Performance Tuning Methodology
(Intel® VTune™ Amplifier XE)

Dr.-Ing. Michael Klemm
Software and Services Group
Intel Corporation
(michael.klemm@intel.com)
Legal Disclaimer & Optimization Notice

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.

Performance tests and ratings are measured using specific computer systems and/or components
and reflect the approximate performance of Intel products as measured by those tests. Any
difference in system hardware or software design or configuration may affect actual performance.
Buyers should consult other sources of information to evaluate the performance of systems or
components they are considering purchasing. For more information on performance tests and on

Copyright © 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, Phi,
VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries. *Other names
and brands may be claimed as the property of others.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

Notice revision #20110804
Agenda

• Performance Tuning Methodology
• Intel® VTune™ Amplifier XE: User Interface
• Fundamental Analysis: Hotspots
• Finding Issues in Parallel Applications
• Using the Performance Monitoring Unit
Agenda

• Performance Tuning Methodology
• Intel® VTune™ Amplifier XE: User Interface
• Fundamental Analysis: Hotspots
• Finding Issues in Parallel Applications
• Using the Performance Monitoring Unit
Holistic toolset for the parallel software development lifecycle

- DESIGN
- CODE & DEBUG
- VERIFY
- TUNE

Intel® Cluster Studio XE adds:

- Intel® MPI
- Intel® Trace Analyzer and Collector
The Software Optimization Process

- The process of improving the software by eliminating bottlenecks so that it operates more efficiently on a given hardware and uses resources optimally
  - Identifying bottlenecks in the target application and eliminating them appropriately is the key to an efficient optimization

- There are many optimization methodologies, which help developers answer the questions of
  - Why to optimize?
  - What to optimize?
  - To what to optimize?
  - How to optimize?

These methods aid developers to reach their performance requirements.
Performance Analysis Methodology
Optimization: A Top-down Approach

• Use top down approach
• Understand application and system characteristics
  – Use appropriate tools at each level

System:
- System Config, BIOS, OS
- Network I/O, Disk I/O, Database Tuning, etc.

Application:
- Application Design
- Algorithmic Tuning
- Driver Turning
- Parallelization

Processor:
- Cache/Memory Instructions
- SIMD
- others
Performance Analysis Methodology
Optimization: A Top-down Approach

1. Create a Benchmark
2. Collect Data
3. Analyze Data and Identify Performance Problems
4. Fix the problems in your code or system
5. Is Problem Fixed?
   - No
   - Yes
6. Are performance requirements met?
   - No
   - Yes

- Repeatable
- Representative
- Easy to run
- Verifiable
- Measure elapsed time
- Reasonable coverage
- Precision

System

Application

Processor
When to Stop

• Is architecture at maximum efficiency?
  – What this means: calculating theoretical maximum.
  – Know about best values for CPI or IPC.
  – Know the maximum FLOPS for the data type used.

• Is the performance requirement fulfilled?
  – What are the performance requirements?
  – Incrementally complete optimizations until done.
  – Key question: Are you “happy” with it?

CPI: Cycles per Instructions
IPC: Instructions per Cycle
FLOPS: Floating-Point Oper. Per Sec.
Questions to Ask Yourself

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

— Donald Knuth

Quality code is:

- Portable
- Readable
- Maintainable
- Reliable

Intelligently sacrifice quality for performance
Amdahl’s Law Slightly Reinterpreted

1 core, t=140

1 core, optimized

1 cores, optimized even more

A 2x improvement in the hotspots overall leads to 1.5x

A 10x improvement in the hotspots leads to 2.8x
Agenda

• Performance Tuning Methodology
• Intel® VTune™ Amplifier XE: User Interface
• Fundamental Analysis: Hotspots
• Finding Issues in Parallel Applications
• Using the Performance Monitoring Unit
Selecting type of data collection

All available analysis types

Different ways to start the analysis

Helps creating new analysis types

Copy correct command line syntax to clipboard
VTune™ Amplifier XE
GUI Layout

Adjust Data Grouping

- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack

... (Partial list shown)

Click [+] for Call Stack

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

Zoom In And Filter On Selection
Filter In by Selection
Remove All Filters

Filter by Module & Other Controls

No filters are applied
Module: [All]
Call Stack Mode: Only user functions
**VTune™ Amplifier XE**

**GUI Layout**

**Hotspots** - View CPU time hotspots and stacks

- **Time on Source / Asm**
  - Quickly scroll to hot spots.
  - Scroll Bar “Heat Map” is an overview of hot spots

- **Quick Asm navigation:**
  - Select source to highlight Asm
  - Right click for instruction reference manual

- **Click jump to scroll Asm**
VTune™ Amplifier XE

GUI Layout

- Optional: Use API to mark frames and user tasks
- Optional: Add a mark during collection
Agenda

• Performance Tuning Methodology
• Intel® VTune™ Amplifier XE: User Interface
• Fundamental Analysis: Hotspots
• Finding Issues in Parallel Applications
• Using the Performance Monitoring Unit
Readying Your Application for Intel VTune Amplifier XE

• You should run Amplifier XE on a “Released/Optimized” build.

