Performance Tuning using VTune and the Thread Profiler

Stephen Blair-Chappell
Technical Consulting Engineer
Intel Compiler Labs
Agenda

VTune Performance Analyzer

Thread Profiler
Features
Intel® VTune – lots of things ‘under one hood’

- Analyzer Projects
  - Counter Monitor Wizard
  - Sampler Wizard
  - Call Graph Wizard

- Threading Wizards
  - Thread Checker
  - Thread Profile
Agenda

- VTune Performance Analyzer Introduction
- Sampling
- Call Graph
- Multi-threading support
- Intel® Performance-Tuning Utility
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 (ex: errata kernels, modified kernels)
  - Red Hat, SuSE production distributions supported
Some Performance Tool Goals

- 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
Profiling with vtlec (Linux*)

- Integrated into the Eclipse* environment
- Sampling and Callgraph profiling
- Wizards walk you through configuration
- Managing of projects, collections, and results all done graphically
Sampling Results - Modules
Sampling Results - Source

![Image of VTune(TM) Performance Tools interface showing source code analysis]

To make it faster, you can try setting compiler switches for streaming instructions if you don't need full performance.

* A table lookup for square root. To view disassembly.

Icons at the top left of the window and clicking them can help.

```c
rassm2[i] = rassm2[i] + rassm2[i] / sqrt(rassm2[i]);
```

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Clocks</th>
<th>Instr. Ret.</th>
<th>Cycles per Retired Instruction - CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x146</td>
<td>rassm2</td>
<td>25.82%</td>
<td>1.29%</td>
<td>0.1111</td>
</tr>
<tr>
<td>0x18</td>
<td>test_f</td>
<td>2.03%</td>
<td>1.83%</td>
<td>0.83/83</td>
</tr>
<tr>
<td>0x86</td>
<td>test_f2</td>
<td>1.65%</td>
<td>9.44%</td>
<td>0.520/12</td>
</tr>
</tbody>
</table>

Threading Tools

Copyright © 2007, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.
Call Graph Results

Threading Tools
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
- Using the sampling collector,
- Tells analyzer what to do: e.g. create an activity,
- Finally, execute this activity
- with this application,
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)
Call Graph Profiling command line

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

Invokes analyzer

Using the callgraph collector,

Tells analyzer what to do: e.g. create an activity,

Finally, execute this activity

Module Of Interest, required

with this application,
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
Profiling with Sampling (standalone viewer example)
Programmatic Control

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

- VTune Performance Analyzer Introduction
- Sampling
- Call Graph
- Multi-threading support
- Intel ® Performance-Tuning Utility
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.
How Event-based Sampling (EBS) Works

Select Event Signal → Count Down → “Sample After” Number

Underflow to Zero → Interrupt CPU to Take Sample

How do you choose a “Sample After” number?
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
Sampling

Module of Interest
Sampling: Process view

Module of Interest

Process view
Sampling: Hotspot view
Sampling: Source View

```
#include "multiply_d.h"

void
dgemm ()
{
    const double *A, const double *B, double *C
    
    unsigned i, j, k;
    
    for (i = 0; i < NUM; i++) {
        const double *Ai = A + i;
        for (j = 0; j < NUM; j++) {
            const double *Bj = B + j * NUM;
            double cij = *(Ai + j * NUM + i);
            
            for (k = 0; k < NUM; k++) {
                cij += *(Ai + k * NUM) * Bk;
            }
        }
    }
}

/*
void
dgemm ()
{
    const double *A, const double *B, double *C
    
    unsigned hi, hj, bk;
    unsigned i, j, k;
```
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 L1, 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.
Sampling Over time view
Agenda

- VTune Performance Analyzer Introduction
- Sampling
- Call Graph
- Multi-threading support
- Intel® Performance-Tuning Utility
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
  - Includes function timing info, call counts, and calling relationships
    - Not system-wide: Only profiles code in applications call path in Ring 3
    - Most useful for algorithmic tuning
## 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

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Thread</th>
<th>Function</th>
<th>Class</th>
<th>Calls</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Callers</th>
<th>Callees</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd99.exe</td>
<td>Thread_E9C</td>
<td>dd_Term</td>
<td>1</td>
<td>0</td>
<td>260</td>
<td>3,125,167</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sd99.exe</td>
<td>Thread_E9C</td>
<td>dd_Unlock</td>
<td>400</td>
<td>0</td>
<td>1,247</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd99.exe</td>
<td>Thread_E9C</td>
<td>DrawPieces</td>
<td>400</td>
<td>1,376,776</td>
<td>3,125,167</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd99.exe</td>
<td>Thread_E9C</td>
<td>exit</td>
<td>1</td>
<td>0</td>
<td>1,261</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd99.exe</td>
<td>Thread_E9C</td>
<td>free</td>
<td>7</td>
<td>0</td>
<td>153</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd99.exe</td>
<td>Thread_E9C</td>
<td>GetOSCPuSpeed</td>
<td>1</td>
<td>0</td>
<td>55</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Focus function**

| sd99.exe | Thread_E9C | DrawPieces | 400 | 1,376,776 | 3,125,167 | 1 |

**Caller Function**

<table>
<thead>
<tr>
<th>Caller</th>
<th>Contribution</th>
<th>Edge Time</th>
<th>Edge Calls</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applets</td>
<td>100.0%</td>
<td>3,125,167</td>
<td>400 Thread_E9C</td>
<td></td>
</tr>
</tbody>
</table>

