Software

INTEL® VTUNE™ AMPLIFIER FOR CACHE PERFORMANCE ANALYSIS

Jackson Marusarz – Intel Corporation

Intel® VTune™ Amplifier

Quick Introduction

Get the Data You Need

- Hotspot (Statistical call tree), Call counts (Statistical)
- Thread Profiling – Concurrency and Lock & Waits Analysis
- Cache miss, Bandwidth analysis…[1]
- GPU Offload and OpenCL™ Kernel Tracing

Find Answers Fast

- View Results on the Source / Assembly
- OpenMP Scalability Analysis, Graphical Frame Analysis
- Filter Out Extraneous Data – Organize Data with Viewpoints
- Visualize Thread & Task Activity on the Timeline

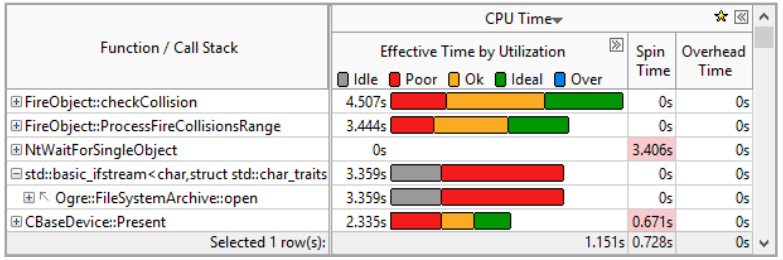
Easy to Use

- No Special Compiles – C, C++, C#, Fortran, Java, ASM
- Visual Studio* Integration or Stand Alone
- Graphical Interface & Command Line
- Local & Remote Data Collection
- Analyze Windows* & Linux* data on OS X*[2]

[1] Events vary by processor. [2] No data collection on OS X*

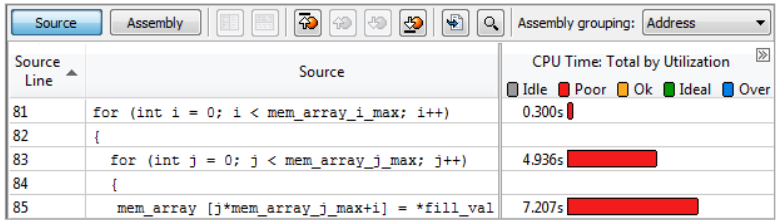
Full lecture April 19th

Quickly Find Tuning Opportunities

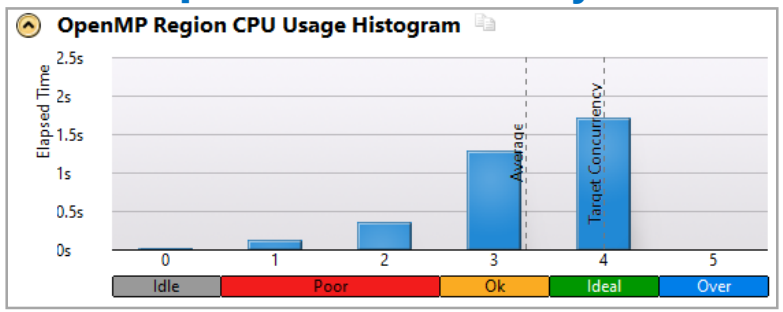


Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Overhead Time
FireObject::checkCollision	4.507s	0s	0s
FireObject::ProcessFireCollisionsRange	3.444s	0s	0s
NtWaitForSingleObject	0s	3.406s	0s
std::basic_ifstream<char,struct std::char_traits	3.359s	0s	0s
Ogre::FileSystemArchive::open	3.359s	0s	0s
CBaseDevice::Present	2.335s	0.671s	0s
Selected 1 row(s):	1.151s	0.728s	0s

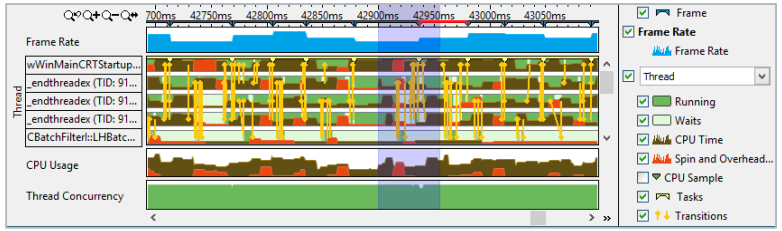
See Results On The Source Code



Tune OpenMP Scalability



Visualize & Filter Data



Performance Monitoring Unit (PMU)

- Hardware registers on any modern Intel processor
- 100s of events in current CPU generations
- Performance counters can be programmed to count Events through specific MSRs
- Events can be divided into the following categories, depending on how they are collected and interpreted:
  - **Fixed events**

    `CPU_CLK_UNHALTED.THREAD` – Cycles running at the rate of the core reflecting frequency changes (Turbo etc…)

    `INST_RETIRED.ANY` – Instructions retired

    `CPU_CLK_UNHALTED.REF` – Cycles at a non-changing reference rate
  - **Programmable events**
  - **Precise events**

Performance Monitoring Unit

Performance counters

Core performance monitoring of CPU of modern CPUs

- Each core has 8 counters; 4 per thread with SMT
- Measure 7 performance events at a time (4 Programmable, 3 Fixed)

Measure “Uncore” events in addition to “Core” events

- Distributed design with separate blocks of counters in different architectural units (MC, LLC, GT, etc.)
- Not thread-specific. Thread-specific counting can only be done in the core

The event names change for each processor generation, but the performance analysis concepts stay the same!

Event Based Performance Analysis

Processor events can be monitored using **sampling** and **counting** technologies

Sampling Mode:

- Sample - a HW interruption happens when a *N* of Events counted
- N is programmable
- In a sample we automatically collect:
  - Thread and process ID's
  - Instruction Pointer (IP)
- Instruction pointer is then used to derive the function name and source line number from the debug information created at compile time
- This creates a statistical representation of where the events are occurring

Counting Mode:

- Running counters for events without any interrupts or program information collected
- This is probably what you have done with PAPI

Sampling Mode vs. Counting Mode

Sampling Mode

- Identifies WHERE events occur
- Used for profiling and tuning
- Statistical representation
- Higher overhead – still low
- Can generate large amounts of data

