INTEL® VTUNE™ AMPLIFIER

PERFORMANCE PROFILER

Kevin O'Leary – Intel Developer Products Division
Tune Applications for Scalable Multicore Performance

Agenda

- Introduction
- Data Collection – Rich set of performance data
- Data Analysis - Find answers fast
- Flexible workflow – User i/f and command line
- Performance Analysis Details
- Summary
Faster, Scalable Code, Faster

Intel® VTune™ Amplifier Performance Profiler

Accurate Data - Low Overhead
- CPU, GPU, FPU, threading, bandwidth...

Meaningful Analysis
- Threading, OpenMP region efficiency
- Memory access, storage device

Easy
- Data displayed on the source code
- Easy set-up, no special compiles

“Last week, Intel® VTune™ Amplifier helped us find almost 3X performance improvement. This week it helped us improve the performance another 3X.”

Claire Cates
Principal Developer
SAS Institute Inc.
Setting up a profile is easy

1. What/where to profile
2. Choose Analysis Type
3. Collection options
4. Push Start

**Hotspots**
- Want to find out where your app spends time and optimize your algorithms?
  - Basic Hotspots
  - Advanced Hotspots
  - Memory Consumption

**Microarchitecture**
- Want to see how efficiently your code is using the underlying hardware?
  - General Exploration
  - Memory Access

**Parallelism**
- Want to assess the compute efficiency of your multi-threaded app?
  - Concurrency
  - Locks and Waits

**Memory Access**
- Measure a set of metrics to identify memory access related issues (for example, specific for NUMA architectures). This analysis type is based on the hardware event-based sampling collection. Learn more (F1)
  - CPU sampling interval, ms: 1
  - Analyze dynamic memory objects
  - Minimal dynamic memory object size to track, in bytes: 1024
  - Evaluate max DRAM bandwidth
  - Analyze OpenMP regions
  - Collect I/O API data:
    - No
    - Collect I/O waits
    - Collect stacks
    - Stack size, in bytes
  - Stack type

*Full command-line also available*
Full Visual Studio* Integration
# Two Great Ways to Collect Data

## Intel® VTune™ Amplifier

<table>
<thead>
<tr>
<th>Software Collector</th>
<th>Hardware Collector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses OS interrupts</td>
<td>Uses the on chip Performance Monitoring Unit (PMU)</td>
</tr>
<tr>
<td>Collects from a single process tree</td>
<td>Collect system wide or from a single process tree.</td>
</tr>
<tr>
<td>~10ms default resolution</td>
<td>~1ms default resolution (finer granularity - finds small functions)</td>
</tr>
<tr>
<td>Either an Intel® or a compatible processor</td>
<td>Requires a genuine Intel® processor for collection</td>
</tr>
<tr>
<td>Call stacks show calling sequence</td>
<td>Optionally collect call stacks</td>
</tr>
<tr>
<td>Works in virtual environments</td>
<td>Works in a VM only when supported by the VM (e.g., vSphere®, KVM)</td>
</tr>
<tr>
<td>No driver required</td>
<td>Requires a driver</td>
</tr>
<tr>
<td></td>
<td>- Easy to install on Windows</td>
</tr>
<tr>
<td></td>
<td>- Linux requires root (or use default perf driver)</td>
</tr>
</tbody>
</table>

## No special recompiles - C, C++, C#, Fortran, Java, Assembly
Example: Hotspots Analysis

Summary View

**Elapsed Time**: 5.554s
- CPU Time: 10.504s
  - Instructions Retired: 21,698,000,000
  - CPI Rate: 1.257
  - CPU Frequency Rate: 1.041
  - Total Thread Count: 9
  - Paused Time: 0s

**Top Hotspots**
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid_intersect</td>
<td>3_techyom_omp.exe</td>
<td>5.530s</td>
</tr>
<tr>
<td>sphere_intersect</td>
<td>3_techyom_omp.exe</td>
<td>3.247s</td>
</tr>
<tr>
<td>func@0x1002e50d</td>
<td>libomp5md.dll</td>
<td>0.148s</td>
</tr>
<tr>
<td>shader</td>
<td>3_techyom_omp.exe</td>
<td>0.117s</td>
</tr>
<tr>
<td>KeDelayExecutionThread</td>
<td>ntdoskml.exe</td>
<td>0.091s</td>
</tr>
<tr>
<td>[Others]</td>
<td>N/A*</td>
<td>1.361s</td>
</tr>
</tbody>
</table>

* N/A is applied to non-countable metrics.

