Top-Down Analysis simplifies Performance Optimizations through VTune™

Ahmad Yasin
Intel Core™ Monitoring & Analysis
Motivation
Motivation
Preface

• Performance Optimization Is Difficult
  - Complicated micro-architectures
  - Application/workload diversity
  - Unmanageable data
  - Tougher constraints
    - Time, Resources, Priorities

• Top Down Analysis Method
  - Identify the true bottleneck in a structured hierarchical process
  - Analysis is made easier for non-expert users
    - Simplified hierarchy avoids the u-arch high-learning curve
    - Adopts microarchitecture independent metrics
Agenda

✓ Motivation

• Challenges

• Top-Down Analysis Method
  - Top-Level deep dive
  - Memory breakdown
  - Frontend breakdown
  - Generality

• Demo with VTune™

• More Examples

• Summary
Performance Analysis

• Process
  - System Level
    - Memory setup
  - Application Level
    - Algorithm
  - Architectural & micro-architectural Levels
    - Vector code, Cache misses

• Assumptions/Caveats
  - CPU Bound (IA)
  - Predefined analysis goal
  - Goal: detect bottleneck
    - Not-a-goal: quantify speedup
  - Forward compatibility
Ahmad Yasin – Top-Down Analysis simplifies Optimization through VTune™ – Technion 2015

Where To Start In This Complex Microarchitecture?

Top Level counters are located here

Intel Core™ µarch

Front end of processor pipeline

Back end of processor pipeline
Challenges

Traditional Methods

• Naïve approach
  \[ \text{Stall}_\text{Cycles} = \sum \text{Penalty}_i \cdot \text{MissEvent}_i \]

• Unsuitable for out-of-orders (Gaps)
  1) Stalls Overlap
  2) Speculative Execution
  3) Workload-dependent penalties
  4) Predefined set of miss-events
  5) Superscalar inaccuracy

Top Down Analysis

• A hierarchy
  - Top-Down designated events at appropriate pipeline stages
  - “Hierarchical safety property”

• Addressing Gaps
  - Bad Speculation at the top
  - Generic top-down events, who
  - count when matters, and
  - count where matters
  - Occupancy events
Top Level Breakdown - the idea

- **Uop Issue?**
  - Yes: **Uop ever Retire?**
    - Yes: Retiring
    - No: Bad Speculation
  - No: **Back-end stall?**
    - Yes: Backend Bound
    - No: Frontend Bound
## Top Level Breakdown

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back End Stall</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Alloc Slot 0</td>
<td>-</td>
<td>v</td>
<td>-</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Alloc Slot 1</td>
<td>-</td>
<td>v</td>
<td>-</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Alloc Slot 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Alloc Slot 3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>v</td>
<td>-</td>
</tr>
<tr>
<td>Frontend Bound</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Backend Bound</td>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Retiring</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bad Speculation</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Classify Each Pipeline Slot Into 1 of 4 Categories

- **BackEnd stall?**
  - no
  - yes
- **Alloc?**
  - no
  - yes
- **FrontEnd Bound**
  - no
- **Bad Speculation**
  - no
- **Retiring**
  - no
  - yes
- **Uop ever Retire?**
  - no
  - yes

Ahmad Yasin - Top-Down Analysis simplifies Optimization through VTune™ - Technion 2015
Top-Down Analysis Hierarchy

Systematically Find True Bottleneck with Less Guess Work

Ahmad Yasin - Top-Down Analysis simplifies Optimization through VTune™ - Technion 2015
Top Level for SPEC CPU2006

```
<table>
<thead>
<tr>
<th>Retiring</th>
<th>Bad Speculation</th>
<th>Frontend Bound</th>
<th>Backend Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.5%</td>
<td>6.5%</td>
<td>13%</td>
<td>37%</td>
</tr>
</tbody>
</table>
```

INT Apps have quiet some Frontend/Bad Spec. issues

Most Apps Are Backend Bound, esp. FP

Top Down Correctly Characterizes All Workloads

SPEC rate 1-copy, Intel Compiler 13, IvyBridge @ 3 GHz
Motivation

Challenges

Top-Down Analysis Method

Top-Level idea
Memory breakdown
Frontend breakdown
Generality

Demo with VTune™
More Examples
Backend Bound

• First distinction
  - Core- vs Memory-Bound

• Memory Bound
  - Loads limited by which level
    - and Ext. Memory Latency vs. Bandwidth
  - Store Issues
  - Traditional perf issues plugged into the hierarchy
    - Data Sharing, Store Forward Blocks, False Sharing, ...

