Intended Audience: Software Developers

• Interested in performance optimizing your application
  > Don't need to be a performance expert
  > But should be an expert in the application!

• Working on a platform with an Intel® Xeon® E5 or E7 v2 Family processor

• Using Intel® VTune™ Amplifier XE performance analyzer
  > The performance information here applies to other tools (PTU, etc) but is focused on VTune Amplifier XE
How to Use this Presentation

• Read through the slides once, then again while collecting data
• Remember performance analysis is a process that may take several iterations
• Software Optimization should begin *after you have*:
  > Utilized any compiler optimization options (/O2, /QxSSE4.2, etc)
  > Chosen an appropriate workload
  > Measured baseline performance
Using Intel® VTune™ Amplifier XE to Tune Software on the Intel® Xeon® Processor E5/E7 v2 Family

Software and Services Group

Ver. 1.0
Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESToppel OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT, EXCEPT AS PROVIDED IN INTEL® TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS. INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO THE USE OF THIS DOCUMENT AND OR INTEL SPECIFICATIONS, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED OR INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must rely on their own knowledge and experience to determine the appropriate application for Intel products.

Covers of documents which have an order number and are referenced in this document or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website.

Intel® Hyper-Threading Technology requires a computer system with a processor supporting HT Technology and an HT Technology-enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information including details on which processors support HT Technology, see: http://www.intel.com/technology/ht

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and, for some uses, certain operating systems, software, and/or service enabled for virtualization. Performance may vary depending on the specific hardware and service you use and may require a hardware capable of virtualization. Intel’s software products do not guarantee any level of performance or security. See: http://www.intel.com/technology/vt

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers, and/or other software enabled for 64-bit computing. Consult your system vendor for more information:

Intel® Turbo Boost Technology requires a PC with a processor with Intel Turbo Boost Technology capability. Intel Turbo Boost Technology performance varies depending on hardware, software and operating system configuration. See: http://www.intel.com/technology/turbo

Intel, the Intel logo, Xeon, Xeon Inside, VTune, Core, and Core i7, Core i5, Core i3, Core i2 and Core are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

“Other names and brands are the property of their respective owners.

Copyright © 2014, Intel Corporation
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
Agenda

- The Intel® Xeon® Processor E5 v2 Family
- The new Intel® VTune™ Amplifier XE
- The Software Optimization Cycle
  > Hotspots
  > Methods for Determining Efficiency
  > Common Architectural Causes of Inefficiency:
    - Cache Misses
    - Other Data Access Issues
    - Execution Stalls
    - Branch Mispredicts
    - Front End Stalls
Intel® VTune™ Amplifier XE

VTune Amplifier XE features:
- Multiple Collection Types
  - Hotspots (statistical call tree)
  - Thread Concurrency
  - Locks and Waits Analysis
  - Event-based Sampling
- Timeline View Integrated into all Analysis Types
- Source/Assembly Viewing
- Compatible with C/C++, Fortran, Java, Assembly, .NET
- Visual Studio Integration, Command-line, or Standalone interface for Windows* or Linux*
The logic for identifying issues on Intel Microarchitecture Codename Ivy Bridge is embedded into the interface. All the formulas and metrics used are the same as the ones given in this guide. You no longer have to apply formulas and rules to the data yourself to figure out what it means – using this guide and the interface tuning features, you can easily pinpoint problems and possible solutions. The formulas and metrics are only applied to the General Exploration profile, and the General Exploration viewpoint (which is automatic). For the other profiles, it will just show the raw data.
The enhanced view is present when running the General Exploration profile with the General Exploration viewpoint selected (the default).

All collected data is presented in hierarchical format (see next slide), with helpful metrics already calculated (see issue slides).
Hierarchical data display corresponds to how available execution slots in each core’s pipeline are utilized.