**Average Bandwidth**

<table>
<thead>
<tr>
<th>Package</th>
<th>Total, GB/sec</th>
<th>Read, GB/sec</th>
<th>Write, GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>5715</td>
<td>3504</td>
<td>2212</td>
</tr>
</tbody>
</table>

**CPU Usage Histogram**
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

**Collection and Platform Info**
This section provides information about this collection, including result set size and collection platform data.
Example: Threading Analysis
Bottom-up View
Example: Memory Access Analysis
Bottom-up View

- Over-Time DRAM Bandwidth
- Over-Time QPI/UPI Bandwidth
- Grid Breakdown by Function (configurable)
Find Answers Fast

Intel® VTune™ Amplifier

Adjust Data Grouping
- Function - Call Stack
- Module - Function - Cell Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack
... (Partial list shown)

Double Click Function to View Source

Click > for Call Stack

Filter by Timeline Selection (or by Grid Selection)

Filter by Process & Other Controls
Tuning Opportunities Shown in Pink. Hover for Tips
See Profile Data On Source / Asm
Double Click from Grid or Timeline

View Source / Asm or both  CPU Time  Right click for instruction reference manual

Quick Asm navigation:
Select source to highlight Asm

Scroll Bar “Heat Map” is an overview of hot spots  Click jump to scroll Asm
Timeline Visualizes Thread Behavior

Intel® VTune™ Amplifier

Optional: Use API to mark frames and user tasks
Optional: Add a mark during collection
Tune OpenMP for Efficiency and Scalability

Fast Answers: Is My OpenMP Scalable? How Much Faster Could It Be?

1) Is the serial time of my application significant enough to prevent scaling?
2) How much performance can be gained by tuning OpenMP?
3) Which OpenMP regions / loops / barriers will benefit most from tuning?
4) What are the inefficiencies with each region? (click the link to see details)
Command Line Interface

Automate analysis

amplxe-cl is the command line:

- **Windows:** C:\Program Files (x86)\IntelSWTools\VTune Amplifier\bin[32|64]\amplxe-cl.exe
  - **Linux:** /opt/intel/vtune_amplifier/bin[32|64]/amplxe-cl

Help: amplxe-cl -help

Use UI to setup
1) Configure analysis in UI
2) Press “Command Line...” button
3) Copy & paste command

Great for regression analysis – send results file to developer
Command line results can also be opened in the UI
Compare Results Quickly - Sort By Difference

Intel® VTune™ Amplifier

Quickly identify cause of regressions.

- Run a command line analysis daily
- Identify the function responsible so you know who to alert

Compare 2 optimizations – What improved?

Compare 2 systems – What didn’t speed up as much?
Optimize Memory Access
Memory Access Analysis - Intel® VTune™ Amplifier 2017

Tune data structures for performance
- Attribute cache misses to data structures (not just the code causing the miss)
- Support for custom memory allocators

Optimize NUMA latency & scalability
- True & false sharing optimization
- Auto detect max system bandwidth
- Easier tuning of inter-socket bandwidth

Easier install, Latest processors
- No special drivers required on Linux*
- Intel® Xeon Phi™ processor MCDRAM (high bandwidth memory) analysis
Memory Object Identification

Microarchitecture Analysis
- General Exploration
- Memory Access
- TSX Exploration
- TSX Hotspots

View allocated objects

Sort by LLC Miss Count

Grouping: Memory Object / Function / Call Stack

<table>
<thead>
<tr>
<th>Memory Object / Function / Call Stack</th>
<th>CPU Time</th>
<th>L1 Cache Miss Count</th>
<th>L2 Cache Miss Count</th>
<th>L3 Cache Miss Count</th>
<th>L4 Cache Miss Count</th>
<th>Data Cache Miss Count</th>
<th>Average Latency (cycles)</th>
<th>Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>memTest.out!main (2 MB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memTest.cpp:10 (4 KB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memTest.cpp:20 (4 KB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memTest.cpp:11 (4 KB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memTest.cpp:21 (4 KB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memTest.cpp:25 (4 KB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memTest.cpp:18 (4 KB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Memory Object Identification

