

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 a 6th generation Intel® Core™ processor

Using Intel® VTune™ Amplifier XE performance analyzer

- The performance information here applies to other tools as well 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, /QxAVX2, etc)
- Chosen an appropriate workload
- Measured baseline performance

Using Intel® VTune™ Amplifier XE to Tune Software on the 6th generation Intel® Core™ processor family

Software and Services Group

Ver. 1.0

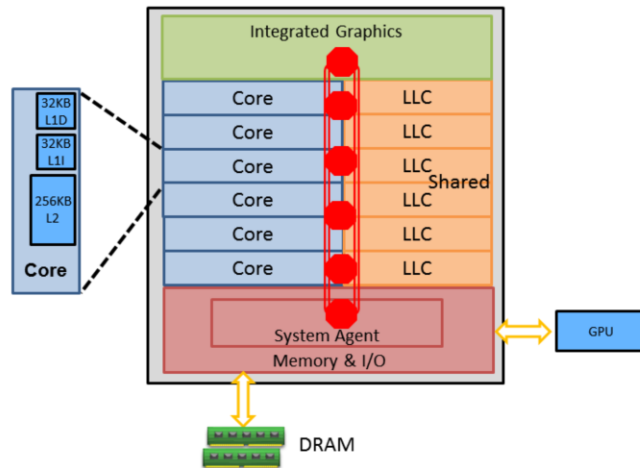
Optimization Notice



Agenda

- Intel® VTune™ Amplifier XE
- The Software Optimization Cycle
 - Find Hotspots
 - Methods for Determining Efficiency
 - Locating the Primary Bottleneck
 - Tuning for Common Architectural Causes of Inefficiency
- Additional Tuning Recommendations

6th Generation Intel® Core™ Processor



Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



This image represents a general CPU layout and is intended to help illustrate the concepts described in this guide. It is not designed to be a definitive representation of the microarchitecture.

Intel® microarchitecture codename Skylake CPUs -
<http://ark.intel.com/products/codename/37572/Skylake>

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*



Most screenshots in this presentation were taken from Intel® VTune™ Amplifier XE 2016 Update 2. Screenshots from different versions of the tool may have minor differences.

Pre-Configured Profiles in Amplifier XE

The General Exploration profile should be used for a top-level analysis of potential issues on the Intel microarchitecture code name Skylake family. It is the subject of this guide.

All the events required are pre-configured - no research needed! Simply click Start to run the analysis.

Event Name	Sample After	Precise	
INSTR_RETIRED.L1CIPS	200000		Any user executed
INSTR_RETIRED.L2CIPS	100000		Counts the total #
IR_BSP_RETRYED.ALL_BRANCHES_PS	400000		Misspredicted branch
CPU_CLK_UNHALTED.REF_TSC	2000000		Reference cycles in
CPU_CLK_UNHALTED.THREAD	2000000		Core cycles when #
CPU_CLK_UNHALTED.THREAD_P			
CYCLE_ACTIVITY.CYCLES_HL_PENDING			
CYCLE_ACTIVITY.STALLS_L2_PENDING			
CYCLE_ACTIVITY.STALLS_L3_PENDING			

The logic for identifying issues on Intel Microarchitecture Codename Skylake 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 all other profiles, it will just show the raw data.

Enhanced General Exploration View

The enhanced view is present when running the General Exploration profile with the General Exploration viewpoint selected (the default).

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring	Module	Function (Full)	Source File	Start Address
grid_intersect	14,948,022,422	12,672,016,000	1.180	7.0%	3.9%	74.8%	14.3%	3_tachyon_omp.exe	grid_intersect	grid.cpp	0x401071
isphere_intersect	9,618,014,427	9,038,013,539	1.066	5.7%	4.5%	79.1%	16.6%	3_tachyon_omp.exe	isphere_intersect	isphere.cpp	0x401060
grid_bound_intersect	1,064,001,596	690,001,005	1.542	0.0%	4.5%	75.0%	12.3%	3_tachyon_omp.exe	grid_bound_intersect	grid.cpp	0x401060
func@0x100207f1	602,000,903	420,000,630	1.423	22.1%	35.1%	9.0%	48.0%	libomp5end.dll	func@0x100207f1		0x100207f1
postgrid	248,000,369	160,000,240	1.538	7.7%	0.0%	84.6%	23.2%	3_tachyon_omp.exe	postgrid	grid.cpp	0x401140
shader	212,000,318	190,000,285	1.116	4.5%	0.0%	91.0%	4.3%	3_tachyon_omp.exe	shader(struct ray *)	shader.cpp	0x4010d0
tri_intersect	190,000,297	182,000,258	1.037	4.8%	4.8%	76.0%	17.4%	3_tachyon_omp.exe	tri_intersect	triangle.cpp	0x4010d0
_amp_00_pause	188,000,278	82,000,120	2.348	0.0%	0.0%	17.6%	10.0%	libomp5end.dll	_amp_00_pause		0x100207d0
Rayopt	172,000,258	256,000,384	0.672	16.6%	16.6%	44.8%	22.1%	3_tachyon_omp.exe	Rayopt(struct ray *, d...	vector.cpp	0x401070
Selected 1 row(s)	14,948,022,422	12,672,016,000	1.180	7.0%	3.9%	74.8%	14.3%				

All collected data is presented in hierarchical format (see next slide), with helpful metrics already calculated (see issue slides).

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others. Optimization Notice

Enhanced General Exploration View

General Exploration General Exploration viewpoint (change) ?

Analysis Target Analysis Type Collection Log Summary Bottom

Hierarchical data display corresponds to how available execution slots in each core's pipeline are utilized.

Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring				
grid_intersect	14,948,022,422	12,672,019,008	1.180	7.0%	3.9%	74.8%	14.3%				
isphere_intersect	9,618,014,427	9,026,013,539	1.066	5.7%	4.5%	73.1%	16.6%				
grid_bounds_intersect	1,064,001,596	690,001,035	1.542	8.0%	4.8%	75.0%	12.5%				
func@0x1002df7f	602,000,903	420,000,630	1.433	22.1%	4.8%	75.0%	48.9%				
pos2grid	246,000,369	160,000,240	1.538	7.7%	4.8%	75.0%	23.2%				
shader	212,000,318	190,000,285	1.116	4.5%	4.8%	75.0%	4.5%				
tri_intersect	198,000,297	192,000,288	1.031	4.8%	4.8%	75.0%	14.4%				
_kmp_x86_pause	184,000,276	82,000,123	2.244	0.0%	4.8%	75.0%	100.0%				
Raypnt	172,000,258	256,000,384	0.672	16.6%	4.8%	75.0%	22.1%				
...				
Selected 1 row(s):				14,948,022,422	12,672,019,008	1.180	7.0%	3.9%	0.0%	4.8%	14.3%

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

Bad Speculation	Branch Mispredict	Machine Clears
3.9%	0.0%	0.0%
4.5%	0.0%	0.0%
4.5%	0.0%	0.0%
0.0%	36.3%	0.0%
0.0%	0.0%	0.0%
0.0%	0.0%	0.0%
0.0%	4.8%	0.0%
0.0%	16.6%	0.0%
0.0%	0.0%	0.0%
3.9%	0.0%	0.0%

Enhanced General Exploration View

General Exploration - General Exploration Intel VTune

Analysis Target Analysis Type Summary Bottom-up Top-down Tree Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	Hardware Ev...		CPI		Filled Pipeline Slots		Unfilled Pipel...		Module
	CPU_CL... THREAD	INST_RETIRED. ANY	Rate	Re.	Branch Mispredict	Machine Clears	Ba. Bo.	Fr. Bo.	
#grid_Intersect	8,130,000,000	10,560,000,000	0.770	0.328	0.162	0.006	0.458	0.125	analyze_locks.exe grid_inter
#sphere_Intersect	7,512,000,000	10,326,000,000	0.727	0.372	0.016	0.027	0.478	0.038	analyze_locks.exe sphere_in
#grid_bounds_Intersect	1,196,000,000	862,000,000	1.387	0.192	0.100	0.000	0.607	0.088	analyze_locks.exe grid_bour
#GdpCreateSolidFill	652,000,000	564,000,000	1.156	0.276	0.000	0.000	0.628	0.073	gdplus.dll GdpCreat
#pos2grid	246,000,000	236,000,000	1.042	0.193	0.000	0.000	0.563	0.071	analyze_locks.exe pos2grid
#[rdpdd.dll]	226,000,000	476,000,000	0.475	0.431	0.531	0.000	0.226	0.232	rdpdd.dll [rdpdd.dll]
#shader	208,000,000	160,000,000	1.300	0.252	0.000	0.000	0.760	0.108	analyze_locks.exe shader(st
#[TBB Scheduler Internals]	170,000,000	46,000,000	3.696	0.544	0.000	0.000	0.441	0.000	tbb.dll [TBB Scheduler Internals]
#Rayprint	166,000,000	253,000,000	0.664	0.392	0.000	0.000	0.548	0.015	analyze_locks.exe Rayprint(st
Selected 1 row(s): 8,130,000,000 10,560,000,000 0.770 0.328 0.162 0.006 0.458 0.125									

Pre-computed metrics for each category of pipeline utilization saves users analysis time.

For a given hotspot, if a cell is highlighted pink, it means the value for that metric is over VTune Amplifier XE's pre-determined threshold and should be investigated.

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others. Optimization Notice

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

Complexities of Performance Measurement

Two features of the 5th generation Intel® Core™ processor family have a significant effect on performance measurement:

- Intel® Hyper-Threading Technology
- Intel® Turbo Boost 2.0 Technology

With these features enabled, it is more complex to measure and interpret performance data

- Most events are counted per thread, some events per core
 - See VTune Amplifier XE Help for specific events

Some experts prefer to analyze performance with these features disabled, then re-enable them once optimizations are complete

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 specifics prior to making changes. Incorrectly modifying BIOS settings from those supplied by the manufacturer can result in rendering the system unusable and may void the warranty.

Don't forget to re-enable these features once you are through with the software optimization process!

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

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

Identify the Hotspots

What: Hotspots are where your application spends the most time

Why: You should aim your optimization efforts there!

- Why improve a function that only takes 2% of your application's runtime?

How: VTune Amplifier XE *Basic Hotspots* or *Advanced Hotspots* analysis type

- Usually hotspots are defined in terms of the CPU_CLK_UNHALTED.THREAD event (aka "clockticks")

For this processor family, the CPU_CLK_UNHALTED.THREAD counter measures unhalted clockticks on a per hardware thread basis. There is no per-core clocktick counter, which has been available on some previous processors. The CPU_CLK_UNHALTED.THREAD counter allows you to see where cycles are being spent on each individual hardware thread.

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. This counter can be useful for removing the variance introduced by Turbo Mode 2.0 or Speedstep Technology when comparing multiple analyses.

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

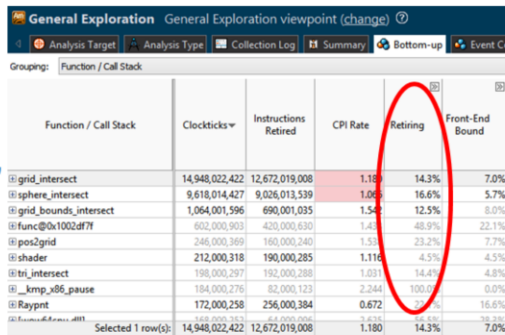
Efficiency Method 1: % Retiring Pipeline Slots

Why: Helps you understand how efficiently your app is using the processors

How: *General Exploration* profile, Metric: *Retiring*

What Now:

- For a given hotspot:
- If 75% or more of pipeline slots are retiring, go to efficiency method 3, code study 1 – to see if vectorization can boost performance further
- Otherwise, see next slide



Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound
grid_intersect	14,948,022,422	12,672,019,008	1.18	14.3%	7.0%
sphere_intersect	9,618,014,427	9,026,013,539	1.06	16.6%	5.7%
grid_bounds_intersect	1,064,001,596	690,001,035	1.54	12.5%	8.0%
func@0x1002df7	602,000,903	420,000,630	1.43	48.9%	22.1%
pos2grid	246,000,369	160,000,240	1.53	23.2%	7.7%
shader	212,000,318	190,000,285	1.116	4.5%	4.5%
tri_intersect	198,000,297	192,000,288	1.031	14.4%	4.8%
_kmp_x86_pause	184,000,276	82,000,123	2.244	100.0%	0.0%
Rayprint	172,000,258	256,000,384	0.672	14.3%	16.6%
Selected 1 row(s):	14,948,022,422	12,672,019,008	1.180	14.3%	7.0%

Formula:

$(\text{UOPS_RETIRED.RETIRE_SLOTS} / (4 * \text{CPU_CLK_UNHALTED.THREAD}))$

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.