Expand a column to see a breakdown of issues pertaining to its category of pipeline utilization: Retiring, Bad Speculation, Back-end Bound, or Front-end Bound Pipeline Slots.
Note that issue highlighting occurs under 2 conditions:

1. The value for the metric is over Intel VTune Amplifier XE’s pre-determined threshold
2. The associated function uses 5% or greater of the CPU clockticks sampled
Both Intel® Hyper-Threading Technology and Intel® Turbo Boost 2.0 Technology can be enabled or disabled through BIOS on most platforms. Contact with the system vendor or manufacturer for the specifics of any platform before attempting this. Incorrectly modifying bios settings from those supplied by the manufacturer can result in rendering the system completely unusable and may void the warranty.

Don’t forget to re-enable these features once you are through with the software optimization process!
Note: While VTune Amplifier XE’s Concurrency, Timeline and Locks and Waits features can also be helpful in threading an application, this slideset is not aimed at the process of introducing threads. The process described here could be used either before or after threading. However, we *do* recommend that you follow a top-down process when optimizing: beginning with system tuning (if appropriate), then algorithmic tuning, then microarchitectural tuning. The name of Software on Hardware tuning just means we are tuning software for specific hardware. Remember for all upcoming slides – that you should only focus on hotspots! Only try to determine efficiency, identify causes, and optimize in hotspots!
The “Software on Hardware” Tuning Process

For each Hotspot

- Determine efficiency
  > If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat
For this processor family, the CPU_CLK_UNHALTED.THREAD counter measures unhalted clockticks on a per thread basis. So for each tick of the CPU's clock, the counter will count 2 ticks if Hyper-Threading is enabled, 1 tick if Hyper-Threading is disabled. There is no per-core clocktick counter. There is also a CPU_CLK_UNHALTED.REF counter, which counts unhalted clockticks per thread, at the reference frequency for the CPU. In other words, the CPU_CLK_UNHALTED.REF counter should not increase or decrease as a result of frequency changes due to Turbo Mode 2.0 or Speedstep Technology.
The “Software on Hardware” Tuning Process

For each Hotspot

- Determine efficiency
  - If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat
Formula:
(UOPS_RETIRED.RETIRE_SLOTS/ (4*CPU_CLK_UNHALTED.THREAD))

Thresholds: Investigate if:
% Retiring < .75

This metric is based on the fact that when operating at peak performance, the pipeline on this CPU should be able to retire 4 micro-operations per clock cycle (or “clocktick”). The formula looks at “slots” in the pipeline for each core, and sees if the slots are filled, and if so, whether they contained a micro-op that retired.

The thresholds are general guidelines. Depending on the domain, some applications can run with less slots retiring than the thresholds above and still be efficient. It is important to remember that the pipeline slot methodology is only looking at how a given code utilizes the pipeline on a core. There still may be algorithmic optimizations, like parallelism and vectorization, that would boost the application’s performance.
Formula:

\[
\frac{\text{CPU_CLK_UNHALTED.THREAD}}{\text{INST RETIRED.ANY}}
\]

Threshold:
In the interface, CPI will be highlighted if > 1. This is a very general rule based on the fact that some well-tuned apps achieve CPIs of 1 or below. However, many apps will naturally have a CPI of over 1—it is very dependent on workload and platform. It is best used as a comparison factor—know your app’s CPI and see if over time it is moving upward (that is bad) or reducing (good!).

Note that CPI is a ratio! Cycles per instruction. So if the code size changes for a binary, CPI will change. In general, if CPI reduces as a result of optimizations, that is good, and if it increases, that is bad. However, there are exceptions! Some code can have a very low CPI but still be inefficient because more instructions are executed than are needed. This problem is discussed using the Code Examination method for determining efficiency.

Another Note: CPI will be doubled if using Intel® Hyper-threading. With Intel® Hyper-Threading enabled, there are actually 2 different definitions of CPI. We call them "Per Thread CPI" and "Per Core CPI". The Per Thread CPI will be twice the Per Core CPI. Only convert between per Thread and per Core CPI when viewing aggregate CPIs (summed for all logical threads).

