Agenda

• Introductory Comments

• Basic VTune

• Advanced VTune
Introductory Comments
Section 1
A Screenshot of VTune
VTune

• Gives Visibility

• Written by ‘engineers for engineers’
VTune – Gives Visibility

• High-level program analysis
  - Identifying hotspots
  - Working calling heirachy of program
  - NB: Alt tools Intel® Parallel Amplifier \ Code Coverage

• System wide view of computer

• Low-level architectural view
Important Design Criteria of VTune

• The act of measuring performance shouldn’t change the performance of the software
  - Accurate, representative results with no instrumentation
  - Low intrusive performance measurements
  - Interrupt based sampling driven by CPUs registers

• Programmers need to see System-wide performance
  - Identify the software that is consuming most of the CPU
  - Application, shared objects, O.S. layer, device driver, ...

• Source code should be the normal view of the performance data
  - Each executable statement annotated with perf. data
Performance Analysis Technologies

- Identify Performance Bottlenecks
  - Interrupt based sampling using CPU registers
  - Lower Overhead, less data

- Examine flow of control through the app
  - Which functions took the longest
  - Which functions were blocked the longest
  - Calling sequence critical path
  - Higher Overhead, more data
VTune™ Performance Analyzer - Products

- VTune™ Performance Analyzer
  - GUI-based tool integrated into Visual Studio*
  - Remote capability to Linux and Windows*
  - Command line interface included

- VTune Performance Analyzer for Linux*
  - GUI-based tool runs integrated into Eclipse
  - Remote capability to Linux*
  - Stand alone viewers
  - Command-line version for script invocation

- VTune™ Analyzer Driver Kit
  - Rebuild VTune™ Analyzer Linux driver for non-standard kernels
Basic VTune

Section 2

Identifying Hotspots.
Call Graphs.
Multicore Support
Identifying Hotspots

Pinpointing places where an application could be parallelised
The Big Question

“How can I make my code run faster?”
Today’s Question

“Where do I split up my code to take advantage of multiple CPU cores?”
The task, Identifying the Hot Spot...
... and Splitting up the Work.
Finding a Hotspot
Sampling
Sampling: Process view
Sampling: Hotspot view
Sampling: Source View

```
#include "multiply_3.h"

void dgemm(
    const double *A, const double *B, double *C)
{
    unsigned i, j, k;
    for (i = 0; i < NUM; ++i) {
        const double *B_i = B + i*NUM;
        for (j = 0; j < NUM; ++j) {
            const double *B_j = B_i + j*NUM;
            double cij = *(C + j*NUM + i);
            for (k = 0; k < NUM; ++k) {
                cij += *(A_i + k*NUM) * *B_j;
            }
        }
        *(C + i*NUM) = cij;
    }
}
```

Introduction to Tuning with the Intel® VTune™ Performance Analyzer

Copyright © 2009, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners*
Call Graph Profiling

- Helps you to better understand call flow

- 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

- Most useful for algorithmic tuning
Creating a 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 Wait Time</td>
<td>Time spent in a function and its children when the thread is blocked</td>
</tr>
<tr>
<td>Wait 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>
Call graph: Application workflow

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.

Filter view by self time
Graph Navigation Window

Use the graph navigation window for an overview of the entire call graph.
**Call List View**

![VTune™ Performance Analyzer](image)  

**Switch between call list and call graph views here.**

### Call List View

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>dd_Term</td>
<td></td>
<td>1</td>
<td>0</td>
<td>260</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>dd_Unlock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>DrawPieces</td>
<td></td>
<td>400</td>
<td>1,376,760</td>
<td>3,125,167</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>exit</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1,261</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>free</td>
<td></td>
<td>7</td>
<td>0</td>
<td>153</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>GetOSCpuSpeed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Focus function**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s399.exe</td>
<td>Thread_E3C</td>
<td>DrawPieces</td>
<td></td>
<td>400</td>
<td>1,376,750</td>
<td>3,125,167</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Caller Function**

<table>
<thead>
<tr>
<th>Caller Function</th>
<th>Contribution</th>
<th>Edge Time</th>
<th>Edge Calls</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>s399.exe</td>
<td>100.0%</td>
<td>3,125,167</td>
<td></td>
<td>Thread_E3C</td>
</tr>
</tbody>
</table>

