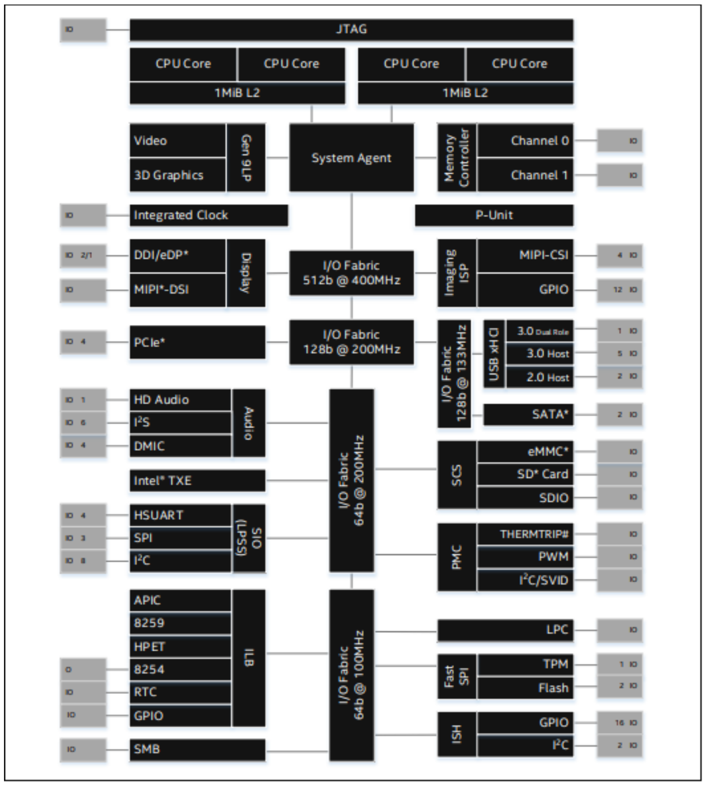
How to Use This Guide

This guide is intended for software developers interested in optimizing their application's performance on the Intel Atom® Processor E3900 Series, and Intel® Pentium® and Celeron® Processor N- and J-Series, using the Intel® VTune™ Amplifier performance profiler. Familiarity with Intel® VTune™ Amplifier or expertise/experience in performance optimization is not necessary, though familiarity with the application being optimized is strongly recommended. While much of the performance information in this guide applies equally to other tools, this document focuses on the use of Intel® VTune™ Amplifier.

The recommended usage model for this tuning guide is to read through it once before beginning the tuning process to familiarize yourself with the steps, then follow it again, step by step, as you work through optimizing your application. You may need to go through the process more than once to fully tune your code.

Before you begin the optimization process, you should make sure that you have used the appropriate compiler optimization flags for the architecture and chosen an appropriate workload for your application. It is also generally beneficial to measure the baseline performance of the program before beginning data collection or optimization.



This image represents a generalized processor layout intended to help illustrate the concepts described in this guide. It is not a definitive representation of the

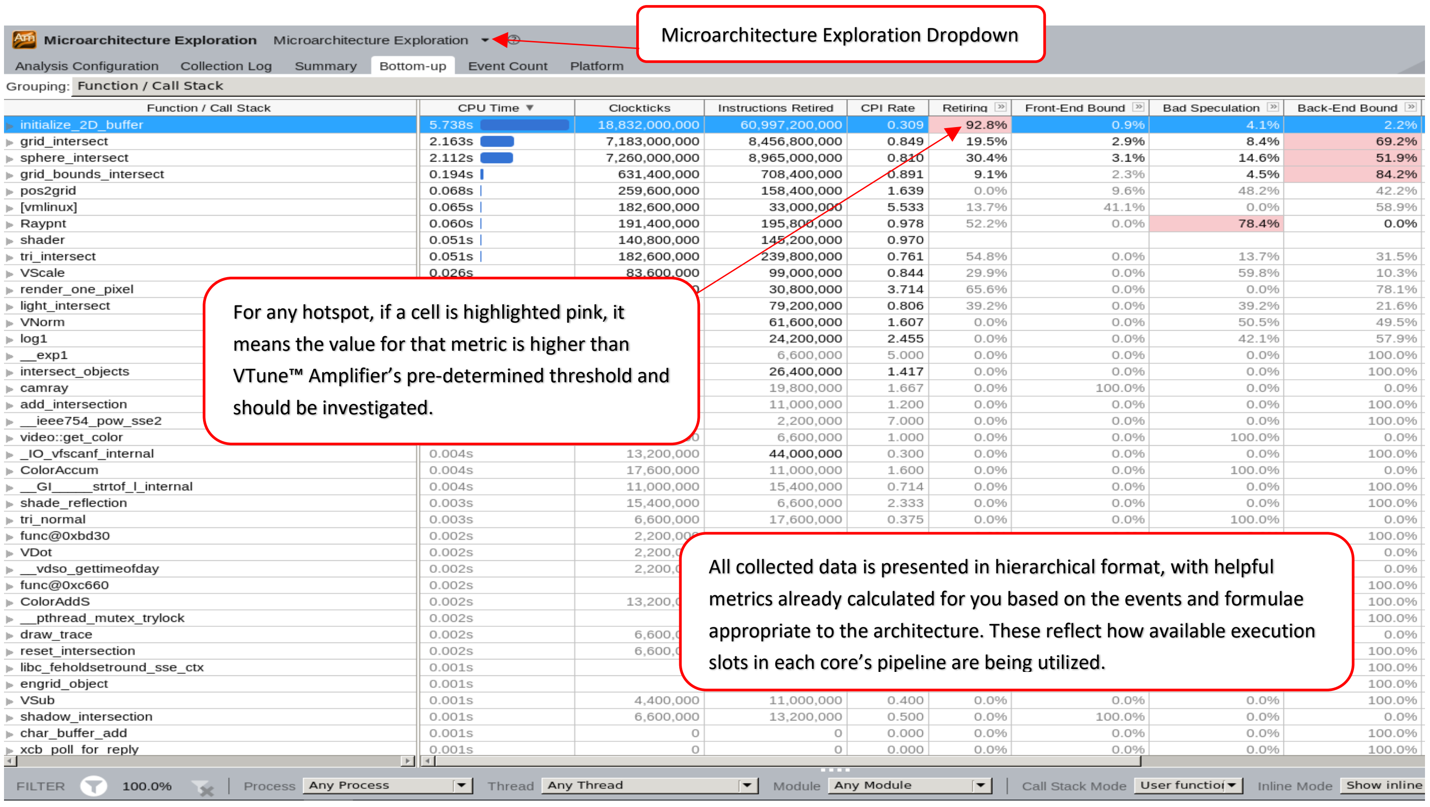
Some features present in the Intel Atom® Processor E3900 Series, and Intel® Pentium® and Celeron® Processor N- and J-Series can have significant effects on performance measurement, and make the process of measuring and interpreting performance data more complex.