---

Assembly view also available

Double-click to see allocation site in source view

---
Storage Device Analysis (HDD, SATA or NVMe SSD)
Intel® VTune™ Amplifier

Are You I/O Bound or CPU Bound?
- Explore imbalance between I/O operations (async & sync) and compute
- Storage accesses mapped to the source code
- See when CPU is waiting for I/O
- Measure bus bandwidth to storage

Latency analysis
- Tune storage accesses with latency histogram
- Distribution of I/O over multiple devices
A Quick Question for the Audience
TANGENT ON A COUPLE OTHER TOOLS
INTEL® ADVISOR 2019
VECTORIZATION OPTIMIZATION AND THREAD PROTOTYPING

• Vectorization Advisor
• Threading Advisor
• Flow Graph Analyzer
Get Faster Code Faster! Intel® Advisor
Thread Prototyping

Have you:
- Threaded an app, but seen little benefit?
- Hit a “scalability barrier”?
- Delayed release due to sync. errors?

Data Driven Threading Design:
- Quickly prototype multiple options
- Project scaling on larger systems
- Find synchronization errors before implementing threading
- Design without disrupting development

Add Parallelism with Less Effort, Less Risk and More Impact

“Intel® Advisor has allowed us to quickly prototype ideas for parallelism, saving developer time and effort”
Simon Hammond
Senior Technical Staff
Sandia National Laboratories

http://intel.ly/advisor-xe
Get Faster Code Faster! Intel® Advisor
Vectorization Optimization

Have you:
- Recompiled for AVX2 with little gain
- Wondered where to vectorize?
- Recoded intrinsics for new arch.?
- Struggled with compiler reports?

Data Driven Vectorization:
- What vectorization will pay off most?
- What’s blocking vectorization? Why?
- Are my loops vector friendly?
- Will reorganizing data increase performance?
- Is it safe to just use pragma simd?

"Intel® Advisor’s Vectorization Advisor permitted me to focus my work where it really mattered. When you have only a limited amount of time to spend on optimization, it is invaluable."

Gilles Civario
Senior Software Architect
Irish Centre for High-End Computing
What is a Roofline Chart?

A Roofline Chart plots application performance against hardware limitations.

- Where are the bottlenecks?
- How much performance is being left on the table?
- Which bottlenecks can be addressed, and which should be addressed?
- What's the most likely cause?
- What are the next steps?

Roofline first proposed by University of California at Berkeley: 
[Roofline: An Insightful Visual Performance Model for Multicore Architectures](#), 2009

Cache-aware variant proposed by University of Lisbon: 
[Cache-Aware Roofline Model: Upgrading the Loft](#), 2013
INTEL® INSPECTOR 2019
MEMORY & THREAD DEBUGGER
Debug Memory & Threading with Intel® Inspector
Find & Debug Memory Leaks, Corruption, Data Races, Deadlocks

Correctness Tools Increase ROI by 12%-21%¹
- Errors found earlier are less expensive to fix
- Races & deadlocks not easily reproduced
- Memory errors are hard to find without a tool

Debugger Integration Speeds Diagnosis
- Breakpoint set just before the problem
- Examine variables and threads with the debugger

What's New in 2019 Release
Find Persistent Memory Errors
- Missing / redundant cache flushes
- Missing store fences
- Out-of-order persistent memory stores
- PMDK transaction redo logging errors


Debug Memory & Threading Errors

Intel® Inspector

Find and eliminate errors
- Memory leaks, invalid access...
- Races & deadlocks
- C, C++, and Fortran (or a mix)

Simple, Reliable, Accurate
- No special recompiles
  Use any build, any compiler
- Analyzes dynamically generated or linked code
- Inspects 3rd party libraries without source
- Productive user interface + debugger integration
- Command line for automated regression analysis

Clicking an error instantly displays source code snippets and the call stack

Fits your existing process
DIVING DEEPER INTO ANALYSIS
Introduction to Performance Tuning

System
- H/W tuning:
  - BIOS (TB, HT)
  - Memory
  - Network I/O
  - Disk I/O
- OS tuning:
  - Page size
  - Swap file
  - RAM Disk
  - Power settings
  - Network protocols

Application
- Better application design:
  - Parallelization
  - Fast algorithms / data bases
  - Programming language and RT libs
  - Performance libraries
  - Driver tuning

Processor
- Tuning for Microarchitecture:
  - Compiler settings/Vectorization
  - Memory/Cache usage
  - CPU pitfalls

Think performance wise (app/sys level)
Choose performance. effective solutions
Apply performance optimization and check results
Add performance regressions to test stage
Collect and analyze performance related issues from users
System-Level Profiling – High-level Overviews