**Callee Function**

<table>
<thead>
<tr>
<th>Callee Function</th>
<th>Contribution</th>
<th>Edge Time</th>
<th>Edge Calls</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd_Unlock</td>
<td>0.0%</td>
<td>1247</td>
<td>400</td>
<td>Thread_E3C</td>
</tr>
<tr>
<td>RectCopy</td>
<td>55.5%</td>
<td>1745403</td>
<td>7500000</td>
<td>Thread_E3C</td>
</tr>
<tr>
<td>dd_Lock</td>
<td>0.0%</td>
<td>757</td>
<td>400</td>
<td>Thread_E3C</td>
</tr>
</tbody>
</table>
## 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>
Advanced Configuration

- Set instrumentation levels. Helps control overhead
- Select which functions are instrumented. Helps control overhead
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**: This is useful for long runs and very large applications. If you do not set this, the machine might run low on memory.
Function Selection

Click here to enable or disable instrumentation for a particular function.
Multi-core Support

- Sampling Over Time

- The CPU Button
Sampling Over time view
Demo 3

Sampling Over Time
Sampling Over Time - Linux

- Feature is disabled by default
- Enable it by setting environment variable

```
export VTUNE_OVER_TIME=1
```
### Introduction to Tuning with the Intel® VTune™ Performance Analyzer

**Sampling Over Time - Linux**

<table>
<thead>
<tr>
<th>ModuleName</th>
<th>ProcessName</th>
<th>CPU_CLK_UNHALTED samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>gzip</td>
<td>2509</td>
</tr>
<tr>
<td>vmlinux-2.6.9-42</td>
<td>gzip</td>
<td>61</td>
</tr>
<tr>
<td>jbd</td>
<td>gzip</td>
<td>22</td>
</tr>
<tr>
<td>libc-2.3.4.so</td>
<td>gzip</td>
<td>9</td>
</tr>
<tr>
<td>ex5</td>
<td>gzip</td>
<td>7</td>
</tr>
<tr>
<td>Others4</td>
<td>gzip</td>
<td>2</td>
</tr>
<tr>
<td>x86-3.4.so</td>
<td>gzip</td>
<td>1</td>
</tr>
<tr>
<td>Others02</td>
<td>gzip</td>
<td>1</td>
</tr>
</tbody>
</table>

**CPU_CLK_UNHALTED, CORE samples**

<table>
<thead>
<tr>
<th>1&lt;&lt;x&lt;&lt;2</th>
<th>2&lt;&lt;x&lt;&lt;3</th>
<th>3&lt;&lt;x&lt;&lt;4</th>
<th>4&lt;&lt;x&lt;&lt;5</th>
<th>5&lt;&lt;x&lt;&lt;6</th>
<th>6&lt;&lt;x&lt;&lt;7</th>
<th>7&lt;&lt;x&lt;&lt;8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Processes
- Threads
- Modules
- Hotspots
- Processes Over Time
- Modules Over Time

*Other brands and names are the property of their respective owners*
Sampling Versus Call Graph

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

<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>Must re-link with /Fixed:no, automatically instruments</td>
</tr>
<tr>
<td>Can sample based time and other processor events</td>
<td>Results are based on time</td>
</tr>
</tbody>
</table>
Advanced VTune
Section 3

Events.
Programmatic control.
Command line.
The life of a program instruction

1. Instruction read from memory
2. Instruction fed to Decoder
3. Micro-ops (uops) generated
4. uops queued in RS
5. uops dispatched
6. Results sent to ROB
7. Instruction marked – all uops executed
8. Instruction sent for retirement

Memory Sub-system
Inst. Fetch Branch Pred
Decoder
Reservation Station
Executive Units
Reorder Buffer
Retirement

*Other brands and names are the property of their respective owners
Hardware Performance Events

BUS_TRANS_ANY.ALL_AGENTS

Inst. Fetch Branch Pred

RS_UOPS_DISPATCHED.CYCLES_NONE

Decoder

MEM_LOAD RETIRED.L2_MISS

CPU_CLK_UNHALTED.CORE

Retirement

INST RETIRED.ANY
All programs consume cycles

These cycles consist of

- Cycles where instructions are usefully executed
- Cycles where instructions are executed, but the results never used
- Cycles when nothing happens

Goal of performance tuning is to reduce each of these
How to tell if a program is running efficiently

- Look at the CPI

- See if there are lots of cycles where nothing is happening
CPI - Cycles Per Instruction

- CPI – Cycles Per Instruction
- Measure of program efficiency
- Low good, High bad
- Using CPI to log tuning improvements can be misleading
  - E.g. Auto-vectorisation
  - # Instructions retired drops (2x or greater)
  - Cycles consumed doesn’t at same rate
  - Hence CPI increases
Types of applications

- Data Layout Dominated
  - Loop dominated
  - Bandwidth dominated
  - Typical HPC

- Data Value Dominated
  - Pointer chasing
  - Latency dominated
  - E.g Databases

*Loop vs branch dominated*
Anecdotal Performance Statements ...