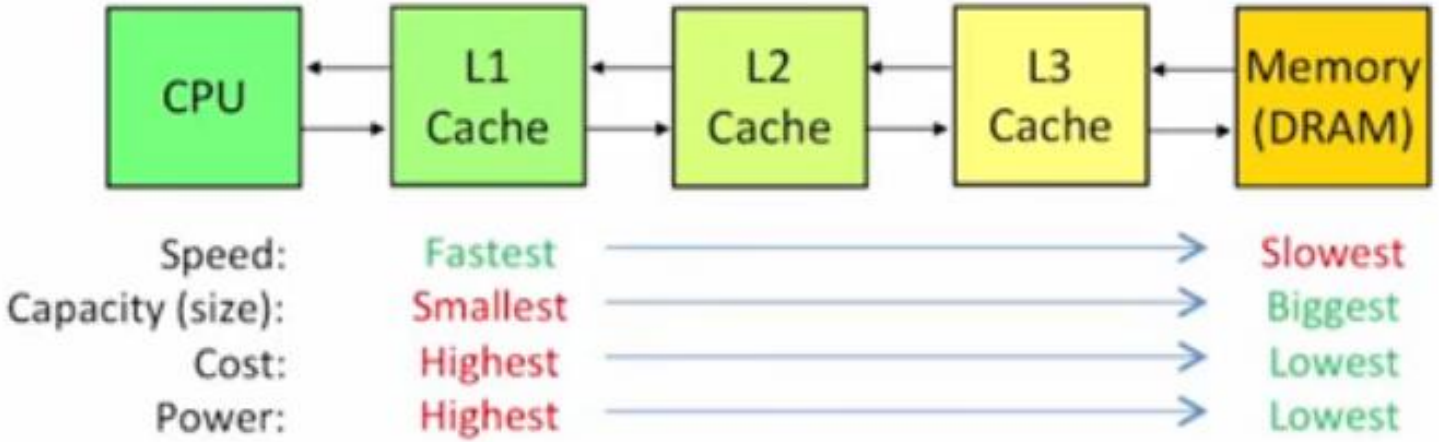
Counting Mode

- Reports how many events occur, not location.
- Mostly for profiling
- True event count
- Very low overhead
- Small data files

Both are available in VTune Amplifier – we will focus on Sampling

EBS for Memory Performance - Motivation

- Large memories are slow
- Small memories are fast, but expensive and consume high power
- **Goal:** give the processor a feeling that it has a memory which is fast, large, consumes low power and cheap
- **Solution:** a Hierarchy of memories



Memory Subsystem PMU Events

Examples:

```
MEM_INST_RETIRED.LOADS
MEM_LOAD_RETIRED.L1D_HIT
MEM_LOAD_RETIRED.L2_HIT
MEM_LOAD_RETIRED.LLC_UNSHARED_HIT
MEM_LOAD_RETIRED.LLC_MISS
```

What if I told you that you have 1,200,000 L2 Cache Hits?

Metrics

Create useful, quantitative data from raw event counts.

Examples:

- CPI = Total Cycles/Instructions
- L3 miss penalty = (L3 miss count * 200)/Total Cycles
- L2 Hit Ratio = L2 Hits/L2 Accesses

With expert knowledge –
thresholds and advice can be assigned to each metric

Two Ways to Measure Memory Impacts

Cycle Accounting

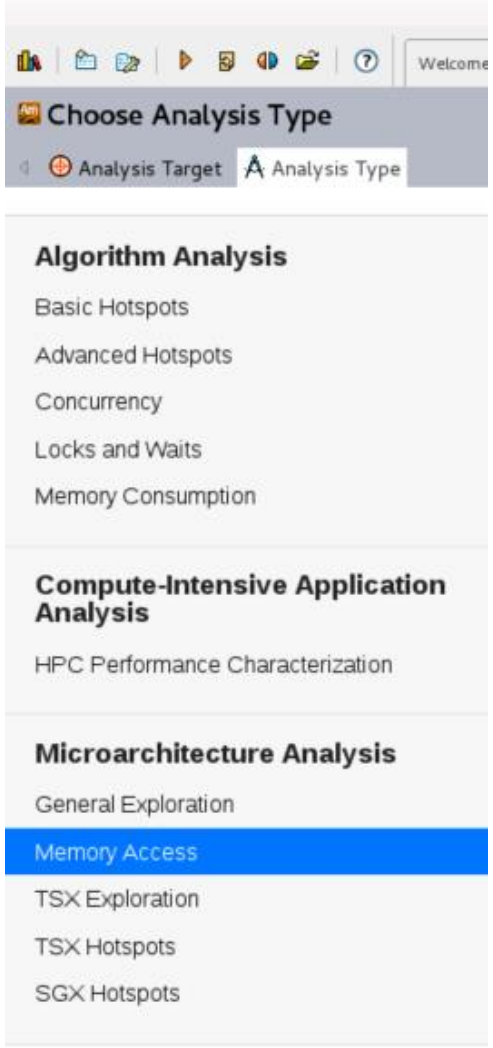
- Original technique based on estimated penalties for various events
- Example: L3 miss penalty = (L3 miss count * 200)/Total Cycles
- Main Issue – Superscalar, Out-of-Order architectures can hide memory latency issues by executing other instructions

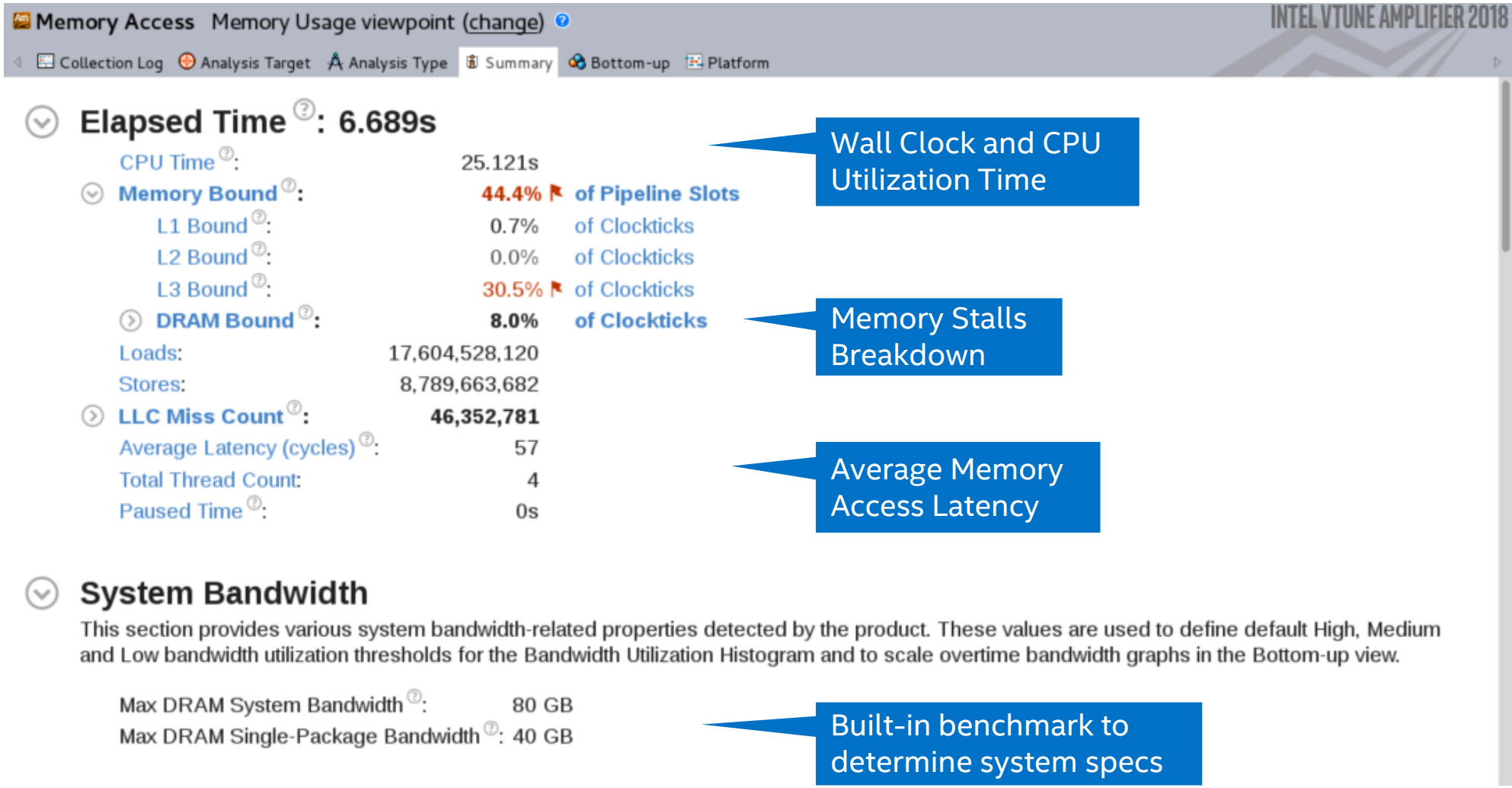
Stall Accounting

- Based on new PMU events determining when the CPU is actually stalled
- Example: CYCLE_ACTIVITY.STALLS_L3_MISS