- General Exploration viewpoint (change)
- Collection Log
- Analysis Target
- Analysis Type
- Summary
- Bottom-up

- Elapsed Time: 6.306s
  - Clockticks: 30,869,300,000
  - Instructions Retired: 25,745,000,000
  - CPI Rate: 1.199
  - MUX Reliability: 0.972
  - Front-End Bound: 7.2% of Pipeline Slots
  - Bad Speculation: 6.0% of Pipeline Slots
  - Branch Mispredict: 5.9% of Pipeline Slots
  - Machine Clears: 0.1% of Pipeline Slots
  - Back-End Bound: 64.1%
    - Memory Bound: 33.3%
    - Core Bound: 38.0%
    - Divider: 0.0%
    - Port Utilization: 29.6%
    - Rating: 22.7%
  - Total Thread Count: 9
  - Paused Time: 0s

- CPU Usage Histogram
  - This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU usage value.
System-Level Profiling – Process/Module Breakdowns

- Processes
- Modules
- Functions
System-Level Profiling – Disk I/O Analysis

Are You I/O Bound or CPU Bound?
- Explore imbalance between I/O opera (async & sync) and compute
- Storage accesses mapped to the source code

See when CPU is waiting for I/O
- Measure bus bandwidth to storage
- Latency analysis
- Tune storage accesses with latency histogram
- Distribution of I/O over multiple devices

> amplxe-cl -collect disk-io -d 10
System-Level Profiling – HPC Characterization

Three Metric Classes
• CPU Utilization
  • Logical core % usage
  • Includes parallelism and OpenMP information
• Memory Bound
  • Break down each level of the memory hierarchy
• FPU Utilization
  • Floating point GFLOPS and density

> amplxe-cl -collect hpc-performance -d 10

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
System-Level Profiling – Memory Bandwidth

Find areas of high and low bandwidth usage. Compare to max system bandwidth based on Stream benchmarks.

-knob collect-memory-bandwidth=true
Application Performance Tuning Process

1. Find Hotspots
2. Determine Efficiency
3. Address Parallelism Issues
4. Address Hardware Issues
5. Rebuild and Compare Results

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Find Hotspots

> amplxe-cl -collect hotspots -- ./myapp.out
Find Hotspots

- Drill to source or assembly
- Hottest areas easy to ID
- Is this the expected behavior
- Pay special attention to loops and memory accesses

- Learn how your code behaves
- What did the compiler generate
- What are the expensive statements
Determine Efficiency

Look for Parallelism, Cycles-per-Instruction (CPI), and Retiring %
Address Parallelism Issues

- Use Concurrency Analysis to ensure you’re using all your threads as often as possible.
- Common concurrency problems can often be diagnosed in the timeline.
- Switch to the Locks And Waits viewpoint or run a Locks and Waits analysis to investigate contention.
Address Hardware Issues

The X86 Processor Pipeline (simplified)
Pipeline Slot Categorization

- Pipeline slots can be sorted into one of four categories on a given cycle by what happens to the uop in that slot.
  - Retiring
  - Bad Speculation
  - Back End Bound
  - Front End Bound

- Each category has an expected range of values in a well tuned application.

### Table: Pipeline Slot Categorization

<table>
<thead>
<tr>
<th>Category</th>
<th>App. Type:</th>
<th>Client/Desktop</th>
<th>Server/Database/Distributed</th>
<th>High Performance Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retiring</td>
<td>▲</td>
<td>20-50%</td>
<td>10-30%</td>
<td>30-70%</td>
</tr>
<tr>
<td>Bad Speculation</td>
<td>▼</td>
<td>5-10%</td>
<td>5-10%</td>
<td>1-5%</td>
</tr>
<tr>
<td>Front End Bound</td>
<td>▼</td>
<td>5-10%</td>
<td>10-25%</td>
<td>5-10%</td>
</tr>
<tr>
<td>Back End Bound</td>
<td>▼</td>
<td>20-40%</td>
<td>20-60%</td>
<td>20-40%</td>
</tr>
</tbody>
</table>
The uop Pipeline
Categorizing the hotspots

• Modern CPUs “pipeline” instructions. This pipeline can be generally divided into two sections.
  • The Front End fetches instructions, decodes them into uops, and allocates them to...
  • The Back End, which is responsible for executing the uops. Once successfully completed, a uop is considered “retired”.