More details on this methodology are available in the coming slides or in this paper: <http://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues>

Efficiency Method 1: % Retiring Pipeline Slots

What Now: For a hotspot with < 70% pipeline slots retiring, consider the application type when determining efficiency. If the hotspot is below the expected range below, it may be inefficient.

	Expected Range of Pipeline Slots in this Category, for a Hotspot in a Well-tuned:		
Category	Client/ Desktop application	Server/ Database/ Distributed application	High Performance Computing (HPC) application
Retiring	20-50%	10-30%	30-70%

For more details see: <http://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues>

Efficiency Method 2: Changes in Cycles per Instruction (CPI)

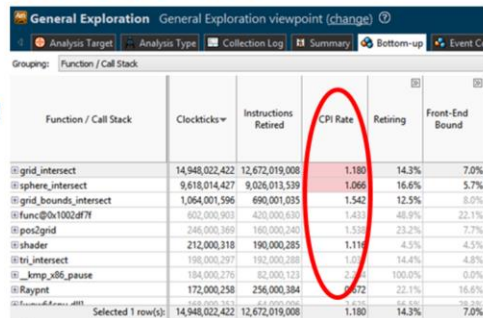
Why: Another measure of efficiency that can be useful when comparing 2 sets of data

- Shows average time it takes one of your workload's instructions to execute

How: *General Exploration* profile, Metric: *CPI Rate*

What Now:

- CPI can vary widely depending on the application and platform!
- If code size stays constant, optimizations should focus on reducing CPI



Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound
grid_intersect	14,948,022,422	12,672,019,008	1.180	14.3%	7.0%
sphere_intersect	9,618,014,427	9,026,013,539	1.066	16.6%	5.7%
grid_bounds_intersect	1,064,001,596	690,001,035	1.542	12.5%	8.0%
func@0x1002df7f	602,000,903	420,000,630	1.433	48.9%	22.1%
pos2grid	246,000,369	160,000,240	1.538	23.2%	7.7%
shader	212,000,318	190,000,285	1.116	4.5%	4.5%
tri_intersect	198,000,297	192,000,288	1.031	14.4%	4.8%
_kmp_x86_pause	184,000,276	82,000,123	2.243	100.0%	0.0%
Raypnt	172,000,258	256,000,384	0.672	22.1%	16.6%
Selected 1 row(s)	14,948,022,422	12,672,019,008	1.180	14.3%	7.0%

Formula:

$$\text{CPU_CLK_UNHALTED.THREAD/INST_RETIRED.ANY}$$

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.

Additionally, CPI can be affected if using Intel® Hyper-threading. In a serial workload, or a workload with Intel® Hyper-threading disabled the theoretical best CPI on a hardware thread is 0.25 because the core can allocate and retire 4 instructions per cycle. In a workload with Intel® Hyper-threading enabled which utilizes both hardware threads effectively, the ideal CPI per-thread would be 0.5 instead of 0.25. This is because the hardware threads share allocation and

retirement resources on the core.

Note: Optimized code (e.g. with AVX instructions) may actually increase the CPI, and increase stall % – but improve the performance. This is because a single vector instruction will generally take more cycles than a single scalar instruction, but it also often performs more work. For example, a vector instruction may take twice as many cycles, but perform the work of four scalar instructions. In that case, the average CPI will increase, but the application will still be running faster.

CPI is just a general efficiency metric – the real measure of efficiency is work taking less time.

Efficiency Method 3: Code Examination

Why: Methods 1 and 2 measure how long it takes instructions to execute. The other type of inefficiency is executing too many instructions.

How: Use VTune Amplifier XE's capability as a source and disassembly viewer

What Now:

- Failure to utilize modern instructions results in larger code size
- See next slides for potential issues

Source	Hardware	Hardware	Code	Hardware	Hardware	Rating	Assess	
Line	Source	CHU... THREAD	INSTR. AVG	Location	CHU... THREAD	INSTR. AVG	LOPS RETR...	IPC...
534	task = offset + (round2sig, end	4,000,000	0	0x00000546	push ebx	0	0	0
535	tblink = g->compression & / - sp+4	46,000,000	20,000,000	0x00000546	movq qword	4,000,000	0	0
536	step = m0_s = -1;			0x00000546	call @vfnz	4,000,000	0	0
537					Block #1			
538	while (0x00000547	movq rax,	8,000,000	0	0
539	task = offset + (round2sig, end			0x00000547	movq qword			
540	tblink = g->compression & / sp+4			0x00000547	movq qword	0	5,000,000	0
541	step = -1;			0x00000547	movq rax,	0	0	0
542	tblink = g->tblink;	96,000,000	42,000,000	0x00000547	movq qword	0	0	0
543				0x00000547	movq rax,	0	0	0
544				0x00000548	movq rax,	6,000,000	0	0
545	tblink = sp+4;	10,000,000	0	0x00000548	add esp, 4			
546	vfnz(vfnzfnzfnz, tblink, 0);	18,000,000	0	0x00000548	jmp vfnz			
547	tblink = sp+4;	8,000,000	2,000,000	0x00000548	movq qword	2,000,000	0	0
548	vfnz(vfnzfnzfnz, tblink, 0);	8,000,000	0	0x00000548	push vdx	0	0	0
549	tblink = sp+4;	10,000,000	0	0x00000548	call @vfnz			
550	vfnz(vfnzfnzfnz, tblink, 0);	12,000,000	0		Block #2			
551				0x00000549	movq rax,	4,000,000	0	0
552	tbl = tbl + tbl; tbl = tbl;	4,000,000	0	0x00000549	movq qword			
553	tbl = tbl + tbl; tbl = tbl;	4,000,000	16,000,000	0x00000549	movq rax,	4,000,000	0	0
554	tbl = tbl + tbl; tbl = tbl;	16,000,000	16,000,000	0x00000549	movq qword	2,000,000	0	0
555	tbl = tbl + tbl; tbl = tbl;	8,000,000	0	0x00000549	highlighted	8,000,000	0	0

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



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 Intel® Advanced Vector Extensions (AVX and AVX2)

Why: Using SIMD instructions can greatly increase floating point performance. For existing FP or Integer SSE code, converting to AVX instructions has several advantages, including support for wider vector data (up to 256-bit), 3- and 4-operand syntax that allows NDS operations, and power savings.