Metrics used will be based on available events

Memory Analysis in VTune Amplifier





Wall Clock and CPU Utilization Time

Memory Stalls Breakdown

Average Memory Access Latency

Built-in benchmark to determine system specs

Memory Analysis in VTune Amplifier

Choose Analysis Type

Analysis Target | Analysis Type

Algorithm Analysis

- Basic Hotspots
- Advanced Hotspots
- Concurrency
- Locks and Waits
- Memory Consumption

Compute-Intensive Analysis

- HPC Performance Characterization

Microarchitecture Analysis

- General Exploration
- Memory Access
- TSX Exploration
- TSX Hotspots
- SGX Hotspots

Memory Access Memory Usage viewpoint (change) INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

Elapsed Time: 6.689s

Wall Clock and CPU

NOTE – Recent generations of hardware (Sandy Bridge+) have many more events available. Course lab machines are older.

This lecture will focus on recent generations, but your lab results will differ. I'll call out the differences when I can.

Paused Time: 0s

Access Latency

System Bandwidth

This section provides various system bandwidth-related properties detected by the product. These values are used to define default High, Medium and Low bandwidth utilization thresholds for the Bandwidth Utilization Histogram and to scale overtime bandwidth graphs in the Bottom-up view.

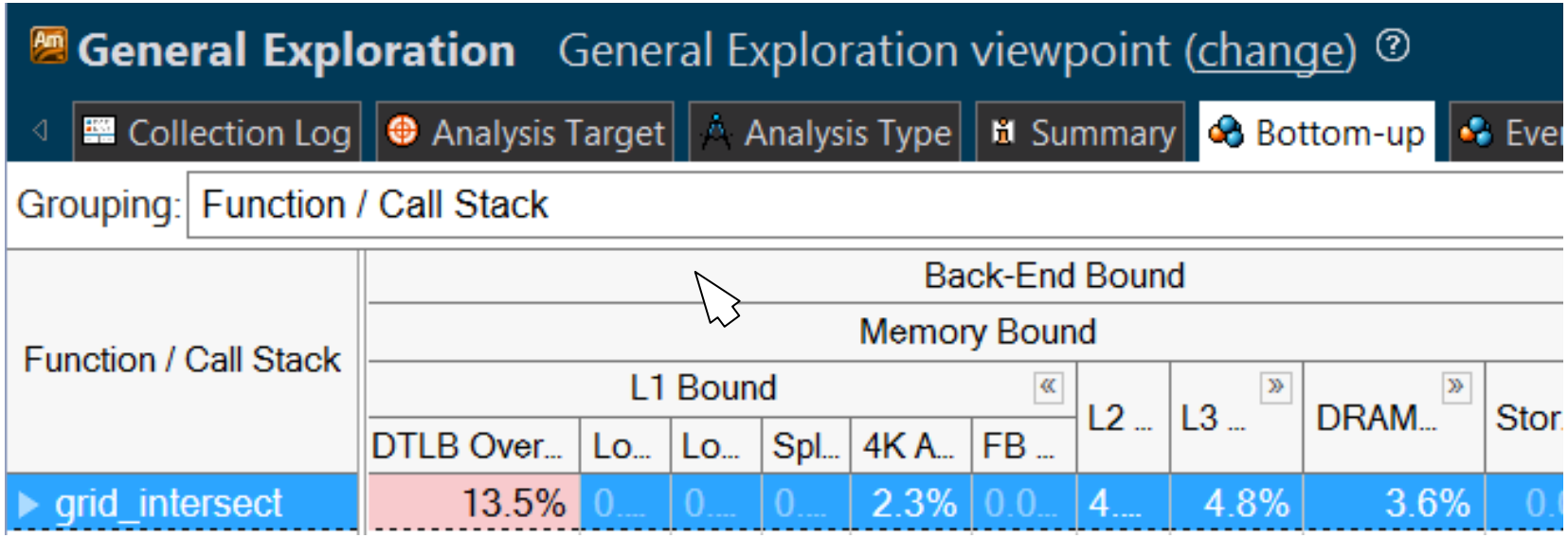
Max DRAM System Bandwidth: 80 GB

Max DRAM Single-Package Bandwidth: 40 GB

Built-in benchmark to determine system specs

General Exploration Analysis

- Intel® VTune™ Amplifier has hierarchical expanding metrics.
- You can expand your way down, following the hotspot, to identify the root cause of the inefficiency.
  - Sub-metrics highlight pink on their own merits, just like top level metrics.
- Hovering over a metric produces a helpful, detailed tooltip (not shown).
- There are tooltips on Summary tabs too: hover over any ⓘ icon.



Categorizing Inefficiencies in the Memory Subsystem

Back-End Bound													
Memory Bound												Core Bound	
L1 Bound	L2 Bound	L3 Bound				DRAM Bound		Store Bound				Divider	Port Utilization
		Contested Acc...	Data Sharing	L3 Latency	SQ Full	Memory Band...	Memory Lat... LLC Miss	Store Latency	False Shari...	Split Sto...	DTLB Store ...		
3.2%		0.0%	0.0%	0.0%	0.0%	0.2%	0.0%	3.3%	0.0%	0.0%	0.2%	0.0%	26.6%
11.3%	4.8%	0.0%	0.0%	100.0%	0.0%	9.5%	0.0%	1.1%	0.0%	0.2%	0.2%	4.8%	17.2%

- Back End bound is the most common bottleneck type for most applications.
- It can be split into Core Bound and Memory Bound
  - **Core Bound** includes issues like not using execution units effectively and performing too many divides.
  - **Memory Bound** involves cache misses, inefficient memory accesses, etc.
    - Store Bound is when load-store dependencies are slowing things down.
    - The other sub-categories involve caching issues and the like. Memory Access Analysis may provide additional information for resolving this performance bottleneck.