- Most common performance problem
  - Cache usage
- Most astounding tuning improvement
  - Vectorisation
How Event-based Sampling (EBS) Works

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?
Sampling: The Statistical Method of Finding Hotspots

- The sampling collector
- Periodically interrupts the processor
  - Event-based: Triggered by the occurrence of a certain number of microarchitectural events
  - Time-based
- 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.
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**
  - Overhead can be reduced further by disabling kernel (Ring 0) sampling
Demo 4

Configuring Events
Sampling: Events

L2 Cache Reads
This event indicates that a normal, unlocked, load memory access was received by the L2 cache. It includes:
- Only L2 cacheable memory accesses; it does not include non-memory accesses, such as I/O accesses, and memory accesses that bypass the L2, such as uncachable memory accesses or write-through.
- It includes L2 cacheable TLB miss memory accesses.
- Splits are irrelevant for L2 accesses, since they are line oriented.
Profiling with command line

• Three steps:
  – Configure
  – Collect
  – View

• Configure: No wizards
• Collect: Sampling and Callgraph
• View: text or graphical
• Configure and Collect can be accomplished at the same time
• Other commands for managing project, collections, and results
Profiling with command line (cont.)

- See man pages for help:
  - ‘man vtl’
  - ‘man sampling’
  - ‘man callgraph’
  - ‘man code’

- Documentation in
  /opt/intel/vtune/doc/users_guide
  - Invoke with:
    - Mozilla/Firefox
    - /opt/intel/vtune/doc/users_guide/index.htm
Profiling with Sampling

- Commands have structure:

```
$ vtl activity -c sampling -app myapp run
```

- Invokes analyzer
- Tells analyzer what to do: e.g. create an activity,
- Using the sampling [c]ollector,
- with this application,
- Finally, execute this activity
Profiling with Sampling (cont.)

• View most recent results:
  • vtl view -- text based output
  • vtl view --gui -- graphical display using X windows

• View project:
  • vtl show

• View older results:
  • vtl view --a <activity#>
  • e.g., vtl view --a a15
Profiling with Sampling (standalone viewer example)
Introduction to Tuning with the Intel® VTune™ Performance Analyzer

Call Graph Profiling command line

$ vtl activity –c callgraph –app myapp –moi myapp run

- Invokes analyzer
- Tells analyzer what to do: e.g. create an activity,
- Using the callgraph [c]ollector,
- with this application,
- Module Of Interest, required
- Finally, execute this activity
Call Graph Profiling command line (cont.)

• View most recent results:
  • vtl view -- text based output
  • vtl view -gui -- graphical display using X windows

• View project:
  • vtl show

• View older results:
  • vtl view -a <activity#>
  • e.g., vtl view -a a15
Programmatic Control

- Supported in Sampling and Call Graph
- Pause / Resume
  - Allows skipping of “uninteresting” code – ex: initialization
  - GUI buttons
  - APIs
Three Functions

- **VTStartSampling**
  
  ```c
  U32 VTStartSampling(VTUNE_SAMPLING_PARAMS *samParams);
  ```

- **VTStopSampling**
  
  ```c
  U32 VTStopSampling(U32 bBindSamplingResults);
  ```

- **VTBindSamplingResults**
  
  ```c
  U32 VTBindSamplingResults(char *tb5Filename);
  ```
Example Application

#include "VtuneApi.h"

int main ()
{
    VTUNE_EVENT events[] = {
        { 10000, 0, 0, 0, "Clockticks" },
        { 10000, 0, 0, 0, "Instructions Retired" },
    };

    U32 numEvents = sizeof(events) / sizeof(VTUNE_EVENT);

    VTUNE_SAMPLING_PARAMS params = {
        sizeof(VTUNE_SAMPLING_PARAMS), sizeof(VTUNE_EVENT),
        0, 0, 0, 0, 1000, 40, 1,
        numEvents, events, "results.tb5",
        0, 0, ""
    };

    U32 u32Return = VTStartSampling(&params);

    // Some user code goes here...

    u32Return = VTStopSampling(1);
Pause \ Resume

- **VTPauseSampling()**: pauses sampling data collection only.

- **VTResumeSampling()**: resumes sampling data collection only.

**Usage**

- Start VTune activity with sampling disabled
  - `-start-paused` option when creating an activity from command line
- Insert resume\pause commands at will