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:

- 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

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



For more on vectorization in general: <http://software.intel.com/en-us/articles/vectorization-toolkit>

SSE instructions will look like: `addps xmm4, xmm5`.

+ `addss` is a **s**(calar) Intel® SSE instruction – packed SSE instructions such as `addps` are a better choice

Efficiency Method 3, Code Study 2: Take Advantage of Fused Multiply Add (FMA) Instructions

Why: FMA instructions in Intel microarchitecture code name Haswell have the same latency as an FP Multiply. 2 new FMA units provide 2x peak FLOPs/cycle of previous generation.

How: Examine your source code for operations of the format $r=(x*y)+z$, $r=(x*y)-z$, $r=-(x*y)+z$, or $r=-(x*y)-z$. Then look at the corresponding assembly to see if FMA instructions are being used. FMA instruction names begin with **VFM** (as in `VFMAADD132PD` or `VFMSUBADD132PD`) or **VFNM** (as in `VFNMAADD132PD`)

What Now:

- Use Compiler switches to generate FMAs (you may need to specify a relaxed floating point model)
- Intel Compiler: `-fma` or `/Qfma` with `CORE-AVX2` or higher for option `-x` or `/Qx`, or `-march` or `/arch`
- GCC: `-xfma` or `-march=core-avx2`

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



Note that FMA instructions perform a multiply, add, and round. A multiply followed by an add would have 2 rounds (one after the multiply and one after the add). Since the FMA eliminates the intermediate rounding operation, results may be different when using FMAs as opposed to multiplies followed by adds. For more information, see the Intel® 64 and IA-32 Architectures Software Developer's Manual at <http://download.intel.com/products/processor/manual/325462.pdf>.

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

Determine the Primary Bottleneck

If Methods 1 or 2 are used to determine code is inefficient, first determine the primary bottleneck.

The Top-Down hierarchy implemented in General Exploration classifies your application's utilization of the CPU cores into 4 categories:

- Front-End Bound
- Back-End Bound
- Bad Speculation
- Retiring

The primary bottleneck has the highest fraction of pipeline slots, and should be investigated first!

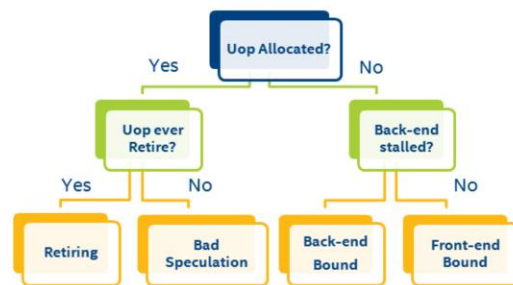
For a hotspot that is inefficient, determining the primary bottleneck is the first step. Optimizing code to fix issues outside the primary bottleneck category may not boost performance – the biggest boost will come from resolving the biggest bottleneck. Generally, if Retiring is the primary bottleneck, that is good. See next slides.

Issue Classification

A **Pipeline Slot** is an abstract concept – it represents the hardware resources needed to process one micro-operation

On this CPU, there are 4 pipeline slots available on each core, each cycle

Performance is classified according to what happened for each slot available to the application or hotspot:



Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



Note the way that 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 <http://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues>

Investigating the Bottleneck

Determine which category is the primary bottleneck, then compare the fraction for that category to the chart below.

- If the % for your hotspot is outside the range, investigate issues from that category

	Expected Range of Pipeline Slots in this Category, for a Hotspot in a Well-tuned:		
Category	Client/ Desktop application	Server/ Database/ Distributed application	High Performance Computing (HPC) application
Retiring	20-50%	10-30%	30-70%
Back-End Bound	20-40%	20-60%	20-40%
Front-End Bound	5-10%	10-25%	5-10%
Bad Speculation	5-10%	5-10%	1-5%

Key:  Higher is Better
 Lower is Better

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



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.

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.

What Now: Helps you **Optimize the Issue**. Gives suggestions for follow-up investigations or optimizations to try.

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 and can be found by hovering over a metric column in the GUI. You only need to use the pre-configured profiles and metrics pointed out in order to know if you may have a problem.

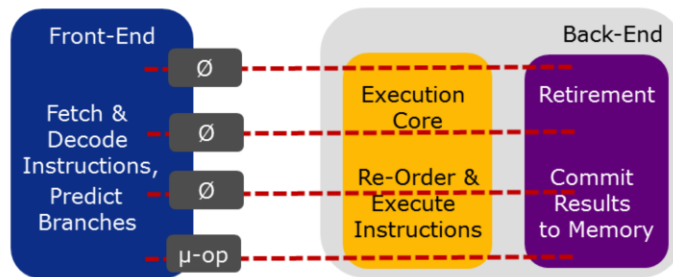
Tuning for the Front-End Bound Category

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 Bound, Illustrated



Occurs when the Front-End cannot deliver micro-ops for all 4 pipeline slots – generally caused by delays in fetching code or in decoding instructions

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

Retiring	Front-End Bound	Bad Speculation	Back-End Bound
14.3%	7.0%	3.9%	74.8%
16.6%	5.7%	4.5%	73.1%
12.5%	8.0%	4.5%	75.0%
48.9%	22.1%	36.3%	0.0%
23.2%	7.7%	0.0%	84.6%
4.5%	4.5%	0.0%	91.0%
14.4%	4.8%	4.8%	76.0%

If Front-end Bound is the primary bottleneck, concentrate on Front-End Latency. Resolve highlighted issues under this category.