VTune Amplifier Workflow Example- Summary View

Memory Access Memory Usage viewpoint (change) INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

Elapsed Time: 6.689s

CPU Time:	25.121s	
Memory Bound:	**44.4%**	**of Pipeline Slots**
L1 Bound:	0.7%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	30.5%	of Clockticks
DRAM Bound:	**8.0%**	**of Clockticks**
Loads:	17,604,528,120	
Stores:	8,789,663,682	
LLC Miss Count:	**46,352,781**	
Average Latency (cycles):	57	
Total Thread Count:	4	
Paused Time:	0s	

High percentage of L3 Bound cycles

System Bandwidth

This section provides various system bandwidth-related properties detected by the product. These values are used to define default High, Medium and Low bandwidth utilization thresholds for the Bandwidth Utilization Histogram and to scale overtime bandwidth graphs in the Bottom-up view.

Max DRAM System Bandwidth:	80 GB
Max DRAM Single-Package Bandwidth:	40 GB

VTune Amplifier Workflow Example- Bottom-Up View

Over-Time DRAM Bandwidth

Over-Time QPI/UPI Bandwidth

Grid Breakdown by Function (configurable)

Memory Access Memory Usage viewpoint (change)

INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

0s 1s 2s 3s 4s 5s 6s

DRAM Bandwidth, GB/sec
- package_0 40.000
- package_1 40.000

dwidth, GB/sec
- package_1 8.728
- package_0 8.728

CPU Time
- package_1 5600.0%
- package_0 5600.0%

DRAM Bandwidth, GB/sec
- Total, GB/sec
- Read, GB/sec
- Write, GB/sec

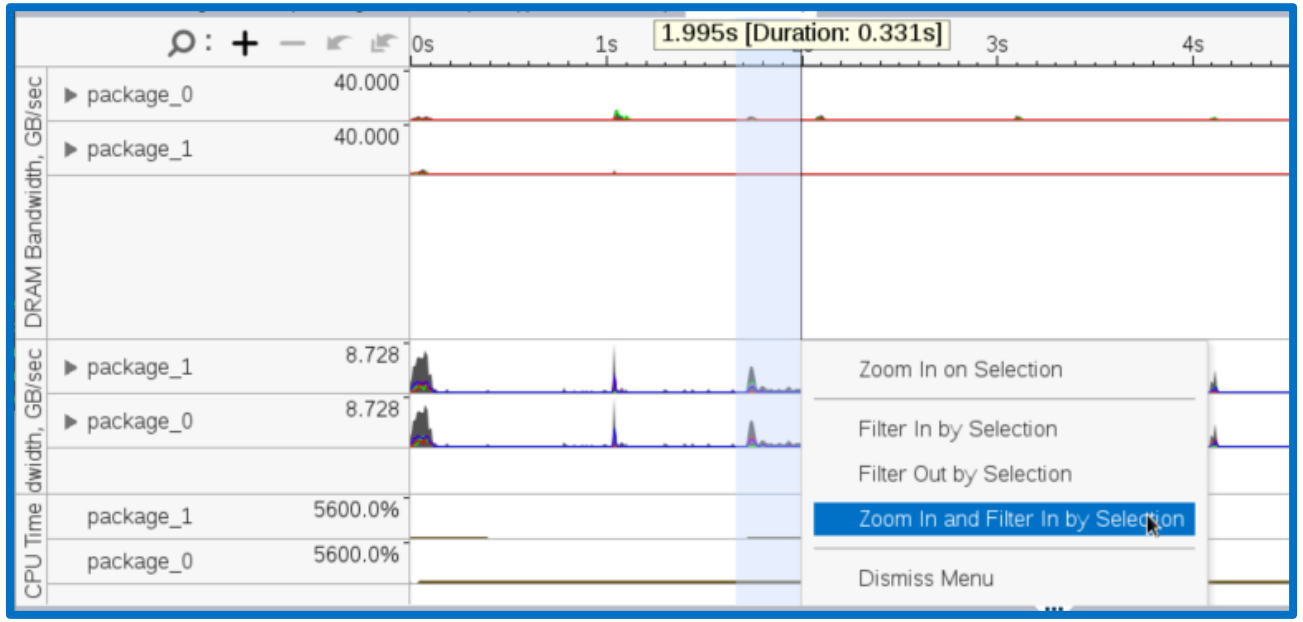
UPI Bandwidth, GB/sec
- Total, GB/sec
- Incoming Data, GB/sec
- Incoming Non-Data, ...
- Outgoing Data, GB/sec
- Outgoing Non-Data, ...

CPU Time
- CPU Time

Grouping: Function / Call Stack

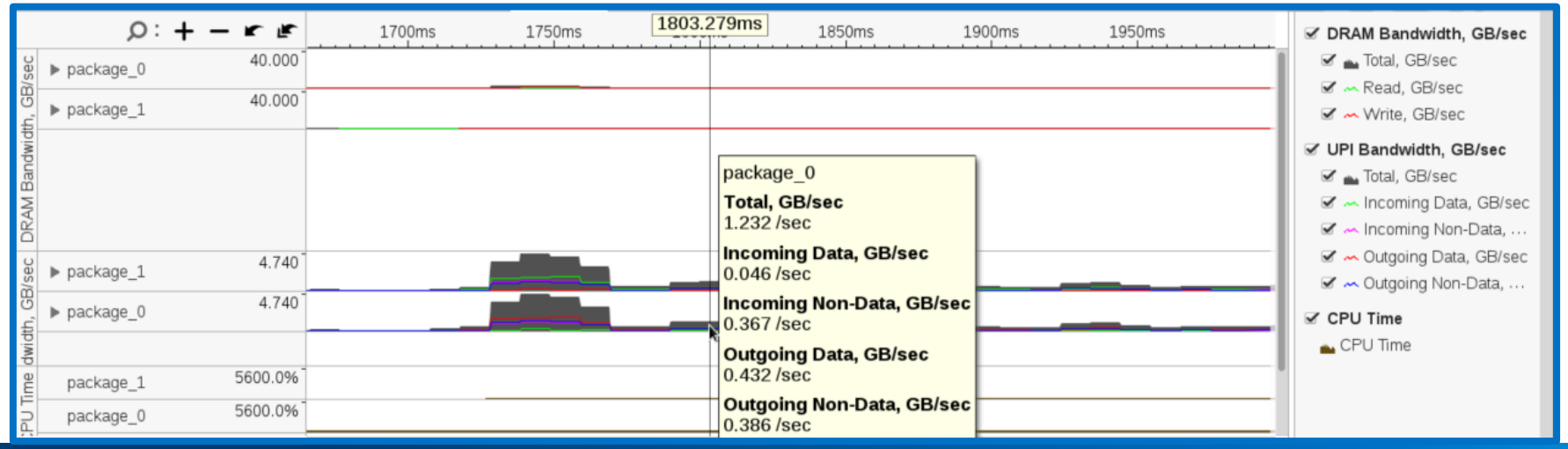
Function / Call Stack	CPU Time ▼	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)	M
multiply1._omp_fn.0	24.920s	44.2%	17,555,926,662	8,765,362,953	45,902,754	57	matrix.g
func@0x18c60	0.045s	40.0%	1,800,054	0	0	0	libgomp
clear_page_c_e	0.026s	0.0%	0	0	0	0	vmlinux
copy_page_rep	0.016s	89.0%	0	0	0	0	vmlinux
ktime_get	0.006s	85.0%	900,027	1,800,054	0	7	vmlinux
pci_conf1_read	0.006s	0.0%	0	900,027	450,027	0	vmlinux
apic_timer_interrupt	0.006s		0	0	0	0	vmlinux