• A Pipeline Slot is an abstract representation of the hardware resources needed to process a uop.

• The front end can only allocate so many uops per cycle, and the same is true of the back end and retiring them. This determines the number of Pipeline Slots. As a general rule, this number is four.
Pipeline Slot Categorization

Retiring

This is the good category! You want as many of your slots in this category as possible. However, even here there may be room for optimization.
Pipeline Slot Categorization

Bad Speculation

This occurs when a uop is removed from the back end without retiring; effectively, it’s cancelled, most often because a branch was mispredicted.
Pipeline Slot Categorization

Back End Bound

This is when the back end can't accept uops, even if the front end can send them, because it already contains uops waiting on data or long execution.
Pipeline Slot Categorization

Front End Bound

This is when the front end can't deliver uops even though the back end can take them, usually due to delays in fetching code or decoding instructions.
Identifying and Diagnosing Inefficiency

Microarchitecture Analysis

- Microarchitecture Exploration (previously General Exploration) is a hardware events analysis. It is preconfigured to sample the appropriate events on your architecture and calculates the proper metrics from them.

- Potential tuning opportunities are highlighted in pink.

- To check the efficiency of a hotspot, look at the Retiring metric. If it’s less than the expected number for your application type, it’s probably inefficient.

  - Hotspots with high retiring values may still have room for improvement.

<table>
<thead>
<tr>
<th>App Type</th>
<th>Expected</th>
<th>Instructions Retired</th>
<th>CPI Rate</th>
<th>Front-End Bound</th>
<th>Bad Speculation</th>
<th>Back-End Bound</th>
<th>Retiring</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize_2D_buffer</td>
<td></td>
<td>85,219,200,000</td>
<td>0.266</td>
<td>0.5%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>grid_intersect</td>
<td></td>
<td>10,963,200,000</td>
<td>0.706</td>
<td>4.6%</td>
<td>15.1%</td>
<td>46.4%</td>
<td>33.9%</td>
</tr>
<tr>
<td>sphere_intersect</td>
<td></td>
<td>10,946,400,000</td>
<td>0.601</td>
<td>2.1%</td>
<td>1.6%</td>
<td>47.5%</td>
<td>48.8%</td>
</tr>
<tr>
<td>grid_bounds_intersect</td>
<td></td>
<td>480,000,000</td>
<td>1.105</td>
<td>13.0%</td>
<td>2.2%</td>
<td>52.3%</td>
<td>32.5%</td>
</tr>
<tr>
<td>tri_intersect</td>
<td></td>
<td>216,000,000</td>
<td>0.789</td>
<td>0.0%</td>
<td>20.2%</td>
<td>39.3%</td>
<td>40.5%</td>
</tr>
</tbody>
</table>

> amplxe-cl -collect uarch-exploration -- ./myapp.out
Categorizing and Correcting Inefficiencies

Microarchitecture Exploration Analysis

- Intel® VTune™ Amplifier has hierarchical expanding metrics categorized by the four slot types.
- You can expand your way down, following the hotspot, to identify the root cause of the inefficiency.
  - Sub-metrics highlight pink on their own merits, just like top level metrics.
- Hovering over a metric produces a helpful, detailed tooltip (not shown).
  - There are tooltips on Summary tabs too: hover over any 📈 icon.
Categorizing and Correcting Inefficiencies

Retiring: *Microarchitecture Exploration Analysis, Intel® Advisor*

- High Retiring percentage is generally good, but may be inefficient if you’re doing work that doesn’t need to be done at all, or could be done faster.

- Retiring can be split based on whether the uops being retired came from the microcode sequencer or not.

  - **Yes?** Try reworking code to avoid microcode assists.
  
  - **No?** Make sure the code is well vectorized.

**Tip:**

Use Vectorization Advisor to fine-tune your vectorization.
HPC Characterization: FPU Utilization

FPU utilization

% of FPU load (100% - FPU is fully loaded, threshold 50%)

Calculation based on PMU events representing scalar and packed single and double precision SIMD instructions

Metrics in FPU utilization section

FLOPs broken down by scalar and packed

Instruction Mix

Top 5 loops/functions by FPU usage

- Detected with static binary analysis

Vectorized vs. Non-vectorized, ISA, and characterization detected by static analysis and Intel Compiler diagnostics

HARDWARE IS BECOMING MORE VECTORIZED, SO SHOULD YOU!
Categorizing and Correcting Inefficiencies

Bad Speculation: Microarchitecture Exploration Analysis

• Bad Speculation is caused by either Machine Clears or Branch Mispredicts.
  • **Machine Clears** can be caused by self-modifying code, etc.
  • **Branch mispredicts** are more common. These occur when the paths taken by `if`, `switch`, `for`, `do-while`, and other conditional branches are incorrectly predicted and the uops have to be thrown out.