Front-End Bound	
Front-End Latency	Front-End Bandwidth
5.6%	1.4%
4.7%	1.0%
7.1%	0.9%
6.3%	15.8%
15.4%	0.0%
0.	

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.

Front-End Bound									
Front-End Latency					Front-End Bandwidth				
ICac... Miss...	ITLB Ove...	Bran... Rest...	DSB Swit...	Len... Cha... Prefi...	MS Swit...	Fron... Band... DSB	Fron... Band... MITE	Fron... Band... LSD	
0.000	0.000	0.083	0.000	0.000	0.000	0.109	0.031	0.000	
0.000	0.000	0.020	0.008	0.000	0.000	0.036	0.008	0.000	
0.000	0.000	0.179	0.000	0.000	0.000	0.000	0.107	0.000	
0.000	0.003	0.000	0.000	0.000	0.000	0.568	0.000	0.000	
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.618	0.000	

Front-End Latency

Front-End
Bound

Why: Front-End latency can lead to the Back-End not having micro-ops to execute (instruction starvation).

How: *General Exploration Profile, Front-End Latency sub-category, Metrics: ITLB Overhead, ICache Misses, Length-Changing Prefixes*

What Now: If any of these metrics are highlighted for your hotspot, try using better code layout and generation techniques:

- Try using profile-guided optimizations (PGO) with your compiler
- Use linker ordering techniques (/ORDER on Microsoft's linker or a linker script on gcc)
- Use switches that reduce code size, such as /O1 or /Os
- For dynamically generated code, try co-locating hot code, reducing code size, and avoiding indirect calls

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



Formulas:

% of cycles spent on ITLB Misses (ITLB Overhead):

$ICACHE_64B.IFTAG_STALL / CPU_CLK_UNHALTED.THREAD$

% of cycles spent on ICache Misses:

$ICACHE_16B.IFDATA_STALL / CPU_CLK_UNHALTED.THREAD$

% of cycles due to LCP stalls:

$ILD_STALL.LCP / CPU_CLK_UNHALTED.THREAD$

Thresholds:

Thresholds: Investigate if –

% of cycles spent on ITLB Misses (ITLB Overhead) \geq .05 (5%)

% of cycles spent on ICache Misses \geq .05 (5%)

% of cycles due to LCP stalls \geq .05 (5%)

Note: To locate the exact areas in your code that are suffering from Front-End latency issues, create a custom analysis to collect the events

FRONTEND_RETIRED.L1I_MISS_PS

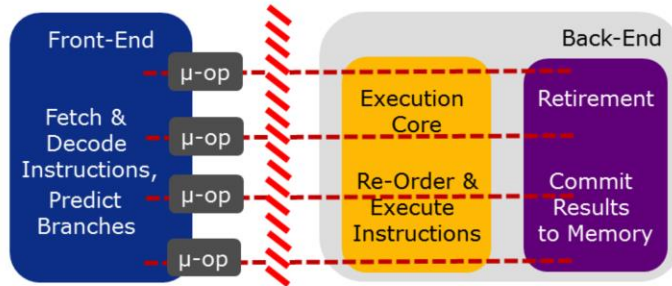
and FRONTEND_RETIRED.L2_MISS_PS. These are precise events for instruction fetch misses in the L1 instruction cache and L2 cache respectively.

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

Back-End Bound, Illustrated



Occurs when the Back-End cannot accept micro-operations for all 4 pipeline slots – usually because internal structures are already filled with micro-ops waiting on data.

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

Retiring	Front-End Bound	Bad Speculation	Back-End Bound
14.3%	7.0%	3.9%	74.8%
16.6%	5.7%	4.5%	73.1%
12.5%	8.0%	4.5%	75.0%
48.9%	22.1%	36.3%	0.0%
23.2%	7.7%	0.0%	84.6%
-4.5%	4.5%	0.0%	91.0%

Back-End Bound	
Memory Bound	Core Bound
40.8%	34.0%
42.5%	30.7%
48.2%	26.8%
0.0%	0.0%
0.0%	84.6%
18.2%	72.8%
0.0%	76.0%
0.0%	17.4%
22.4%	22.4%
0.0%	0.0%
40.8%	34.0%

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 saturated execution ports.

Back-End Hierarchical Breakdown in Amplifier XE

Back-End Bound	
Memory Bound	Core Bound
40.8%	34.0%
42.5%	30.7%
48.2%	26.8%
0.0%	0.0%
0.0%	84.6%
18.2%	72.8%
0.0%	76.0%
0.0%	17.4%
22.4%	22.4%
0.0%	0.6%
40.8%	34.0%

Back-End Bound			
Memory Bound			
L1 Bound	L2 Bound	L3 Bound	DRAM Bound
0.159	0.093	0.031	0.000
0.104	0.020	0.052	0.000
0.287	0.000	0.048	0.000
0.498	0.000	0.000	0.000

Expand the Memory Bound Category to see issues related to the various levels of the memory hierarchy.

Back-End Bound														
Memory Bound														
L1 Bound				L2 Bound				L3 Bound			DRAM Bound		Store Bound	
DTLB Ove...	Loa... Bloc...	Lock Late...	Split Loads	4K Alia...	FB Full	L2 Bou...	Con... Acc...	Data Shar...	L3 Late...	L3 Band...	SQ Full	Memory Bandwidth	Memory La... LLC Miss	Store Bound
0.094	0.000	0.000	0.000	0.000	0.000	0.063	0.000	0.000	0.411	0.247	0.000	0.000	0.000	0.000
0.084	0.000	0.000	0.000	0.012	0.000	0.071	0.000	0.000	0.547	0.369	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.342	0.000	0.000	0.000	0.000	0.000
0.003	0.000	0.000	0.000	0.020	0.000	0.058	0.000	0.000	0.596	0.404	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.411	0.247	0.000	0.000	0.000	0.000



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: *General Exploration Profile, Memory Bound* sub-category, Metrics: *L3 Bound>L3 Latency, DRAM Bound>Memory Latency>LLC Miss*

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 cache-line replacement analysis outlined in section B.3.4.2 of Intel® 64 and IA-32 Architectures Optimization Reference Manual
- Use streaming stores
- Use software prefetch instructions