VTune Amplifier Workflow Example- Bottom-Up View



Focus on areas of interest with "Zoom In and Filter"

Fine-grained details in Zoomed-in view



VTune Amplifier Workflow Example- Bottom-Up View

Memory Access Memory Usage viewpoint (change) INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform | multiply.c

Analyze Bandwidth over Time

0s 1s 2s 5s 6s

DRAM Bandwidth, GB/sec — package_0 40.000; package_1 40.000

UPI Bandwidth, GB/sec — package_1 8.728; package_0 8.728

CPU Time — package_1 5600.0%; package_0 5600.0%

DRAM Bandwidth, GB/sec: Total, GB/sec; Read, GB/sec; Write, GB/sec

UPI Bandwidth, GB/sec: Total, GB/sec; Incoming Data, GB/sec; Incoming Non-Data, G…; Outgoing Data, GB/sec; Outgoing Non-Data, G…

CPU Time: CPU Time

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency
multiply1._omp_fn.0	24.920s	44.2%	17,555,926,662	8,765,362,953	45,902,754	
func@0x18c60	0.045s	40.0%	1,800,054	0	0	
clear_page_c_e	0.026s	0.0%	0	0	0	
copy_page_rep	0.016s	89.0%	0	0	0	
ktime_get	0.006s	85.0%	900,027	1,800,054	0	
pci_conf1_read	0.006s	0.0%	0	900,027	450,027	
apic_timer_interrupt	0.006s		0	0	0	
init_arr	0.006s	0.0%	0	0	0	
init_arr	0.006s	0.0%	0	0	0	

DRAM and UPI Bandwidth are low.

Memory Bound function. 44% of pipeline slots are stalled.

Double-click a function for source view.

VTune Amplifier Workflow Example- Source View

Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform | multiply.c | multiply.c

Source | Assembly | Assembly grouping: Address

S. Li.	Source	CPU Time	Memory Bound				Loa..	Sto..
			L1 Bou.	L2 Bou.	L3 Bound	D. Bo.		
170	}							
171	}							
172	}							
179	#pragma omp parallel for							
180	for(i=0; i<msize; i++) {							
181	for(j=0; j<msize; j++) {	0.004s	0.0%		81.8%		4,5..	0
182	for(k=0; k<msize; k++) {	15.027s	0.0%		27.1%		90..	90..
183	c[i][j] = c[i][j] + a[i][k] * b[k][j];	9.889s	2.9%	0.0%	35.9%	0.0%	17,..	8,7..
184	}							

Metrics at a source line granularity

Inefficient array access pattern in nested loop

VTune Amplifier Workflow Example- Source View

Source | Assembly | Assembly grouping: Address

S. Li.	Source	CPU Time	Memory Bound	Loa..	Sto..	LL. Mi. C.	Ave.. Lat.. (cy...	So.. File
208	for(i=0; i<msize; i++) {							
209	for(k=0; k<msize; k++) {							
210	#pragma ivdep							
211	for(j=0; j<msize; j++) {	0.024s	0.0%	24,..	0	0	17	mu...
212	c[i][j] = c[i][j] + a[i][k] * b[k][j];	5.838s	6.1%	22,..	9,1..	0	16	mu...
213	}							

Loop interchange

Memory A

Collection L

Source | Asse

S. Li.	Source							
170								
171	}							
172	}							
179	#pragma omp parallel for							
180	for(i=0; i<msize; i++) {							
181	for(j=0; j<msize; j++) {	0.004s	0.0%	81.8%	4,5..	0		
182	for(k=0; k<msize; k++) {	15.027s	0.0%	27.1%	90..	90..		
183	c[i][j] = c[i][j] + a[i][k] * b[k][j];	9.889s	2.9%	0.0%	35.9%	0.0%	17,..	8,7..
184	}							

Compare the Results

Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

Elapsed Time: 6.689s

CPU Time:	25.121s	
Memory Bound:	44.4%	of Pipeline Slots
L1 Bound:	0.7%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	30.5%	of Clockticks
DRAM Bound:	8.0%	of Clockticks
Loads:	17,604,528,120	
Stores:	8,789,663,682	
LLC Miss Count:	46,352,781	
Average Latency (cycles):	57	
Total Thread Count:	4	
Paused Time:	0s	

System Bandwidth

This section provides various system bandwidth-related properties detected by the
and Low bandwidth utilization thresholds for the Bandwidth Utilization Histogram and

Max DRAM System Bandwidth:	80 GB
Max DRAM Single-Package Bandwidth:	40 GB

Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform | multiply.c

Elapsed Time: 1.640s

CPU Time:	5.988s	
Memory Bound:	7.1%	of Pipeline Slots
L1 Bound:	0.3%	of Clockticks
L2 Bound:	N/A*	of Clockticks
L3 Bound:	0.4%	of Clockticks
DRAM Bound:	N/A*	of Clockticks
Loads:	22,926,387,771	
Stores:	9,153,274,590	
LLC Miss Count:	0	
Average Latency (cycles):	16	
Total Thread Count:	4	
Paused Time:	0s	

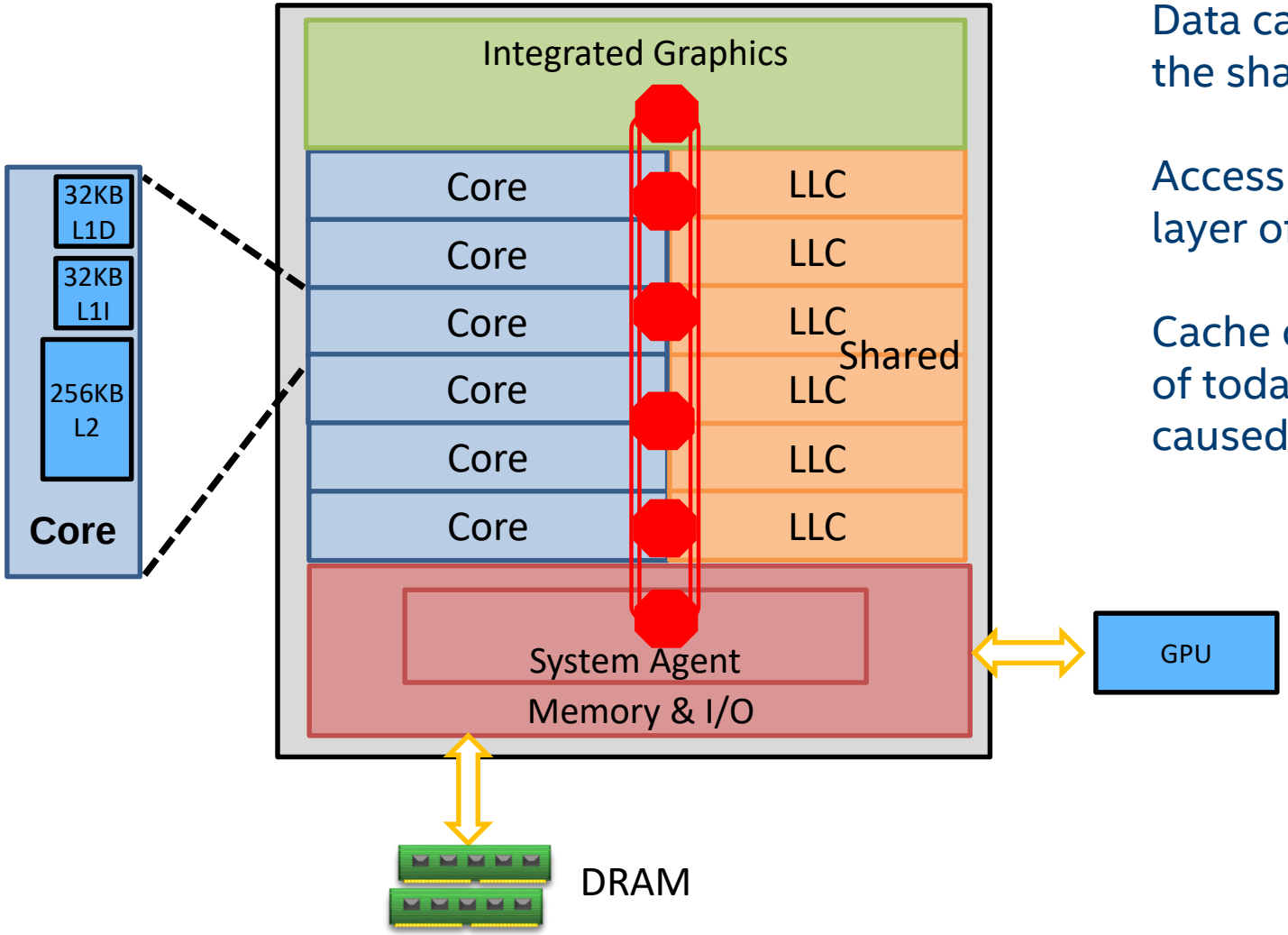
*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