Note: Optimized code (i.e.: SSE instructions) may actually lower the CPI, and increase stall %—but it will increase the performance. CPI is just a general efficiency metric—the real measure of efficiency is work taking less time.
This method involves looking at the disassembly to make sure the most efficient instruction streams are generated. This can be complex and can require an expert knowledge of the Intel instruction set and compiler technology. What we have done is describe how to find 3 easy-to-detect code patterns and suggest how they may be implemented more efficiently using new features of the CPU.
Efficiency Method 3, Code Study 1: Convert Legacy Floating Point or Integer Code to SIMD

- **Why:** Using SIMD instructions can greatly increase performance. For existing FP SSE code, converting to Intel® Advanced Vector Extensions (AVX) instructions has several advantages.

- **How:** Examine your assembly code for existing SSE instructions (using xmm registers), MMX instructions (using mmx registers), or for floating point instructions that are not packed (such as faddp, fmul, or scalar SSE instructions like addss)

- **What Now:**
  - Convert applicable instructions to AVX automatically
    - Intel Compiler: /QxAVX or /QaxAVX (Windows*), –axAVX or -axAVX (Linux)
    - GCC: -march='corei7-avx' (no dispatching for non-AVX systems)
    - MSVS: /arch:avx (no dispatching for non-AVX systems)


A few notes:
- Integer vector code is not supported by AVX in the first generation, but integer code can still use 128-bit SSE instructions.
- The /Qax and –ax versions of the Intel Compiler switches support automatic dispatching, to create a AV-optimized codepath for appropriate systems and a default codepath otherwise.

- **Why:** Intel AES-NI can accelerate the execution of AES algorithms by implementing some of the functionality in hardware
- **How:** If the application’s implements AES algorithms, check to see if Intel AES-NI instructions are being used. Blocks of aes instructions with xmm registers as operands may be using parallel modes:
  
  ```
  aesenc %xmm5, %xmm7
  aesenc %xmm5, %xmm8
  aesenc %xmm5, %xmm9
  aesenc %xmm5, %xmm10
  ```

- **What Now:**
  
  - If AES-NI are being used in parallel modes (such as ECB, CTR, and CBC-Decrypt), increase performance by redefining the number of blocks to be processed in parallel (8 on Haswell).
The “Software on Hardware” Tuning Process

For each Hotspot

- Determine efficiency
  > If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat
For a hotspot that is inefficient, determining the primary bottleneck is the first step. Optimizing code to fix issues outside the primary bottleneck category may not boost performance – the biggest boost will come from resolving the biggest bottleneck. Generally, if Retiring is the primary bottleneck, that is good. See next slides.
Note that the way this methodology allows us to classify what percentage of all pipeline slots end up in each category, for each cycle and for each core. It is possible that for a given dataset, there may be a significant percentage of pipeline slots in multiple categories that merit investigation. Start with the category with the highest percentage of pipeline slots. Ideally a large percentage of slots will fall into the “Retiring” category, but even then, it may be possible to make your code more efficient.

For a complete description of this methodology, see the Intel® 64 and IA-32 Architectures Optimization Reference Manual, Appendix B.3.
The distribution of pipeline slots in these four categories is very useful for developers. Although metrics based on events have been possible for many years, before this characterization there was no approach for identifying which possible performance issues were the most impactful. When performance metrics are placed into this framework, a developer can see which issues need to be tackled first. Within VTune Amplifier XE, if an issue is highlighted, it doesn’t need to be investigated unless it is from the primary bottleneck category. Fixing issues that are not within the primary bottleneck category may not improve performance.
The “Software on Hardware” Tuning Process

For each Hotspot

– Determine efficiency

> If inefficient:

• Determine primary bottleneck

• Identify architectural reason for inefficiency

• Optimize the issue

Repeat
Identifying Architectural Reasons for Inefficiency: The Issue Slides

- Issues are listed by category, and each category is explained.
- For each potential issue, there are several important pieces of information:
  - Why? – Why you should be concerned about this potential problem.
  - How? – Which profile and metric to use in the Amplifier XE interface. If the data is highlighted, then it should be investigated.
  - Event Names and Metric Formulas are given in the Notes. These are not included on the slide because they are already embedded in the Amplifier XE logic. You only need to use the pre-configured profiles and metrics pointed out in order to know if you may have a problem.
Tuning for the Front-End Bound Category