• Core Bound
  - Non-memory core-internal issues
  - Example: Divider, Execution Ports Utilization
Memory Bound (1-core vs 4-core)

Source: http://www.jaleels.org/ajaleel/workload/

L3 Cache Bound → External MEM Bound
Motivation

Challenges

Top-Down Analysis Method

- Top-Level idea

Memory breakdown

- Frontend breakdown

Generality

Demo with VTune™

- More Examples
Frontend Bound

• Front-end issues
  - Less encountered in traditional client/HPC, more common in servers/enterprise

• Breakdown
  - Rough Frontend Latency vs BW classification
  - Frontend Latency
    - Intervals with uop delivery starvation
    - Buckets: i-Cache Miss, iTLB Miss, Branch Resteers
  - Frontend Bandwidth
    - Intervals when supplied non optimal # of uops per cycles
    - Breakdown by Fetch source unit (DSB, MITE, LSD)
Ahmad Yasin – Top-Down Analysis simplifies Optimization through VTune™ – Technion 2015

Results: Front-end drilldown

Note Difference in Mis-prediction cost
The Generic Hierarchy (ISPASS 2014)

A user-defined criteria for analyzing a hotspot: CPU Bound ⇒ Analyze

Frontend Bound
- Fetch Latency
  - iTLB Miss
  - iCache Miss
  - Branch Resteers
  - Other
  - Fetch src 1
  - Fetch src 2
- Fetch Bandwidth

Bad Speculation
- Branch Misspredicts
- Machine Clears

Retiring (commit)
- BASE
  - FP-arith.
  - Other

Backend Bound
- Core Bound (compute)
  - Divider
  - Execution Ports Utilization
  - Stores Bound
    - L1 Bound
    - L2 Bound
    - L3 Bound
- Memory Bound
  - Ext. Memory Bound
  - MEM Bandwidth
  - MEM Latency

The Hierarchy – Intel Core™

A user-defined criteria for analyzing a hotspot: CPU Bound → Analyze

Frontend Bound
- Frontend Latency
- Bandwidth

Bad Speculation
- Branch Misspredicts
- Machine Clears

Retiring
- BASE
- Micro Sequencer

Backend Bound
- Core Bound
- Memory Bound

Memory Bound
- L1 Bound
- L2 Bound
- L3 Bound
- Ext. Memory Bound

Frontend Bound
- iTLB Miss
- Cache Misses
- Branch Retesters
- DSB switches
- MS Switches
- MITE
- DSB
- LSD

* Blue filled nodes denote Intel Core™ μarch-specific

Ahmad Yasin – Top-Down Analysis simplifies Optimization through VTune™ – Technion 2015
Example 1: Matrix Multiply

DEMO
**VTune “new General Exploration” interface**

Hover to see Metric description + formula of PMU events, or click arrow to expand a column to see a breakdown of issues pertaining to that category.
“Follow the pink”

Expand each column marked pink until you come to the actual issue.

Here is the problem, read tooltip to learn more.

Pick the actual HW event from the formula \((\text{LD BLOCKS. STORE FORWARD})\) – typically counts bigger-size loads blocked by smaller stores to the same address) for further detailed analysis.