System Bandwidth

This section provides various system bandwidth-related properties detected by the proc
default High, Medium and Low bandwidth utilization thresholds for the Bandwidth Utilizati
bandwidth graphs in the Bottom-up view.

Max DRAM System Bandwidth:	80 GB
Max DRAM Single-Package Bandwidth:	40 GB

Understanding the Memory Hierarchy



Data can be in any level of any core's cache, or in the shared L3, DRAM, or on disk.

Accessing data from another core adds another layer of complexity

Cache coherence protocols – beyond the scope of today's lecture. But we will cover some issues caused by this.

Cache Misses

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

Back-End Bound				
Memory Bound				
L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound
20.0%	0.0%	6.7%	0.0%	0.0%
0.0%		0.0%		0.0%

"<memory level> Bound" = Percentage of cycles when the CPU is stalled, waiting for data to come back from <memory level>

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 streaming stores
- Use software prefetch instructions

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

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.

4K Aliasing

Why: Aliasing conflicts caused by associative caches result in having to re-issue loads.

What Now: If this metric is highlighted for your hotspot, investigate at the source code level.

Fix these issues by changing the alignment of the load. Try aligning data to 32 bytes, changing offsets between input and output buffers (if possible), or using 16-Byte memory accesses on memory that is not 32-Byte aligned.

DTLB Misses

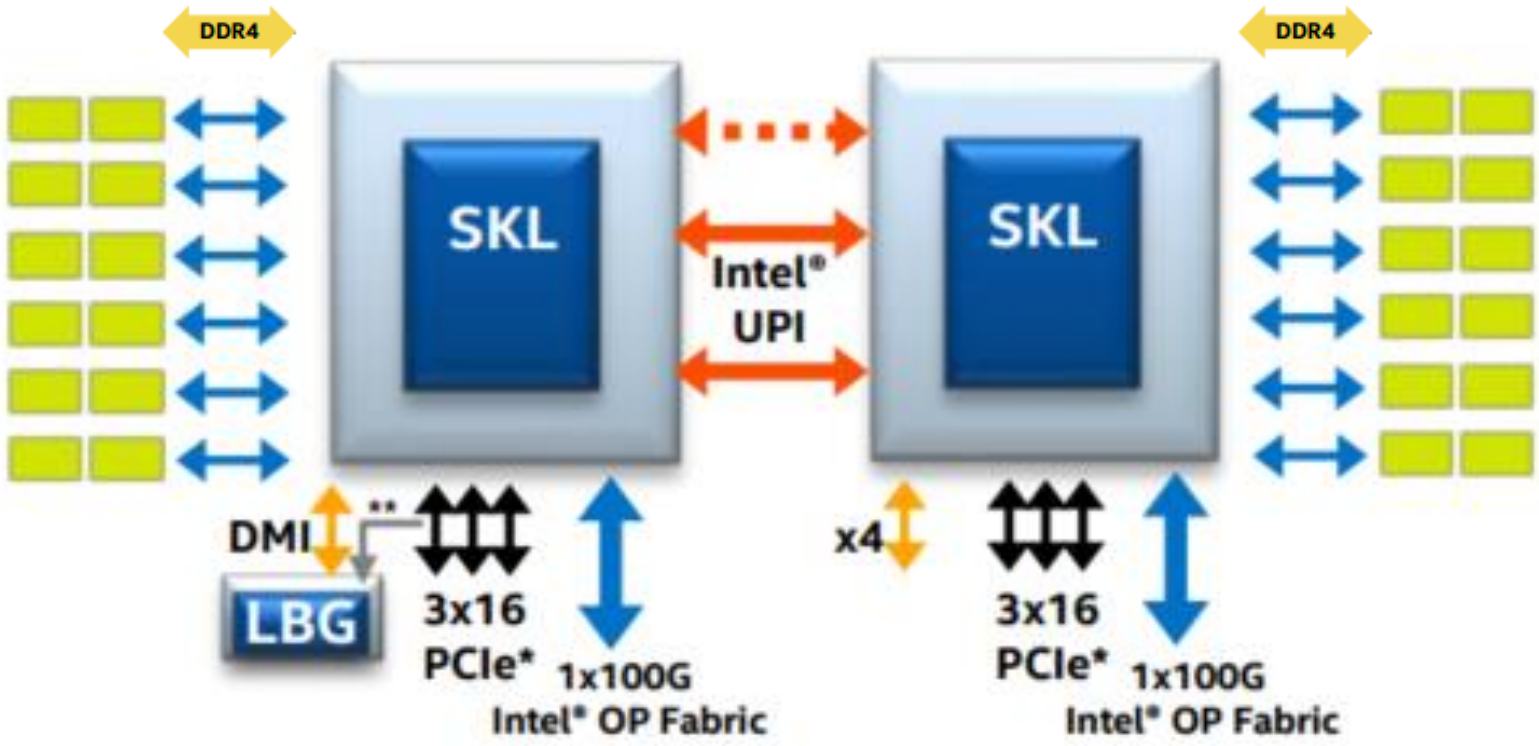
Why: First-level DTLB Load misses (Hits in the STLB) incur a latency penalty. Second-level misses require a page walk that can affect your application's performance.

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

Multi-Socket Systems

- This is still a single address space
- Accessing other socket is expensive
- Important to be aware of memory accesses for performance



Remote Memory Accesses

Why: With a Non-Uniform Memory Access (NUMA) architecture, loading from memory can have varying latency. Remote memory loads cost more.