The Front-End of the pipeline
- Fetches instructions
- Decodes instructions into micro-operations
- Delivers up to 4 micro-operations per cycle to the Back-End
Front-End issues are caused by delays in fetching code (due to caching or ITLB issues) or in decoding instructions (due to specific instruction types or queuing issues). Front-End issues are generally resolved by compiler techniques like code layout (co-locating hot code), reducing code footprint, and Profile-Guided Optimization (PGO).
Front-End Hierarchical Breakdown in Amplifier XE

Expand Front-end Bound to see the percentage of the Front-end Bound cycles classified as “Front-End Latency”, where no micro-ops were being delivered vs. “Front-End Bandwidth”, where <4 micro-ops were being delivered.

If Front-end Bound is the primary bottleneck, concentrate on Front-End Latency. Resolve highlighted issues under this category.
Formulas:
% of cycles spent on ITLB Misses (ITLB Overhead):
\[ \frac{ITLB_{\text{MISSES.WALK\_DURATION}}}{CPU\_CLK\_UNHALTED.THREAD} \]

% of cycles spent on ICache Misses:
\[ \frac{(ICACHE\_IFETCH\_STALL - ITLB_{\text{MISSES.WALK\_DURATION})}{CPU\_CLK\_UNHALTED.THREAD} \]

% of cycles due to LCP stalls:
\[ \frac{ILD\_STALL\_LCP/CPU\_CLK\_UNHALTED.THREAD} \]

Thresholds:
Thresholds: Investigate if –
% of cycles spent on ITLB Misses (ITLB Overhead) ≥ .05 (5%)
% of cycles spent on ICache Misses ≥ .05 (5%)
% of cycles due to LCP stalls ≥ .05 (5%)
Tuning for the Back-End Bound Category

The Back-End of the pipeline
• Accepts micro-operations from the Front-End
• Re-orders them as necessary to schedule their execution in execution units
• Retrieves needed operands from memory
• Executes the operations
• Commits results to memory
The back-end is the most common category for hotspots, and is most likely to be the primary bottleneck.
Back-End Hierarchical Breakdown in Amplifier XE

Expand Back-End Bound to see metrics in the Back-End classified as “Memory Bound”, where the back-end could not accept new micro-ops due to too many outstanding memory operations, vs “Core Bound”, where the issue is saturation execution ports.
Cache Misses

**Why:** Cache misses raise the CPI of an application
- Focus on long-latency data accesses coming from 2nd and 3rd level misses

**How:** Memory Bound sub-category, Metrics: *L3 Latency, Local DRAM*

**What Now:**
- If either metric is highlighted for your hotspot, consider reducing misses:
  - Change your algorithm to reduce data storage
  - Block data accesses to fit into cache
  - Check for sharing issues (See Contested Accesses)
  - Align data for vectorization (and tell your compiler)
  - Use the cacheline replacement analysis outlined in section B.3.4.2 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, section B.3.4.2
  - Use streaming stores
  - Use software prefetch instructions

Formulas:

% of cycles spent on memory access (Local DRAM):
\[
\frac{(MEM\_LOAD\_UOPS\_LLC\_MISS\_RETIRED.\_LOCAL\_DRAM \times 130)}{CPU\_CLK\_UNHALTED.\_THREAD}
\]

L3 Latency
% of cycles spent on L3 Latency:
\[
\frac{(MEM\_LOAD\_UOPS\_RETIRED.LLC\_HIT\_PS \times 29)}{CPU\_CLK\_UNHALTED.\_THREAD}
\]

Thresholds: Investigate if
- % cycles for Local DRAM \( \geq .2 \)
- % cycles for L3 Latency \( \geq .2 \)
Remote Memory Accesses

- **Why:** With a Non-Uniform Memory Access (NUMA) architecture, loading from memory can have varying latency. Remote memory loads cost more.
- **How:** General Exploration profile, Metrics: Remote DRAM
- **What Now:**
  - If this metric is highlighted for your hotspot, improve NUMA affinity:
    - Ensure that memory is first touched (accessed, not allocated) by the thread that will be using it.
    - If thread migration is a problem, try affinitizing or pinning threads to cores.
    - Use affinity environment variable for OpenMP.
    - Use NUMA-aware options for supporting applications if available (for example, softnuma for SQL Server).
    - Use a NUMA-efficient thread scheduler (such as Intel® Threading Building Blocks or Intel® Cilk™ Plus).