• Symbols Allow you to view the source (not just the assembly)
  – Windows: /Zi
  – Linux: –g

• Intel Threading Runtimes need instrumented runtimes
  – TBB: Define TBB_USE_THREADING_TOOLS
  – OpenMP: Use Intel Dynamic Version of OpenMP

• Call Stack Mode – Requires use of the dynamic version of the C Runtime library to properly attribute System Calls
  – Windows use:/MD(d)
  – Linux do not use: -static
Analysis Types

Hotspots

- For each sample, capture execution context, time passed since previous sample and thread CPU time
- Allows time spent in system calls to be attributed to user functions making the call
- Provides additional knobs:
  - The defaults for Hotspot analysis are configurable and can be done so by creating a custom analysis type inherited from Hotspots
Analysis Types

Lightweight Hotspots

• Similar to Hotspot Analysis
  – Sampling is performed with the SEP collector
  – Driver is required

• Stack walking is not performed
  – Only hotspots are reported

• Samples are taken more frequently, but may have less accurate timing information

• Analysis may be performed for a single application or for the entire system
Lightweight Hotspots vs. Hotspots

Smoke

- Mostly correlates, however the default attribution of system time in Hotspots is to the user function making the system call.
Lightweight Hotspots vs. Hotspots

Smoke

• Setting __________ gives you better correlation to the hotspot report by Lightweight Hotspot analysis type
Agenda

• Performance Tuning Methodology
• Intel® VTune™ Amplifier XE: User Interface
• Fundamental Analysis: Hotspots
• Finding Issues in Parallel Applications
• Using the Performance Monitoring Unit
Issues in Parallel Applications

• Load imbalance
  – Work distribution is not optimal
  – Some threads are heavily loaded, while others idle
  – Slowest thread determines total speed-up

• Locking issues
  – Locks prohibit threads to concurrently enter code regions
  – Effectively serialize execution

• Parallelization overhead
  – With large no. of threads, data partition get smaller
  – Overhead might get significant (e.g. OpenMP startup time)
Issues in Parallel Applications

• Load imbalance
  – Work distribution is not optimal
  – Some threads are heavily loaded, while others idle
  – Slowest thread determines total speed-up

• Locking issues
  – Locks prohibit threads to concurrently enter code regions
  – Effectively serialize execution

• Parallelization overhead
  – With large no. of threads, data partition get smaller
  – Overhead might get significant (e.g. OpenMP startup time)
**Threading Analysis Terminology**

- **Elapsed Time**: 6 seconds
- **CPU Time**: $T_1 \ (4s) + T_2 \ (3s) + T_3 \ (3s) = 10 \ seconds$
- **Wait Time**: $T_1(2s) + T_2(2s) + T_3 \ (2s) = 6 \ seconds$
Analysis Types

Concurrency
Hotspots Analysis vs. Concurrency Analysis

• Hotspot Analysis and Concurrency Analysis are similar:
Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the thread spent running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting higher than CPU usage if threads are in the runnable state and not consuming CPU time.

Average

Target
Issues in Parallel Applications

• Load imbalance
  – Work distribution is not optimal
  – Some threads are heavily loaded, while others idle
  – Slowest thread determines total speed-up

• Locking issues
  – Locks prohibit threads to concurrently enter code regions
  – Effectively serialize execution

• Parallelization overhead
  – With large no. of threads, data partition get smaller
  – Overhead might get significant (e.g. OpenMP startup time)
## Analysis Types