**Callee Function**

<table>
<thead>
<tr>
<th>Callee</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 Thread_E9C</td>
<td></td>
</tr>
<tr>
<td>RectCopy</td>
<td>55.9%</td>
<td>1745403</td>
<td>7500000 Thread_E9C</td>
<td></td>
</tr>
<tr>
<td>dd_Lock</td>
<td>0.0%</td>
<td>757</td>
<td>400 Thread_E9C</td>
<td></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.
Function Selection

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

<table>
<thead>
<tr>
<th>Instrument</th>
<th>Function</th>
<th>Status</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ActivateActCtx</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddAtomA</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddAtomW</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddConsoleAliasA</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddConsoleAliasW</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddLocalAlternateComputerNameA</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddLocalAlternateComputerNameW</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddRefActCtx</td>
<td>Failed</td>
<td>The method is an Import Stub</td>
</tr>
<tr>
<td></td>
<td>AddVectoredExceptionHandler</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllocateUserPhysicalPages</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AllocConsole</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AreFileApisANSI</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AssignProcessToJobObject</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AttachConsole</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BackupRead</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BackupSeek</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BackupWrite</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BaseCheckAppcompatCache</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BaseCleanupAppcompatCache</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BaseCleanupAppcompatCacheSupport</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BaseDumpAppcompatCache</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BaseFlushAppcompatCache</td>
<td>Ok</td>
<td></td>
</tr>
</tbody>
</table>
## 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>
Agenda

- VTune Performance Analyzer Introduction
- Sampling
- Call Graph
- Multi-threading support
- Intel® Performance-Tuning Utility
VTune Can Help with Multi-Threading

- Identifies the time consuming regions
- Used to find proper level in the call-tree to thread
- Helps adjust load balance across threads

This is the level in the call tree where we need to thread

Work is not evenly distributed across threads
Agenda

- VTune Performance Analyzer Introduction
- Sampling
- Call Graph
- Multi-threading support
- Intel® Performance-Tuning Utility
Tuning Assistant
Two ways to get ‘Insight and Advice’

- 1. On a new Project (Automatic)

- 2. Context-sensitive advice in existing project
  - Highlight area of interest
  - Press <F8>
The Tuning Assistant Provides Tuning Advice Based on Performance Data
Intel® Tuning Assistant

Tuning Analysis for Sampling Results
[BNSHAH-P4HT] - Fri Dec 05 19:00:18 2003

Top 5 Hotspot Insights

Time-Based Coding Pitfalls

- 64K Aliasing: 0.95 sec processor time
  Seen in \textit{unsigned long cal\_pixel(struct tag\_Complex)}
  (RVA: 0x2e20-0x2ea1, process: mandelvtune.exe, module: mandelvtune.exe)

- 64K Aliasing: 0.43 sec processor time
  Seen in \textit{unsigned long}
  \texttt{calc\_Mandel\_32N\_Opt\_Offset\_cal\_Complex\_MAND \_\_struct (struct tag\_CALC\_MAND *)}
  (RVA: 0x2f0-0x30db, process: mandelvtune.exe, module: mandelvtune.exe)

- Blocked Store Forwards: 0.38 sec processor time
  Seen in \textit{unsigned long cal\_pixel(struct tag\_Complex)}
  (RVA: 0x2e20-0x2ea1, process: mandelvtune.exe, module: mandelvtune.exe)
For more detail, click hyperlink.
Thread Profiler
Agenda

- Look at Intel® Thread Profiler features
- Define Critical Path Analysis
- Examine Thread Profiler data views available
- Review common performance issues of multithreaded applications
  - Focus on Load imbalance
  - Focus on Synchronization contention
- Describe general optimizations to gain better performance
Why use the Thread Profiler?