• Use Intel® VTune™ Amplifier’s Source Viewer to identify problematic branches.

• Avoid unnecessary branching:
  • Remove branches entirely if possible
  • Move branches outside of loops if possible.
Categorizing and Correcting Inefficiencies
Front End: *Microarchitecture Exploration Analysis*

- Front End Bound pipeline slots are common in JIT or interpreted code.
- Front End Bound can be bandwidth or latency:
  - **Bandwidth** issues are caused by inefficient instruction decoding, or restrictions in caching decoded instructions, etc.
  - **Latency** is caused by instruction cache misses, delays in instruction fetching after branch mispredicts, switching to the microcode sequencer too often, etc.
Categorizing and Correcting Inefficiencies

Back End: *Microarchitecture Exploration Analysis, Memory Bandwidth*

- Back End bound is the most common bottleneck type for most applications.
- It can be split into Core Bound and Memory Bound
  - **Core Bound** includes issues like not using execution units effectively and performing too many divides.
  - **Memory Bound** involves cache misses, inefficient memory accesses, etc.
    - Store Bound is when load-store dependencies are slowing things down.
    - The other sub-categories involve caching issues and the like. Memory Access Analysis may provide additional information for resolving this performance bottleneck.
Rebuild and Compare Results

Elapsed Time: 7.420s - 5.541s = 1.879s

Instructions Retired: 24,654,400,000 - 22,868,400,000 = 1,786,000,000
CPI Rate: 1.326 - 1.363 = -0.037
CPU Frequency Ratio: 1.040 - 1.042 = -0.003
Total Thread Count: Not changed, 4
Paused Time: Not changed, 0s
CPU Time: 12.603s - 11.967s = 0.636s

CPU Usage Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running.
Example: Poor NUMA Utilization

If Memory Bound is high and local caches are not the problem, focus on "Remote" metrics.
Example: Poor NUMA Utilization

Look for areas of high QPI/UPI bandwidth

QPI/UPI BANDWIDTH IS COMMUNICATION BETWEEN THE SOCKETS. THIS MAY INDICATE SOME SORT OF NUMA ISSUE.
EXAMPLE: POOR NUMA UTILIZATION

Common causes of poor NUMA utilization

- Allocation vs. first touch memory location
- False sharing of cache lines
  - Use padding when necessary
- Arbitrary array accesses
- Poor thread affinity

WHERE IS YOUR MEMORY ALLOCATED AND WHERE ARE YOUR THREADS RUNNING?
Tuning Guides Available Online


### Intel® VTune™ Amplifier Tuning Guides

Our tuning guides explain how to identify common software performance issues using VTune Amplifier and give suggestions for optimization.

<table>
<thead>
<tr>
<th>Microarchitecture Code Name</th>
<th>Processors Covered</th>
<th>Tuning Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo Lake</td>
<td>Intel Atom® Processor E3800 Series, and Intel® Pentium® and Celeron® Processor N and J Series</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Skylake-X</td>
<td>Intel® Xeon Processor Scalable Family 1st Gen</td>
<td>Download New PDF, Download Old PDF</td>
</tr>
<tr>
<td>Knights Landing</td>
<td>Intel® Xeon Phi™ Processor</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Broadwell-E® (Server)</td>
<td>Intel® Xeon Processor E5-2600 v4 Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Skylake</td>
<td>6th Generation Intel® Core™ Processor Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Broadwell</td>
<td>5th Generation Intel® Core™ Processor Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Haswell-E® (Server)</td>
<td>Intel® Xeon® Processor E5 v3 Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Ivy Bridge-E® (Server)</td>
<td>Intel® Xeon® Processor E5-2600 v3 Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Haswell</td>
<td>4th Generation Intel® Core™ Processor Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Sandy Bridge-E®/EX®/EX®/EN (Server)</td>
<td>Intel® Xeon® Processor E5 Family</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>3rd Generation Intel® Core™ Processor Families</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>2nd Generation Intel® Core™ Processor Families</td>
<td>Download PDF</td>
</tr>
<tr>
<td>Mary Integrated Core Architecture</td>
<td>Intel® Xeon Phi™ coprocessor</td>
<td>Read the Article</td>
</tr>
</tbody>
</table>
Example 1 – Matrix Multiply