Thresholds: Investigate if –
% cycles for LLC Miss $\geq .1$
% cycles for L3 Latency $\geq .2$

More information on many of these suggestions can be found in the Intel® 64 and IA-32 Architectures Optimization Reference Manual at:
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>

Contested Accesses

Why: Sharing modified data among cores (at L2 level) can raise the latency of data access

How: *General Exploration Profile, 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_L3_HIT_RETIRED.XSNP_HITM_PS event which will tag to the next instruction after the one that generated the HITM.
- 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 cache line boundaries

Formula:

% of cycles spent accessing data modified by another core:

$$\left((60 * \text{MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM_PS}) + \text{MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS_PS} \right) / \text{CPU_CLK_UNHALTED.THREAD}$$

Thresholds: Investigate if –

% cycles accessing modified data \geq .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: *General Exploration Profile, Memory Bound* sub-category, Metric: *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_L3_HIT_RETIRED.XSNP_HIT_PS event which will tag to the next instruction after the one that generated the HIT.
- 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 cache line boundaries

Formula:

% of cycles spent on Data Sharing:

$$\left(43 * \text{MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT_PS} \right) / \text{CPU_CLK_UNHALTED.THREAD}$$

Thresholds: Investigate if –

% cycles accessing clean shared data \geq .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: *General Exploration Profile, 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 = (13 * LD_BLOCKS.STORE_FORWARD) / 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.

Other Data Access Issues: 4K Aliasing

Why: Aliasing conflicts result in having to re-issue loads.

How: *General Exploration Profile, Memory Bound* sub-category,
Metric: *4K Aliasing*

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.

Formula:

$$\text{Aliasing Conflicts Cost} = (7 * \text{LD_BLOCKS_PARTIAL.ADDRESS_ALIAS}) / \text{CPU_CLK_UNHALTED.THREAD}$$

Threshold: Investigate if

Aliasing conflicts cost $\geq .1$

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 STL) incur a latency penalty. Second-level misses require a page walk that can affect your application's performance.

How: *General Exploration Profile, Memory Bound* sub-category, Metric: *DTLB 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:

$$\text{DTLB Overhead} = ((7 * \text{DTLB_LOAD_MISSES.STLB_HIT}) + \text{DTLB_LOAD_MISSES.WALK_DURATION}) / \text{CPU_CLK_UNHALTED.THREAD}$$

Threshold: Investigate if-
DTLB Overhead \geq .1

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

TLB: Translation Lookaside Buffer

DTLB: Data Translation Lookaside Buffer

STLB: Second-level Translation Lookaside Buffer

Back-End Hierarchical Breakdown in Amplifier XE

Back-End Bound	
Memory Bound	Core Bound
40.8%	34.0%
42.5%	30.7%
48.2%	26.8%
0.0%	0.0%
0.0%	84.6%
18.2%	72.8%
0.0%	76.0%
0.0%	17.4%
22.4%	22.4%
0.0%	0.0%
40.8%	34.0%

Expand the Core Bound Category to see issues related to execution core.

Back-End Bound												
Memory Bound	Divider	Core Bound										
		Port Utilization										
		Cycles of 0 Ports Utilized	Cycles of 1 Port Utilized	Cycles of 2 Ports Utilized	Cycles of 3+ Ports Utilized							
					Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
0.000	0.000	0.033	0.072	0.166	0.649	0.695	0.059	0.095	0.205	0.616	0.845	
0.230	0.000	0.008	0.232	0.199	0.207	0.248	0.712	0.637	0.224	0.091	0.339	0.174
0.072	0.000	0.000	0.275	0.321	0.378	0.481	0.814	0.837	0.298	0.092	0.195	0.115
0.073	0.000	0.061	0.061	0.061	0.122	0.337	0.184	0.153	0.153	0.398	0.245	0.122
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.282	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000



Divider

Why: Divide instructions take longer than other arithmetic instructions and can only be executed on a limited number of ports.

How: *General Exploration Profile, Core Bound* sub-category, Metric: *Divider*

What Now: If this metric is highlighted for your hotspot, locate the source code line(s) that is generating divides by viewing the source.

- Look for the `ARITH.DIVIDER_ACTIVE` event.
- Ensure the divide code is being compiled with optimizations turned on
- Use vectorized divide instructions
- Use reciprocal-multiplication whenever possible

Formula:

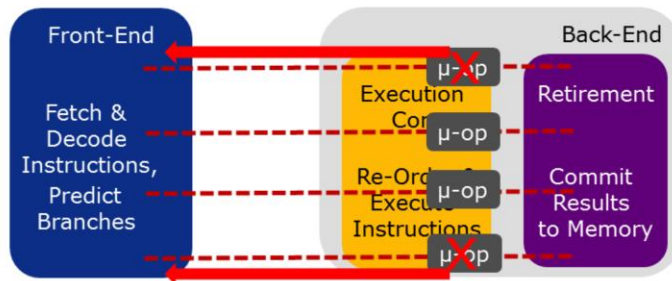
% of cycles spent with the divider unit active:

`ARITH.DIVIDER_ACTIVE / CPU_CLK_UNHALTED.THREAD`

Thresholds: Investigate if –

% cycles with an active divider $\geq .05$

Bad Speculation, Illustrated



Occurs when micro-ops are removed from the Back-End and do not retire.

Micro-operations that are removed from the Back-End most likely happen because the Front-End mispredicted 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 redirected 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:

$$\text{Branch Mispredict} = \frac{\text{BR_MISP_RETIRED.ALL_BRANCHES_PS}}{(\text{BR_MISP_RETIRED.ALL_BRANCHES_PS} + \text{MACHINE_CLEARS.COUNT}) * (\text{UOPS_ISSUED.ANY} - \text{UOPS_RETIRED.RETIRE_SLOTS} + 4 * \text{INT_MISC.RECOVERY_CYCLES})} / (\text{CPU_CLK_UNHALTED.THREAD} * 4)$$

Threshold: Investigate if -
Cost is $\geq .2$

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. Try and determine which code path led to this destination.

Machine Clears

Why: Machine clears cause the pipeline to be flushed and the store buffers emptied, resulting in a significant latency penalty.