What Now: If this metric is highlighted for your hotspot, improve NUMA affinity:

- If thread migration is a problem, try affinitizing or pinning threads to cores
- Ensure that memory is first touched (accessed, not allocated) by the thread that will be using it
- 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)

Example: Poor NUMA Utilization

Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

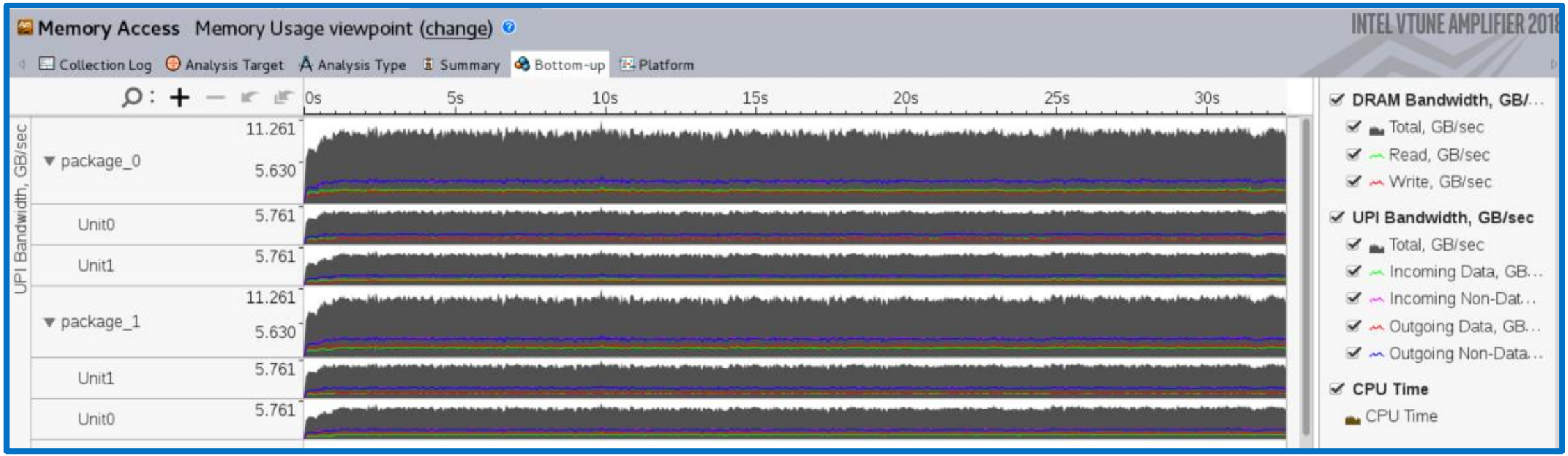
Elapsed Time: 32.626s

CPU Time:	508.508s	
Memory Bound:	**73.3%**	**of Pipeline Slots**
L1 Bound:	8.4%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	7.4%	of Clockticks
DRAM Bound:	**8.0%**	**of Clockticks**
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
Memory Latency:		
Remote / Local DRAM Ratio:	0.000	
Local DRAM:	1.2%	of Clockticks
Remote DRAM:	0.0%	of Clockticks
Remote Cache:	11.4%	of Clockticks
Loads:	163,640,209,059	
Stores:	34,303,629,078	
LLC Miss Count:	**331,669,899**	
Average Latency (cycles):	12	
Total Thread Count:	17	
Paused Time:	0s	

If Memory Bound is high and local caches are not the problem

Focus on “Remote” metrics

Example: Poor NUMA Utilization



Look for areas of high QPI/UPI bandwidth

QPI/UPI Bandwidth is communication between the sockets.
This may indicate some sort of NUMA issue.

Causes of poor NUMA utilization

- Allocation vs. first touch memory location
- False sharing of cache lines
  - Use padding when necessary
- Arbitrary array accesses
- Poor thread affinity

Where is your memory allocated and where are your threads running?

Memory Object Identification

Identifying line of source code may not be sufficient to identify the problem

Microarchitecture Analysis

- General Exploration
- Memory Access
- TSX Exploration
- TSX Hotspots

1

☑ Analyze dynamic memory objects

Minimal dynamic memory object size to track, in bytes

1024

View allocated objects

Sort by LLC Miss Count

Grouping: Memory Object / Function / Call Stack

Memory Object / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores ▼	LLC Miss Count	Average Latency (cycles)	Module	Function
▶ memTest.out!main (2 MB)			236,276,88...	20,334,310,011	83,705,022	9		
▶ memTest.cpp:10 (4 KB)			0	108,903,267	0	0		
▶ memTest.cpp:20 (4 KB)			0	66,601,998	0	0		
▶ memTest.cpp:11 (4 KB)			0	64,801,944	0	0		
▶ memTest.cpp:21 (4 KB)			0	58,501,755	0	0		
▶ memTest.cpp:25 (4 KB)			0	53,101,593	0	0		
▶ memTest.cpp:18 (4 KB)			0	53,101,593	0	0		

FILTER 0.0% | Any Process | Thread Any Thread | Module Any Module | Show inline functions | Functions only

Memory Object Identification

Memory Access Memory Usage viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform | memTest.... | memTes...

Source | Assembly | Assembly grouping: Address

S.	Source
1	#define SZ 1024
2	
3	#include <iostream>
4	#include "omp.h"
5	
6	using namespace std;
7	
8	int main() {
9	omp_set_num_threads(16);
10	int * a0= new int[SZ];
11	int * a1= new int[SZ];
12	int * a2= new int[SZ];
13	int * a3= new int[SZ];
14	int * a4= new int[SZ];
15	int * a5= new int[SZ];
16	int * a6= new int[SZ];
17	int * a7= new int[SZ];