Formulas:

% of cycles spent on remote memory access (Remote DRAM):

\[
\frac{(MEM\_LOAD\_UOPS\_LLC\_MISS\_RETIRRED.REMOTE\_DRAM \times 310)}{CPU\_CLK\_UNHALTED.THREAD}
\]

Thresholds: Investigate if –

% cycles for remote access ≥ .2

Malloc() or VirtualAlloc() is not touching memory. The o/s only reserves a virtual address for the request. Physical memory is not allocated until the address is accessed. Each 4K page will be physically allocated on the node where the thread makes the first reference. Note that this metric measures only remote memory (DRAM) accesses, and does not include data found in cache in the remote socket.
Contested Accesses

- **Why:** Sharing modified data among cores (at L2 level) can raise the latency of data access
- **How:** Memory Bound sub-category, Metric: Contested Accesses
- **What Now:**
  - If this metric is highlighted for your hotspot, locate the source code line(s) that is generating HITMs by viewing the source. Look for the MEM_LOAD_UOPS_LLCC_HIT RETIRED.XSNP_HITM_PS event which will tag to the next instruction after the one that generated the HITM.
  - Then use knowledge of the code to determine if real or false sharing is taking place. Make appropriate fixes:
    - For real sharing, reduce sharing requirements
    - For false sharing, pad variables to cacheline boundaries

**Formula:**
% of cycles spent accessing data modified by another core:

\[
\text{(MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRED.XSNP\_HITM\_PS * 60) / CPU\_CLK\_UNHALTED.THREAD}
\]

**Thresholds:** Investigate if –
% cycles accessing modified data ≥ .05

This metric is also called write sharing. It occurs when one core needs data that is found in a modified state in another core’s cache. This causes the line to be invalidated in the holding core’s cache and moved to the requesting core’s cache. If it is written again and another core requests it, the process starts again. The cacheline ping pong-ing between caches causes longer access time than if it could be simply shared amongst cores (as with read-sharing).

Write sharing can be caused by true sharing, as with a lock or hot shared data structure, or by false sharing, meaning that the cores are modifying 2 separate pieces of data stored on the same cacheline. This metric measures write sharing at the L2 level only – that is, within one socket. If write sharing is observed at this level it is possible it is occurring across sockets as well.

Note that in the case of real write sharing that is caused by a lock, Amplifier XE’s Locks and Waits analysis should also indicate a problem. This hardware-level analysis will detect other cases as well though (such as false sharing or write sharing a hot data structure).
**Data Sharing**

- **Why:** Sharing clean data (read sharing) among cores (at L2 level) has a penalty at least the first time due to coherency
- **How:** Memory Bound sub-category, Metrics: *Data Sharing*
- **What Now:**
  - If this metric is highlighted for your hotspot, locate the source code line(s) that is generating HITs by viewing the source. Look for the MEM_LOAD_UOPS LLC_HIT RETIRED/XSNP_HIT PS event which will tag to the next instruction after the one that generated the HIT.
  - Then use knowledge of the code to determine if real or false sharing is taking place. Make appropriate fixes:
    - For real sharing, reduce sharing requirements
    - For false sharing, pad variables to cacheline boundaries

---

**Formula:**

\[
\frac{(MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRED.XSNP\_HIT\_PS \times 43)}{CPU\_CLK\_UNHALTED.THR\_EAD}
\]