- Developing efficient multithreaded applications is hard

- New performance problems are caused by the interaction between concurrent threads
  - Load imbalance
  - Contention on synchronization objects
  - Threading overhead

- Intel Thread Profiler gives good visibility - takes away the ‘guess work’
Critical Path Analysis

- Examines processor utilization to determine concurrency level of the application
- Concurrency is the number of active threads

Categorization shown for a system configuration with 2 processors

<table>
<thead>
<tr>
<th>Thread 3</th>
<th>Thread 2</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire lock L</td>
<td>Release L</td>
<td>Wait for Threads 2 &amp; 3</td>
</tr>
<tr>
<td>Release L</td>
<td>Acquire L</td>
<td>Threads 2 &amp; 3 Done</td>
</tr>
<tr>
<td>Wait for L</td>
<td>Release L</td>
<td></td>
</tr>
</tbody>
</table>

ConcURRENCY LEVEL

- **Idle**
- **Serial**
- **Under-subscribed**
- **Parallel**
- **Over-subscribed**

Time

- 0
- 5
- 10
- 15

Concurrency Level
Critical Path Analysis

- System Utilization
  - Relative to the system executing the application
    - **Idle**: no threads
    - **Serial**: a single thread
    - **Under-subscribed**: more than one thread, less than cores
    - **Parallel**: # threads == # cores
    - **Oversubscribed**: # threads > # cores

- Thread interaction categories
  - **Cruise**: threads running without interference
  - **Overhead**: thread operation overhead
  - **Blocking**: thread waiting on external event
  - **Impact**: thread preventing some other thread from executing

If the critical path is shortened, the application will run in less time
Execution Time Categories

- Analyze thread interaction and behavior along critical path
- Record objects that cause CP transitions

Categorization shown for a system configuration with 2 processors
Merging Concurrency and Behavior

- Start with system utilization
- Further categorize by behavior
Thread Profiler Views

- Critical Path View
  - Shows breakdown of the critical path
- Profile View
  - Shows the breakdown of selected critical paths
  - User can select other views of the selected profile
  - Concurrency level, threads, objects
- Timeline View
  - Shows thread activity and critical path transitions for the entire application
- Source View
  - Transition source view, creation source view
Profile View - Concurrency Level

Let’s look at the threads view. Threads ran simultaneously ~35% of the time.
Profile View - Thread View
Profile View - Objects View

Let’s look at Timeline View

This object caused all of the impact
Timeline View

![Timeline View Image]

- Thread active/unknown
- Thread inactive
- Show Critical Path
- Show Forks/Joins
- Critical Paths
- Edit Timeline

Threading Tools

Copyright © 2007, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Source View

```c
double do;

start = myThreadNum + 1;

// Break the iterations required to compute PI equally among all
// of the threads created by this application

for( int i = start; i < MaxIterations; i += globalNumThreads )
{
    dx = (i - 0.5) * globalInterval;
    // Since globalDSum is being updated, the access to global
    // has to be protected.
    //
    EnterCriticalSection(&globalCS);
    globalDSum = globalDSum + I(dx);
    //
    LeaveCriticalSection(&globalCS);
}

return myThreadNum; // thread exit code

} // PThreadFunc

// GetCommandLineNumThreads

// Grabs the number of threads from the commandline

int GetCommandLineNumThreads(int argc, char **argv)
{
    if ( argc == 1 )
        {
```
Common Performance Issues

- Load balance
  - Improper distribution of parallel work
- Synchronization
  - Excessive use of global data, contention for the same synchronization object
- Parallel Overhead
  - Due to thread creation, scheduling
- Granularity
  - No sufficient parallel work
Unbalanced Workloads

Threads are unbalanced
Active Times not equal
Synchronization

- By definition, synchronization serializes execution
- Lock contention means more idle time for threads
Synchronization Fixes

- Eliminate synchronization
  - Expensive but necessary “evil”
  - Use storage local to threads
    - Use local variable for partial results, update global after local computations
    - Allocate space on thread stack (alloca)
    - Use thread-local storage API (TlsAlloc)
  - Use atomic updates whenever possible
    - Some global data updates can use atomic operations (Interlocked API family)
Atomic Updates

• Use Win32 Interlocked* intrinsics in place of synchronization object

```c
static long counter;

// Fast
InterlockedIncrement (&counter);

// Slower
EnterCriticalSection (&cs);
    counter++;
LeaveCriticalSection (&cs);
```
Synchronization Fixes

- Reduce size of critical regions protected by synchronization object
  - Larger critical regions tie up sync objects longer; other threads sit idle longer waiting to acquire objects
  - Only accesses to shared variables need to be protected
Synchronization Fixes

• Use best synchronization object for job
  - Critical Section
    • Local object
    • Available to threads within the same process
    • Lower overhead (~8X faster than mutex)
  - Mutex
    • Kernel object
    • Accessible to threads within different processes
    • Deadlock safety (can only be released by owner)
• Other objects are available
Object Contention

These four threads...

...are impacting threads by this object
General Optimizations

- Serial Optimizations
  - Serial optimizations along the critical path should affect execution time
- Parallel Optimizations
  - Reduce synchronization object contention
  - Balance workload
  - Functional parallelism
- Analyze benefit of increasing number of processors
- Analyze the effect of increasing the number of threads on scaling performance