Assembly view also available

Double-click to see allocation site in source view

Go further with Storage Device Analysis

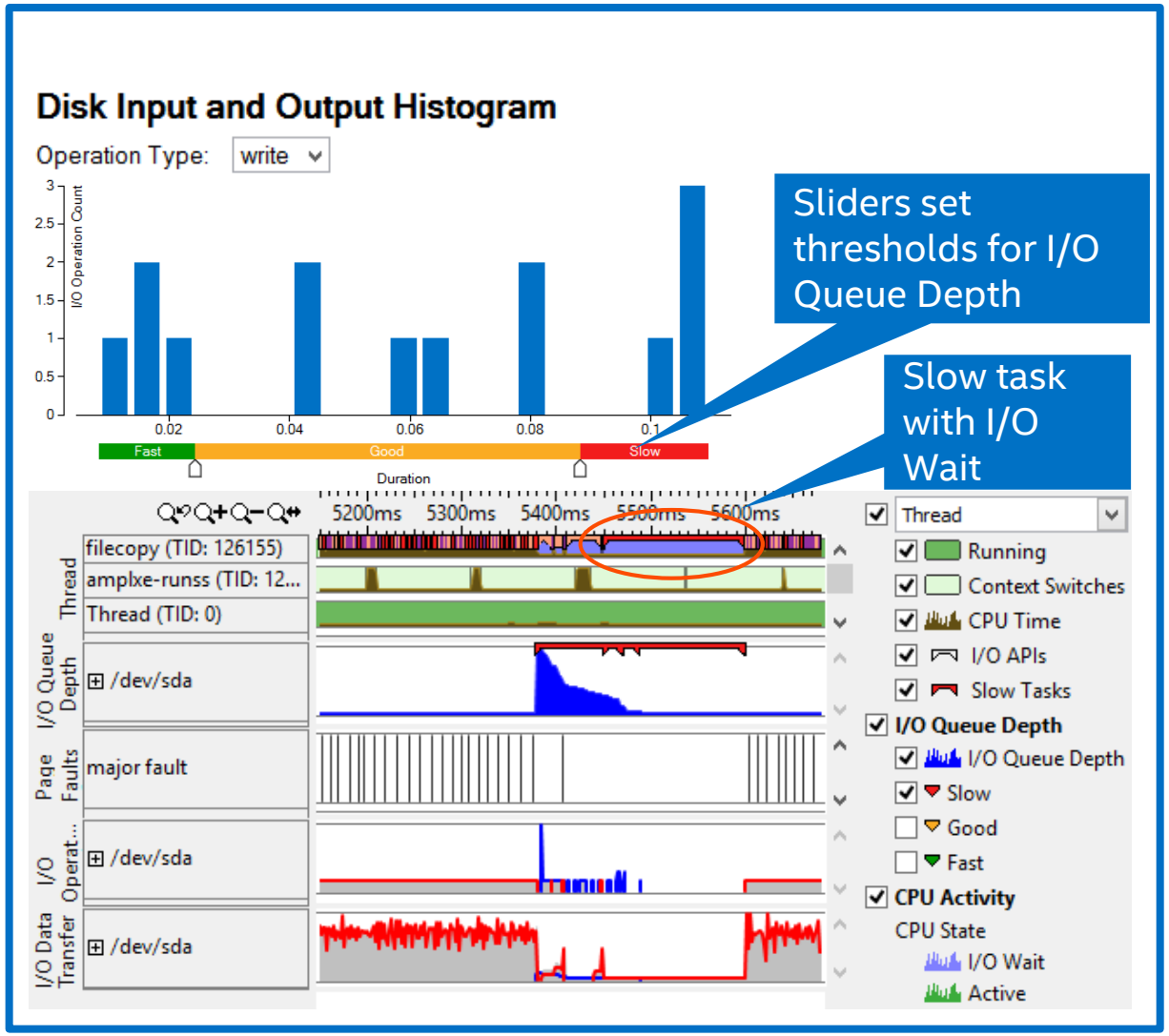
(HDD, SATA or NVMe SSD)

Are You I/O Bound or CPU Bound?

- Explore imbalance between I/O operations (async & sync) and compute
- Storage accesses mapped to the source code
- See when CPU is waiting for I/O
- Measure bus bandwidth to storage

Latency analysis

- Tune storage accesses with latency histogram
- Distribution of I/O over multiple devices



Bonus Topic – Instruction Fetch Issues

- Processor fetches instructions from the application binary in order
- Jumps and branches can cause the fetch to take longer
- % of cycles spent on ICache Misses (newer processors):
  - ICACHE_16B.IFDATA_STALL / CPU_CLK_UNHALTED.THREAD
- Instruction Starvation
  - UOPS_ISSUED.CORE_STALL_CYCLES-RESOURCE_STALLS.ANY/CPU_CLK_UNHALTED.THREAD
- Interpreted code (Python, Java etc…) and branchy code may have these issues
- Look for Profile Guided Optimizations from the compiler

Bonus Topic – Branch Misprediction

- Mispredicted branches cause instructions to execute, but never retire
- Result in wasted work
  - BR_MISP_RETIRED.ALL_BRANCHES_PS (newer processors)
- Cycle accounting on lab machines
  - 15 * BR_MISP_EXEC.ANY / CPU_CLK_UNHALTED.THREAD
- 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

Bonus Topic – Floating Point Arithmetic

Why: Floating point arithmetic can be expensive if done inefficiently.

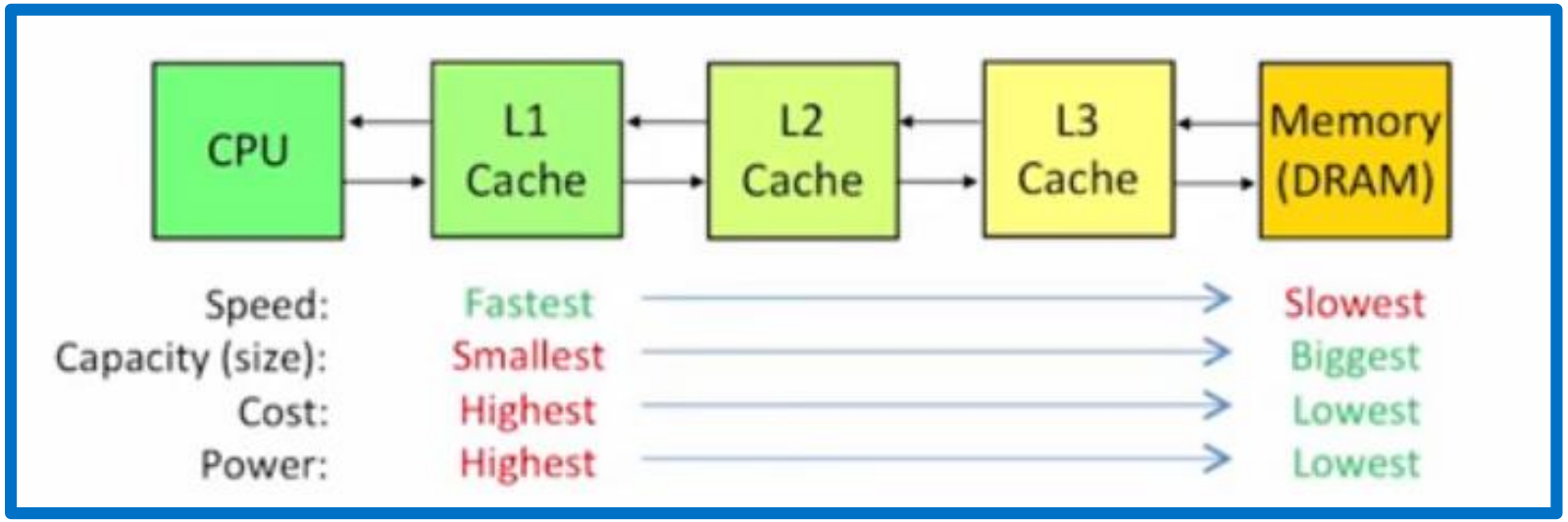
What Now: If FP x87 or FP Scalar metrics are significant, look to increase vectorization.

- Intel Compiler /QxCORE-AVX2 (Windows*) or –xCORE-AVX2 (Linux*) switches
- GCC: -march=core-avx2
- Optimize to AVX – See the Intel® 64 and IA-32 Architectures Optimization Reference Manual, chapter 11

General Retirement			
FP Arithmetic			Other
FP x87	FP Scalar	FP Vector	
0.000	0.140	0.000	0.860
0.000	0.192	0.000	0.808
0.000	0.000	0.000	1.000

Summary

- Memory continues to be the most common bottleneck for performance
- It's not enough to just profile and characterize
- Performance engineers need to pinpoint the problem
- Tools like VTune Amplifier are essential



intel®

Software