Ack: Stas Bratanov
### Un-tuned

#### General Exploration

**Grouping:** Function / Call Stack

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Hardware Event C...</th>
<th>Hardware Event...</th>
<th>Filled Pipeline Slots</th>
<th>Unfilled Pipeline Slots (Stalls)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>multiply1</strong></td>
<td>CPU_CLK_UNHALTED_THREAD</td>
<td>INST_RETIRED_ANY</td>
<td>488,292,732,438</td>
<td>0.974</td>
</tr>
<tr>
<td><strong>KeWaitForMultipleObjects</strong></td>
<td></td>
<td></td>
<td>86,000,129</td>
<td>0.244</td>
</tr>
<tr>
<td><strong>KeSetTimer</strong></td>
<td></td>
<td></td>
<td>86,000,129</td>
<td>0.244</td>
</tr>
</tbody>
</table>

The CPI Rate for **multiply1** is 11.329, indicating a significant area for potential optimization.

#### General Exploration

**Grouping:** Function / Call Stack

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Retiring</th>
<th>Bad Speculation</th>
<th>Memory Bound</th>
<th>L1 Bound</th>
<th>L2 Bound</th>
<th>L3 Bound</th>
<th>DRAM Bound</th>
<th>Store Bound</th>
<th>Core Bound</th>
<th>Front-end Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>multiply1</strong></td>
<td>0.022</td>
<td>0.001</td>
<td>0.070</td>
<td>0.023</td>
<td>0.064</td>
<td>0.745</td>
<td>0.036</td>
<td>0.022</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td><strong>KeWaitForMultipleObjects</strong></td>
<td>0.081</td>
<td>0.244</td>
<td>0.000</td>
<td>0.326</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.244</td>
</tr>
</tbody>
</table>
Loop Interchange

void matrix_multiply ()
{
    // Multiply the two matrices

    for (int i = 0 ; i < ROWS ; i++) {
        for (int j = 0 ; j < COLUMNS ; j++) {
            for (int k = 0 ; k < COLUMNS ; k++) {
                matrix_r[i][j] = matrix_r[i][j] + matrix_a[i][k] * matrix_b[k][j];
            }
        }
    }
}
### Loop Interchange

#### General Exploration

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Hardware Event</th>
<th>Hardware Event</th>
<th>Filled Pipeline Slots</th>
<th>Unfilled Pipeline Slots (Stalls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply2</td>
<td>CPU_CL... THREAD</td>
<td>INST RETIRE ANY</td>
<td>0.852</td>
<td>0.353</td>
</tr>
<tr>
<td>KeSetTimer</td>
<td>24,000,030</td>
<td>0</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>init_arr</td>
<td>20,000,030</td>
<td>16,000,024</td>
<td>1.250</td>
<td>0.000</td>
</tr>
<tr>
<td>KeSynchronizeExecution</td>
<td>18,000,027</td>
<td>0</td>
<td>0.000</td>
<td>0.389</td>
</tr>
<tr>
<td>FxReleaseRundownPoint</td>
<td>14,000,021</td>
<td>6,000,009</td>
<td>2.333</td>
<td>0.000</td>
</tr>
</tbody>
</table>

#### Unfilled Pipeline Slots (Stalls)

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Memory Bound</th>
<th>Back-end Bound</th>
<th>Core Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Bo.</td>
<td>L2 Bo.</td>
<td>L3 Bo.</td>
</tr>
<tr>
<td>multiply2</td>
<td>0.060</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>KeSetTimer</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>init_arr</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>KeSynchronizeExecution</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>FxReleaseRundownPoint</td>
<td>0.060</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
Example 2: Software prefetching

Original Code

Tuned (1.35x speedup)

Prefetching can help Memory Latency Bound Apps. Use Carefully
Example 3: Microarchitecture comparison

- Haswell (4\textsuperscript{th} Core gen) has improved front-end
  - Speculative iTLB and cache accesses with better timing to improve the benefits of prefetching
- Benefiting benchmarks clearly show reduction in Frontend Bound

Using Top Down, forward compatibility is assured on Intel Core™
Example 4: Full workload Analysis


- Performance is limited due managing the data
  - Measured 65% speedup through Proof-by-optimization approach
Summary

- **Top Down Analysis**
  - An effective method to identify the true bottleneck
  - Uses microarchitecture-independent metrics

- **Integrated into VTune™, Linux perf wrapper, and other tools**

- **Forward compatibility on Intel Core™ and coming processors**

Try it out and share your feedback