**Thresholds:** Investigate if –

% cycles accessing clean shared data ≥ .05

This metric measures read sharing, or sharing of “clean” data, across L2 caches within 1 CPU socket. The L3 cache has a set of “core valid” bits that indicate whether each cacheline could be found in any L2 caches on the same socket, and if so, which ones. The first time a line is brought into the L3 cache, it will have core valid bits set to 1 for whichever L2 cache it went into. If that line is then read by a different core, then it will be fetched from L3, where the core valid bits will indicate it is present in one other core. The other L2 will have to be snooped, resulting in a longer latency access for that line. This metric measures the impact of that additional access time, when the cacheline in question is only being read-shared. In the case of read-sharing, the line can co-exist in multiple L2 caches in shared state, and for future accesses more than one core valid bit will be set. Then when other cores request the line, no L2 caches will need to be snooped, because the presence of 2 or more core valid bits tells the LLC that the line is shared (for reading) and ok to serve. Thus the impact of this only happens the first time a cacheline is requested for reading by a second L2 after it has already been placed in the L3 cache. The impact of sharing modified data across L2s is different and is measured with the “Contested Accesses” metric.
Other Data Access Issues: Blocked Loads Due to No Store Forwarding

- **Why:** If it is not possible to forward the result of a store through the pipeline, dependent loads may be blocked
- **How:** Memory Bound sub-category, Metric: *Loads Blocked by Store Forwarding*
- **What Now:**
  - If the metric is highlighted for your hotspot, investigate:
  - View source and look at the LD_BLOCKS.STORE_FORWARD event. Usually this event tags to next instruction after the attempted load that was blocked. Locate the load, then try to find the store that cannot forward, which is usually within the prior 10-15 instructions. The most common case is that the store is to a smaller memory space than the load. Fix the store by storing to the same size or larger space as the ensuing load.

**Formula:**
Blocked Store Forwarding Cost = \[
\frac{(\text{LD\_BLOCKS\_STORE\_FORWARD} \times 13)}{\text{CPU\_CLK\_UNHALTED\_THREAD}}
\]

**Threshold:** Investigate if –
Cost ≥ .05

Store forwarding occurs when there are two memory instructions in the pipeline, a store followed by a load from the same address. Instead of waiting for the data to be stored to the cache, it is “forwarded” back along the pipeline to the load instruction, saving a load from the cache. Store forwarding is the desired behavior, however, in certain cases, the store may not be able to be forwarded, so the load instruction becomes blocked waiting for the store to write to the cache and then to load it.
Formula:
Aliasing Conflicts Cost = (LD_BLOCKS_PARTIAL.ADDRESS_ALIAS * 5) / CPU_CLK_UNHALTED.THREAD

Threshold: Investigate if
Aliasing conflicts cost ≥ .1

This occurs when a load is issued after a store and their memory addresses are offset by (4K). When this is processed in the pipeline, the issue of the load will match the previous store (the full address is not used at this point), so pipeline will try to forward the results of the store and avoid doing the load (this is store forwarding). Later on when the address of the load is fully resolved, it will not match the store, and so the load will have to be re-issued from a later point in the pipe. This has a 5-cycle penalty in the normal case, but could be worse in certain situations, like with un-aligned loads that span 2 cache lines.
Other Data Access Issues: DTLB Misses

- **Why:** First-level DTLB Load misses (Hits in the STLB) incur a latency penalty. Second-level misses require a page walk that can affect your application’s performance.
- **How:** Memory Bound sub-category, Metric: DTLB Overhead, DTLB Store Overhead
- **What Now:**
  - If this metric is highlighted for your hotspot, investigate at the source code level.
  - To fix these issues, target data locality to TLB size, use the Extended Page Tables (EPT) on virtualized systems, try large pages (database/server apps only), increase data locality by using better memory allocation or Profile-Guided Optimization

Formula:
DTLB Overhead = ((DTLB_LOAD_MISSES.STLB_HIT * 7) + DTLB_LOAD_MISSES.WALK_DURATION) / CPU_CLK_UNHALTED.THREAD
DTLB Store Overhead = ((DTLB_STORE_MISSES.STLB_HIT * 7) + DTLB_STORE_MISSES.WALK_DURATION) / CPU_CLK_UNHALTED.THREAD

Threshold: Investigate if-
DTLB Overhead ≥ .1

On target data locality to TLB size: this is accomplished via data blocking and trying to minimize random access patterns.
Note: this is more likely to occur with server applications or applications with a large random dataset
Max theoretical bandwidth, per socket, for processor with DDR 1600 and 4 memory channels: 51.2 GB/s
View both total and read memory bandwidth per socket, over time.
Tuning for the Bad Speculation Category