### Lock and Waits

<table>
<thead>
<tr>
<th>Sync Object</th>
<th>Function</th>
<th>Wait Time</th>
<th>Wait Count</th>
<th>Spin Time</th>
<th>Module</th>
<th>Object Type</th>
<th>Object Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Objects</td>
<td></td>
<td>112.517s</td>
<td>5,325</td>
<td>0s</td>
<td>[Unknown]</td>
<td>Constant</td>
<td>[Unknown]</td>
</tr>
<tr>
<td>Manual Reset Event 0xae37</td>
<td></td>
<td>109.238s</td>
<td>41</td>
<td>0s</td>
<td>[Unknown]</td>
<td>Manual Reset Event DLLStopPlugin</td>
<td></td>
</tr>
<tr>
<td>Manual Reset Event 0xfa6d</td>
<td></td>
<td>74.068s</td>
<td>26</td>
<td>0s</td>
<td>[Unknown]</td>
<td>Manual Reset Event LdrGetProcedureAddressEx</td>
<td></td>
</tr>
<tr>
<td>Thread Pool</td>
<td></td>
<td>57.628s</td>
<td>235</td>
<td>0s</td>
<td>[Unknown]</td>
<td>Constant</td>
<td>[Unknown]</td>
</tr>
<tr>
<td>Sleep</td>
<td></td>
<td>57.371s</td>
<td>5,234</td>
<td>0.193s</td>
<td>[Unknown]</td>
<td>Constant</td>
<td>[Unknown]</td>
</tr>
<tr>
<td>Unknown 0x991c9877</td>
<td></td>
<td>56.974s</td>
<td>6,337</td>
<td>0s</td>
<td>[Unknown]</td>
<td>Unknown</td>
<td>LdrGetProcedureAddressEx</td>
</tr>
<tr>
<td>TBB Scheduler</td>
<td></td>
<td>41.457s</td>
<td>2,200</td>
<td>11.301s</td>
<td>[Unknown]</td>
<td>Constant</td>
<td>TaskManagerTBB::Init</td>
</tr>
<tr>
<td>[Unknown]</td>
<td></td>
<td>17.061s</td>
<td>865</td>
<td>0s</td>
<td>[Unknown]</td>
<td>[Unknown]</td>
<td>[Unknown]</td>
</tr>
<tr>
<td>Stream ../media/graphics/</td>
<td></td>
<td>0.457s</td>
<td>183</td>
<td>0.057s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>Ogre::FileSystemArchive::open</td>
</tr>
<tr>
<td>Stream ../media/sounds/h</td>
<td></td>
<td>0.440s</td>
<td>171</td>
<td>0.063s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>Framework::GDFParser::EndElement</td>
</tr>
<tr>
<td>Stream Ogre.log 0x501382c</td>
<td></td>
<td>0.397s</td>
<td>193</td>
<td>0.059s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>Ogre::Log::Log</td>
</tr>
<tr>
<td>Stream Smoke.gdf 0xf2b92</td>
<td></td>
<td>0.386s</td>
<td>11</td>
<td>0.006s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>PlatformManager::FileSystem::GetFilename</td>
</tr>
<tr>
<td>Stream ../media/sounds/inv</td>
<td></td>
<td>0.306s</td>
<td>119</td>
<td>0.037s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>Framework::GDFParser::EndElement</td>
</tr>
<tr>
<td>Stream ../media/physicsINV</td>
<td></td>
<td>0.247s</td>
<td>5</td>
<td>0.011s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>hkStdioStreamReader::hkStdioS</td>
</tr>
<tr>
<td>Stream ../media/graphics/</td>
<td></td>
<td>0.136s</td>
<td>41</td>
<td>0.022s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>Ogre::FileSystemArchive::open</td>
</tr>
<tr>
<td>Stream ../media/sounds/h</td>
<td></td>
<td>0.134s</td>
<td>13</td>
<td>0.018s</td>
<td>[Unknown]</td>
<td>Stream</td>
<td>TaskManagerTBB::ParallelFor</td>
</tr>
</tbody>
</table>

Selected 1 row(s): 112.517s 5,325
Timeline Visualizes Thread Behavior

- Retrieve additional information about waiting threads
Timeline Visualizes Thread Behavior

- Retrieve additional information on thread transitions
Drilling down into Thread Behavior

- Reveal source code that causes thread transitions
Agenda

• Performance Tuning Methodology
• Intel® VTune™ Amplifier XE: User Interface
• Fundamental Analysis: Hotspots
• Finding Issues in Parallel Applications
• Using the Performance Monitoring Unit
Performance Monitoring Unit (PMU)

- **Per-core PMU:**
  - Each core provides 2 programmable counters and 1 fixed counters.
  - The programmable per-core counters can be configured to investigate front-end/micro-op flow issues, stalls inside a processor core.

- **Uncore PMU:**
  - Uncore of the coprocessor has four counters to monitor uncore events
  - Can be used to investigate memory behavior and global on-chip issues
Event Based Performance Analysis
Event Based Sampling (EBS)

- Both architectural and non-architectural processor events can be monitored using sampling and counting technologies

**Sampling:** Allows to profile all active software on the system, including operating system, device driver, and application software.

- Event-based samples are collected periodically after a specific number of processor events have occurred while the program is running
- The program is interrupted, allowing the interrupt handling driver to collect the Instruction Pointer (IP), load module, thread and process ID's
- Instruction pointer is then used to derive the function and source line number from the debug information created at compile time
How Event Based Sampling (EBS) Works

A performance counter increments on the CPU every time an event occurs. A sample of the execution context is recorded every time a performance counter overflows.

