpreted as the inverse of the average throughput of a physical processor package, while the physical processor package is executing instructions.

If a physical processor package contains only one logical processor, non-sleep clockticks is essentially the same as non-halted clockticks.

Instructions Retired

This event counts the number of instructions retired.

- By default, this count does not include bogus instructions. In this event description, the term bogus refers to instructions that were canceled because they are on a path that should not have been taken due to a mispredicted branch.
- Branch mispredictions incur a large penalty on deeply pipelined microprocessors. In general, branches can be predicted accurately. However in the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.

This event is used in all per instruction ratios.

MOB Loads Replays Retired

(Blocked Store-to-Load Forwards Retired)

This event counts the number of retired load instructions that experienced memory order buffer (MOB) replays because store-to-load forwarding restrictions were not observed.

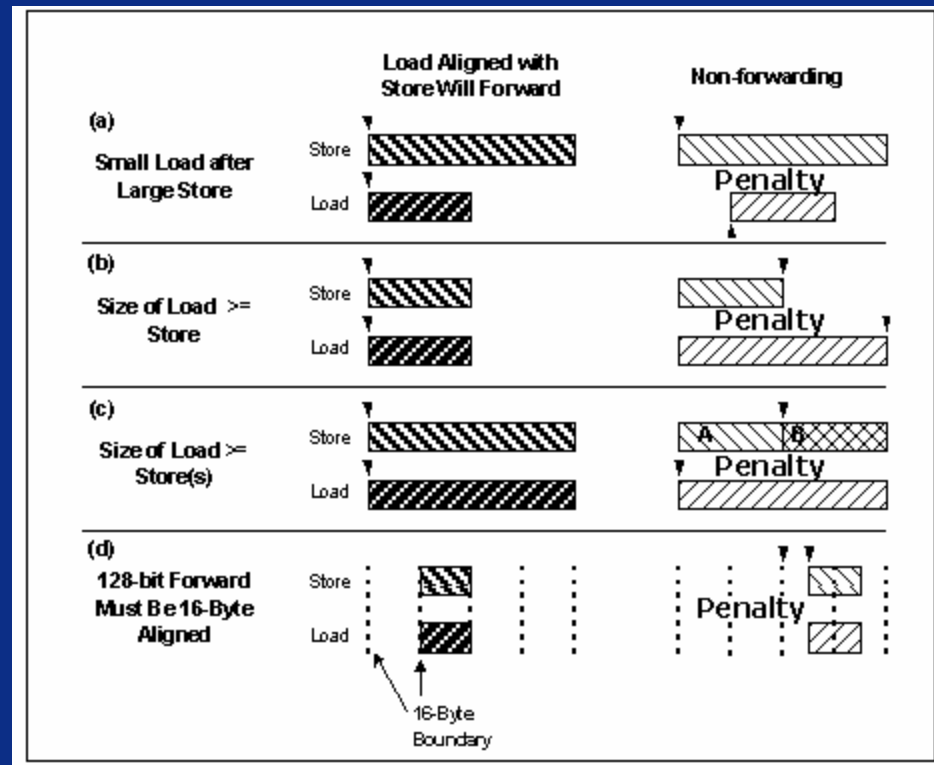
- A MOB replay may occur for several reasons. This event is programmed to count those MOB replays caused by loads in which store-to-load forwarding is blocked.

Intel® Pentium® 4 processors use a store-to-load forwarding technique to enable certain memory load operations (loads from an address whose data has just been modified by a preceding store operation) to complete without waiting for the data to be written to the cache. There are size and alignment restrictions for store-to-load forwarding cases to succeed.



MOB Load Replays Retired

(Blocked Store-to-Load Forwards Retired)



Split Loads Retired

This event counts the number of retired instructions that caused split loads.

- A split load occurs when a data value is read, and part of the data is located in one cache line and part in another.
- Split loads reduce performance because they force the processor to read two cache lines separately and then paste the two parts of data back together.
- Reading data from two cache lines is several times slower than reading data from a single cache line even if the data is not otherwise properly aligned.

On the Pentium® 4 processor, each first-level data-cache line contains 64 bytes of data, so the address of the data at the beginning of each line is a multiple of 64.

Mispredicted Branches Retired

This event counts the number of retired branch instructions that were mispredicted by the processor.

- This means that the processor predicted that the branch would be taken, but it was not, or vice-versa.
- Mispredicted branches cause reduced performance in your application because the processor starts executing instructions along the path it predicts.
- When the misprediction is discovered, all the work done on those instructions must be discarded, and the processor must start again on the right path.

To determine the branch misprediction ratio, divide the number of these events by the number of Branches Retired events.

To determine the number of mispredicted branches per instruction, divide the number of these events by the number of Instructions Retired events.

Summary

The Performance Tuning Methodology is simple and obvious

- Define your benchmarks
- Calculate the baseline performance
- Analyze the application to find hotspots and bottlenecks
- Fix the bottleneck
- Record the new performance
- Repeat until desired performance improvement is achieved

Use the performance tools as appropriate

- Counter monitor to see OS issues
 - Spot system issues early (page faults, context switches, ...)
- Call Graph to see code relationships
 - Algorithmic changes have largest impact
- Sampling to detect hotspots
 - Clockticks is most general event, it shows effects of all problems