How: *General Exploration Profile, Metric: Machine Clears*

Now What: If this metric is highlighted for your hotspot try to determine the cause using the specific events:

- If `MACHINE_CLEARS.MEMORY_ORDERING` is significant, investigate at the source code level. This could be caused by 4K aliasing conflicts or contention on a lock (both previous issues).
- If `MACHINE_CLEARS.SMC` is significant, the clears are being caused by self-modifying code, which should be avoided.

Threshold: Investigate if -
Cost is $\geq .02$

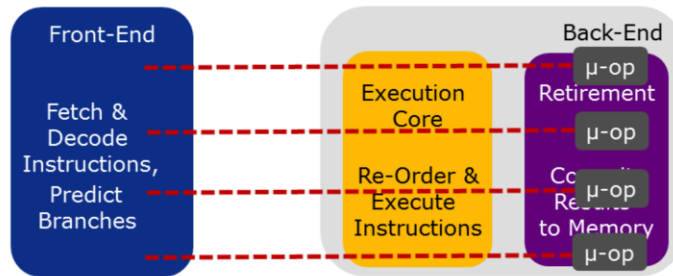
Machine clears are generally caused by either contention on a lock, or failed memory disambiguation from 4k aliasing (both earlier issues). The other potential cause is self-modifying code (SMC).

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...)

Retiring, Illustrated



Occurs when a pipeline slot is filled with a micro-op that retired. The desirable category! But some issues are still possible.

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 for more on this. The second is whether microcode assists can be eliminated from the instruction stream.

Retiring Hierarchical Breakdown in Amplifier XE

Retiring	Front-End Bound	Bad Speculation	Back-End Bound
14.3%	7.0%	3.9%	74.8%
16.6%	5.7%	4.5%	73.1%
12.5%	8.0%	4.5%	75.0%
48.9%	22.1%	36.3%	0.0%
23.2%	7.7%	0.0%	84.6%
4.5%	4.5%	0.0%	91.0%

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.

Retiring		
General Retirement		
FP Arithmetic	Other	Microcode Sequencer
0.140	0.860	0.000
0.192	0.808	0.000
0.000	1.000	0.000
0.000	1.000	0.000

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



FP Arithmetic

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

How: *General Exploration Profile*, Metrics: *FP Arithmetic, FP x87, FP Scalar, FP Vector*

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

Formula:

FP x87 % = UOPS_EXECUTED.X87 / UOPS_EXECUTED.THREAD

FP Scalar % = (FP_ARITH_INST_RETIRED.SCALAR_SINGLE +
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE) / UOPS_RETIRED.RETIRE_SLOTS

FP Vectory % = (FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE +
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE +
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE +
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE) /
UOPS_RETIRED.RETIRE_SLOTS

These metrics represent a breakdown of each type of instruction (x87, Scalar, Vector) as a percentage of all retired uops. Try to improve vectorization to increase the FP Vector percentage and decrease the x87 and FP Scalar percentages.

Microcode Assists

Why: Assists from the microcode sequencer can have long latency penalties.

How: *General Exploration Profile, Metric: Microcode Sequencer*

What Now: If this metric is highlighted for your hotspot, re-sample using the additional assist events to determine the cause.

- If FP_ASSISTS.ANY / INST_RETIRED.ANY is significant, check for denormals. To fix enable FTZ and/or DAZ if using SSE/AVX instructions or scale your results up or down depending on the problem
- If $((\text{OTHER_ASSISTS.AVX_TO_SSE_NP} * 75) / \text{CPU_CLK_UNHALTED.THREAD})$ or $((\text{OTHER_ASSISTS.SSE_TO_AVX_NP} * 75) / \text{CPU_CLK_UNHALTED.THREAD})$ is greater than .1, reduce transitions between SSE and AVX code. See <http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties>

Formula:

$$\text{Assist \%} = (\text{UOPS_RETIRED.RETIRE_SLOTS} / \text{UOPS_ISSUED.ANY}) * (\text{IDQ.MS_UOPS} / (\text{CPU_CLK_UNHALTED.THREAD} * 4))$$

Threshold: Investigate if –
Assist Cost \geq .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

Additional Topic: Memory Access

Why: Memory bandwidth bottlenecks increase the latency at which cache misses are serviced

How: *Memory Access Profile*

What Now:

- Compute the maximum theoretical memory bandwidth per socket for your platform in GB/s: $(\text{<MT/s> * 8 \text{ Bytes/clock} * \text{<num channels>}) / 1000$
- Run bandwidth analysis on your application. If total bandwidth per socket is > 75% of the maximum theoretical bandwidth, your application may be experiencing loaded (higher) latencies
- If appropriate, make system tuning adjustments (upgrading / balancing DIMMs, disabling HW prefetchers)
- Reduce bandwidth usage if possible: remove ineffective SW prefetches, make algorithmic changes to reduce data storage/sharing, reduce data updates, and balance memory access across sockets

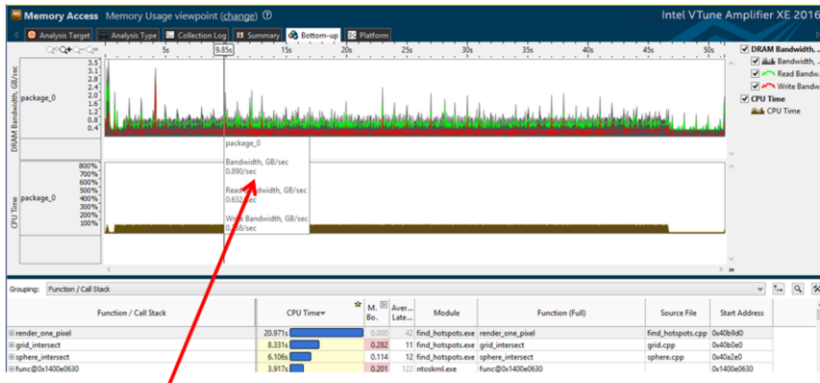
Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

Optimization Notice



Max theoretical bandwidth, per socket, for processor with DDR 1600 and 4 memory channels: 51.2 GB/s

Additional Topic: Memory Bandwidth



View both total, read, and write memory bandwidth per socket, over time.