Speculation is when:
- A micro-operation is allowed to execute, before it is known whether that operation will retire
- Allows for greater Instruction-Level Parallelism in an out-of-order pipeline
Micro-operations that are removed from the Back-End most likely happen because the Front-End mis-predicted a branch. This is discovered in the Back-End when the branch operation is executed. At this point, if the target of the branch was incorrectly predicted, the micro-operation and all subsequent incorrectly predicted operations are removed and the Front-End is re-directed to begin fetching instructions from the correct target.
Branch Mispredicts

- **Why:** Mispredicted branches cause pipeline inefficiencies due to wasted work or instruction starvation (while waiting for new instructions to be fetched)
- **How:** General Exploration Profile, Metric: Branch Mispredict
- **What Now:**
  - If this metric is highlighted for your hotspot try to reduce misprediction impact:
  - Use compiler options or profile-guided optimization (PGO) to improve code generation
  - Apply hand-tuning by doing things like hoisting the most popular targets in branch statements.

**Formula:**
Mispredicted branch cost: \( \frac{BR\_MISP\_RETIRED.ALL\_BRANCHES\_PS}{(BR\_MISP\_RETIRED.ALL\_BRANCHES\_PS + MACHINE\_CLEARS.COUNT) \times \left( (UOPS\_ISSUED.ANY - UOPS\_RETIRED.RETIRE\_SLOTS + 4 \times INT\_MISC.RECOVERY\_CYCLES) \right)} \) / \( (CPU\_CLK\_UNHALTED.THREAD \times 4) \)

**Threshold:** Investigate if -
Cost is ≥ .2

Note that all applications will have some branch mispredicts - it is not the number of mispredicts that is the problem but the impact. To do hand-tuning, you need to locate the branch causing the mispredicts. This can be difficult to track down due to the fact that this event will normally tag to the first instruction in the correct path that the branch takes.
Threshold: Investigate if -
Cost is ≥ .02

Machine clears are generally caused by either contention on a lock, or failed memory disambiguation from 4k aliasing (both earlier issues). The other potential cause is self-modifying code (SMC).
Tuning for the Retiring Category

Retirement is:
• The completion of a micro-op’s execution
• If the micro-op is the last micro-op for an instruction, it is also the completion of an instruction’s execution
• When results of an instruction’s execution are committed to the architectural state (cache, memory, etc)
In general, having as many pipeline slots retiring per cycle as possible is the goal. Besides algorithmic improvements like parallelism, there are two potential areas to investigate for the retiring category. The first is whether vectorization can be applied to make the instructions that are retiring even more efficient. See Code Study 1 – slide 22 – for more on this. The second is whether microcode assists can be eliminated from the instruction stream. See next 2 slides.
Retiring Hierarchical Breakdown in Amplifier XE

Fixing performance issues will often increase the portion of uops classified as General Retirement.

Expand Retiring to see the percentage of the Retiring slots classified as "General Retirement", which is the best case, vs "Microcode Sequencer", where the micro-ops retired were generated from the microcode sequencer (see next slide).
Formula:
Assist % = IDQ.MS_CYCLES / (CPU_CLK_UNHALTED.THREAD * 4)

Threshold: Investigate if –
Assist Cost ≥ .2

There are many instructions that can cause assists when there is no performance problem. If you see MS_CYCLES it doesn’t necessarily mean there is an issue, but whenever you do see a significant amount of MS_CYCLES, check the other metrics to see if it’s one of the problems we mention.
The “Software on Hardware” Tuning Process

For each Hotspot

- Determine efficiency
  - If inefficient:
    - Determine primary bottleneck
    - Identify architectural reason for inefficiency
    - Optimize the issue

Repeat

Repeat until there are no significant hotspots or Retiring pipeline slots meet expectations
Good Luck!
For more information:

VTune Amplifier XE Videos, Forums, and Resources:

Intel® 64 and IA-32 Architecture Software Developer's Manuals:

VTune Amplifier XE Tuning Guides for Other microarchitectures:

For optimization of the integrated graphics controller:
www.intel.com/software/gpa