Bound by the L3 Cache while reading arrays
Example 1 – Matrix Multiply

Interchange loop indices and collapse loops

```c
void multiply3(int msize, int tidx, int numt, TYPE a[][],
    int i,j,k;

    #pragma omp parallel for collapse (2)
    for(i=0; i<msize; i++) {
        for(k=0; k<msize; k++) {
            #pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
```
Example 1 – Matrix Multiply

Elapsed Time: 6.241s
- Clockticks: 71,522,000,000
- CPI: 0.923

General Exploration viewpoint (change)
- Elapsed Time: 1.684s
- Clockticks: 18,512,000,000
- CPI: 0.307
- MUX Reliability: 0.007
- Front-End Bound: 6.3% of Pipeline Slots
- Bad Speculation: 0.5% of Pipeline Slots
- Back-End Bound: 26.6% of Pipeline Slots
- Memory Bound: 26.4% of Pipeline Slots
- Core Bound: 11.5% of Pipeline Slots
- Divider: 0.0% of Clockticks
- Port Utilization: 6.7% of Clockticks
- Cycles of 0 Ports Utilized: 0.0% of Clockticks
- Cycles of 1 Port Utilized: 3.1% of Clockticks
- Cycles of 2 Ports Utilized: 3.6% of Clockticks
- Cycles of 3+ Ports Utilized: 79.3% of Clockticks
- Retiring: 66.0% of Pipeline Slots
- Total Thread Count: 4
- Paused Time: 0s
Example 2 – Calculating Prime Numbers

```c
41 int _tmain(int argc, _TCHAR* argv[])
42 {
43     DWORD msBegin = timeGetTime();
44
45     #pragma omp parallel for
46     for(int p = 3; p <= limit; p += 2) {
47         if (IsPrime(p)) Tick();
48     } 
49     DWORD msDuration = timeGetTime() - msBegin;
50
51     printf("MS: %d\n", msDuration);
52     printf("primes = %d\n", primes);
53     return primes != correctCount;
54 }
```

- **OpenMP uses Static Scheduling**
- **Load Imbalance**
Example 2 – Calculating Prime Numbers

```c
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD msBegin = timeGetTime();
    #pragma omp parallel for schedule (dynamic, 1000)
    for(int p = 3; p <= limit; p += 2) {
        if (IsPrime(p)) Tick();
    }
    DWORD msDuration = timeGetTime() - msBegin;
    printf("MS: %d\n", msDuration);
    printf("primes = %d\n", primes);
    return primes != correctCount;
}
```
Summary: Top Down Tuning Method

- Make system-level optimizations
- Make algorithmic optimizations
  - Use Threading Advisor to add threading
  - Use Concurrency Analysis and Locks & Waits Analysis to tune threading
- Make microarchitectural optimizations
  - Find your hotspots
    - Use Hotspots Analysis or Advanced Hotspots Analysis
  - For each hotspot, determine efficiency.
    - Use General Exploration Analysis to identify inefficient hotspots.
  - If inefficient: Categorize the bottleneck, identify the cause, and optimize it!
    - Hierarchical metrics in General Exploration Analysis focus your attention where it's needed most and allow you to easily identify the issue.
    - Memory Access Analysis can help with Back End Bound code.
    - Vectorization Advisor can help improve the efficiency of Retiring code.
Intel® VTune™ Amplifier

Faster, Scalable Code Faster

Get the Data You Need
- Hotspot (Statistical call tree), Call counts (Statistical)
- Thread Profiling – Concurrency and Lock & Waits Analysis
- Cache miss, Bandwidth analysis...
- GPU Offload and OpenCL™ Kernel Tracing

Find Answers Fast
- View Results on the Source / Assembly
- OpenMP Scalability Analysis, Graphical Frame Analysis
- Filter Out Extraneous Data – Organize Data with Viewpoints
- Visualize Thread & Task Activity on the Timeline

Easy to Use
- No Special Compiles – C, C++, C#, Fortran, Java, ASM
- Visual Studio* Integration or Stand Alone
- Local & Remote Data Collection, Command Line
- Analyze Windows* & Linux* data on OS X*
Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2018, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804