Additional Topic: TSX Exploration

The screenshot displays the Intel VTune Performance Analyzer interface. On the left, the 'Choose Target and Analysis Type' panel shows a tree view of analysis types, with 'TSX Exploration' selected under 'Microarchitecture Analysis'. A red box highlights this selection with the text 'Select TSX Exploration Analysis Type'. On the right, the 'TSX Exploration' summary panel shows the following data:

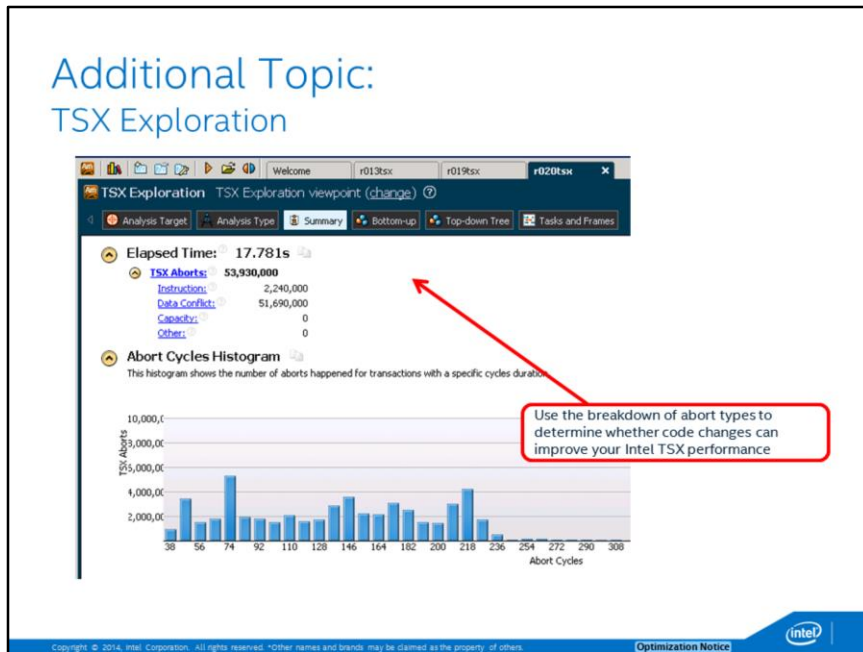
Metric	Value
Elapsed Time	17.844s
Clockticks	163,286,159,575
Transactional Cycles	15,842,125,660
Abort Cycles	5,342,553,597
Abort Cycles (%)	33.724

Red arrows point from the 'Abort Cycles' and 'Abort Cycles (%)' values to a red box containing the text 'Cycles spent within a TSX section that was aborted'. Another red arrow points from the 'Transactional Cycles' value to a red box containing the text 'Cycles spent within a TSX section'. The bottom of the screenshot shows the Intel logo and 'Optimization Notice'.

Intel® Transactional Synchronization Extensions (Intel® TSX) provide hardware transactional memory support. They expose a speculative execution mode to the programmer to improve locking performance. For more detailed information about developing software with Intel TSX see <http://www.intel.com/software/tsx>.

A large percentage of aborted cycles may represent a negative performance impact from the use of Intel TSX. Use this Analysis Type along with other performance metrics like elapsed time, CPI, or Retiring Percentage to measure how Intel TSX is affecting your performance.

Additional Topic: TSX Exploration



For a detailed description of Intel® TSX performance recommendations, see Chapter 12 of the Intel® 64 and IA-32 Architectures Optimization Reference Manual

Additional Topic: Metric Reliability

General Exploration General Exploration viewpoint (change) ⓘ

Collection Log Analysis Target Analysis Type Summary Bottom-up PMU Events Platform

Grouping: Function / Call Stack

Function / Call Stack	Clocktic...	Instructions Retired	CPI Rate	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)	
				Retiring	Bad Speculation	Back-End Bound	Front-End Bound
grid_intersect	14,076,021,114	12,468,018,702	1.129	0.210	0.076	0.650	0.063
sphere_intersect	9,306,013,959	9,206,013,809	1.011	0.282	0.038	0.615	0.065
grid_bounds_intersect	1,098,001,647	690,001,035	1.591	0.123	0.020	0.781	0.075
func@0:1002e3d5	922,001,383	700,001,050	1.317	0.000	0.000	1.000	0.000
_kmp_x86_pause	354,000,531	212,000,318	1.670	0.000	0.000	1.000	0.000
tri_intersect	222,000,333	152,000,228	1.461	0.405	0.000	0.561	0.101
pos2grid	212,000,318	186,000,279	1.140	0.248	0.000	0.717	0.035
pos2grid	14,076,021,114	12,468,018,702	1.129	0.210	0.076	0.650	0.063

Selected 1 row(s):

Grayed out metric values represent low reliability of the metrics for each value in the grid.

Copyright © 2014, Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others. Optimization Notice intel

The General Exploration analysis type multiplexes hardware events during collection, which can result in imprecise results if too few samples are collected. The GUI will gray out metrics if the reliability is low based on the number of samples collected. If a metric is grayed out for your area of interest, consider increasing the runtime of the analysis or allowing multiple runs via the project properties.

Previous versions of the tool used a MUX Reliability metric for each row, however this was unable to distinguish between different metrics on the same row.

Good Luck!

For more information:

VTune Amplifier XE Videos, Forums, and Resources:

<http://software.intel.com/en-us/intel-vtune-amplifier-xe/#pid-3659-760/>

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

<http://www.intel.com/products/processor/manuals/index.htm>

VTune Amplifier XE Tuning Guides for Other microarchitectures:

<http://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers>

For optimization of the integrated graphics controller:

www.intel.com/software/gpa

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'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies 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 [here](#).

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and, for some uses, certain computer system software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations and may require a BIOS update. Software applications may not be compatible with all operating systems. Please check with your application vendor.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with 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 overall system configuration. Check with your PC manufacturer on whether your system delivers Intel Turbo Boost Technology. For more information, see <http://www.intel.com/technology/turboboost>.

Intel, the Intel logo, Xeon, Xeon Inside, VTune, inTru, 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

Copyright © 2014, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Optimization Notice



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