Events = samples * sample after value
Native Launch configuration

- Application settings:
  - Application: ssh
  - Parameters: mic0 "<app startup>"
  - Working directory: Usually does not matter
  - Don’t forget to set search directories under “All files”
Application Configuration

Choose Analysis Type

- Access Contention
- Branch Analysis
- Client Analysis
- Core Port Saturation
- Cycles and uOps
- Loop Analysis
- Memory Access
- Port Saturation
- Intel Atom Processor Analysis
- General Exploration
- Knights Corner Platform Analysis
- Lightweight Hotspots
- Power Analysis
- CPU Sleep States
- CPU Frequency
- Custom Analysis
- Branch Mispredicts
- Cache Misses
- Execution Stalls
- False Sharing

Knights Corner Platform - Lightweight Hotspots

Identify your most time-consuming source code. Unlike Hotspots, Lightweight Hotspots has lower overhead because it does not collect stack information. It can also be used to sample all processes on a system. This analysis type uses hardware event-based sampling collection. Press F1 for more details.

List of MIC cards (e.g. 0.1.2.3): 0

Details

Events configured for CPU: Intel(R) Xeon(R) / Core i7 980X Processor

NOTE: For analysis purposes, Intel VTune Amplifier XE 2013 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Sample After</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED</td>
<td>10000000</td>
<td>Number of cycles during which</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTRUCTIONS_EXECUTED</td>
<td>10000000</td>
<td>Number of instructions execut</td>
</tr>
</tbody>
</table>
Configuring a User-defined Analysis
### Some useful events and metrics

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Event name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall-clock profiling</td>
<td>CPU_CLK_UNHALTED, INSTRUCTIONS_EXECUTED (or EXEC_STAGE_CYCLES)</td>
</tr>
<tr>
<td>Main memory bandwidth</td>
<td>L2_DATA_READ_MISS_MEM_FILL, L2_DATA_WRITE_MISS_MEM_FILL</td>
</tr>
<tr>
<td>L1 Cache misses</td>
<td>DATA_READ_MISS_OR_WRITE_MISS</td>
</tr>
<tr>
<td>TLB misses and page faults</td>
<td>DATA_PAGE_WALK, LONG_DATA_PAGE_WALK, DATA_PAGE_FAULT</td>
</tr>
<tr>
<td>Vectorized code execution</td>
<td>VPU_INSTRUCTIONS_EXECUTED, VPU_ELEMENTS_ACTIVE</td>
</tr>
<tr>
<td>Various hazards</td>
<td>BRANCHES_MISPREDICTED</td>
</tr>
<tr>
<td>Cycles per instruction</td>
<td>CPU_CLK_UNHALTED / INSTRUCTIONS_EXECUTED</td>
</tr>
<tr>
<td>Memory Bandwidth (used by all cores at once)</td>
<td>(L2_DATA_READ_MISS_MEM_FILL + L2_DATA_WRITE_MISS_MEM_FILL) * 64 / CPU_CLK_UNHALTED / Frequency</td>
</tr>
</tbody>
</table>
Example: Hotspots of OpenFOAM

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>C</th>
<th>Instructions R...</th>
<th>CPI</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDI_CH3i_Progress</td>
<td>23.1%</td>
<td>203,160,000,000</td>
<td>0.744</td>
<td>libmpi.so.4.1</td>
</tr>
<tr>
<td>[vmlinux]</td>
<td>21.2%</td>
<td>479,780,000,000</td>
<td>3.386</td>
<td>vmlinux</td>
</tr>
<tr>
<td>List</td>
<td>7.5%</td>
<td>286,010,000,000</td>
<td>2.032</td>
<td>libOpenFOAM.so</td>
</tr>
<tr>
<td>__svml_div8_mask</td>
<td>6.5%</td>
<td>215,520,000,000</td>
<td>2.308</td>
<td>simpleFoam.so</td>
</tr>
<tr>
<td>[libc-2.14.90.so]</td>
<td>2.2%</td>
<td>64,720,000,000</td>
<td>2.655</td>
<td>libc-2.14.90.so</td>
</tr>
<tr>
<td>List</td>
<td>2.2%</td>
<td>67,780,000,000</td>
<td>2.474</td>
<td>libOpenFOAM.so</td>
</tr>
<tr>
<td>__l_MPI__intel_irb_mempcy</td>
<td>2.0%</td>
<td>7,410,000,000</td>
<td>20.333</td>
<td>libmpi.so.4.1</td>
</tr>
<tr>
<td>Foam:lduMatrix::Amul</td>
<td>1.7%</td>
<td>56,470,000,000</td>
<td>2.301</td>
<td>libOpenFOAM.so</td>
</tr>
<tr>
<td>__intel_irb_mempcy</td>
<td>0.1%</td>
<td>7,880,000,000</td>
<td>12.996</td>
<td>simpleFoam.so</td>
</tr>
</tbody>
</table>

Selected 1 row(s): 23.1% 263,160,000,000 6.744