WARNING! Incorrectly modifying BIOS settings from those supplied by the manufacturer may render the system unusable, and may void the warranty. You should contact the system vendor or manufacturer for specifics before making any changes.

About Intel® VTune™ Amplifier

Intel® VTune™ Amplifier is a versatile performance analysis tool available as a standalone product or as part of suites like Intel® Parallel Studio XE and Intel® System Studio. It can be run on Windows* and Linux* operating systems via command line, Graphical User Interface (GUI), or integration with Microsoft* Visual Studio*. Data can be viewed on macOS* systems as well. VTune™ Amplifier is compatible with multiple programming languages, including C/C++, C#, Google Go*, Java, Assembly, Python, and more.

Intel® VTune™ Amplifier contains several pre-configured analysis types like hotspots, memory access, threading and microarchitecture. This guide will focus primarily on the microarchitecture analysis. No familiarity with the hardware events is necessary, as the pre-configured analysis types are already set to collect the appropriate hardware events for your microarchitecture.



Microarchitecture Exploration Dropdown

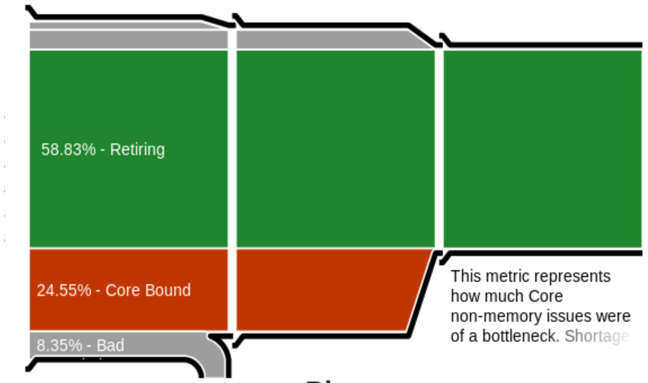
Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Speculation	Back-End Bound
initialize_2D_buffer	5.738s	18,832,000,000	60,997,200,000	0.309	92.8%	0.9%	4.1%	2.2%
grid_intersect	2.163s	7,183,000,000	8,456,800,000	0.849	19.5%	2.9%	8.4%	69.2%
sphere_intersect	2.112s	7,260,000,000	8,965,000,000	0.810	30.4%	3.1%	14.6%	51.9%
grid_bounds_intersect	0.194s	631,400,000	708,400,000	0.891	9.1%	2.3%	4.5%	84.2%
pos2grid	0.068s	259,600,000	158,400,000	1.639	0.0%	9.6%	48.2%	42.2%
[vmlinux]	0.065s	182,600,000	33,000,000	5.533	13.7%	41.1%	0.0%	58.9%
Raypnt	0.060s	191,400,000	195,800,000	0.978	52.2%	0.0%	78.4%	0.0%
shader	0.051s	140,800,000	145,200,000	0.970				
tri_intersect	0.051s	182,600,000	239,800,000	0.761	54.8%	0.0%	13.7%	31.5%
VScale	0.026s	83,600,000	99,000,000	0.844	29.9%	0.0%	59.8%	10.3%
render_one_pixel			30,800,000	3.714	65.6%	0.0%	0.0%	78.1%
light_intersect			79,200,000	0.806	39.2%	0.0%	39.2%	21.6%
VNorm			61,600,000	1.607	0.0%	0.0%	50.5%	49.5%
log1			24,200,000	2.455	0.0%	0.0%	42.1%	57.9%
__exp1			6,600,000	5.000	0.0%	0.0%	0.0%	100.0%
intersect_objects			26,400,000	1.417	0.0%	0.0%	0.0%	100.0%
camray			19,800,000	1.667	0.0%	100.0%	0.0%	0.0%
add_intersection			11,000,000	1.200	0.0%	0.0%	0.0%	100.0%
__ieee754_pow_sse2			2,200,000	7.000	0.0%	0.0%	0.0%	100.0%
video::get_color			6,600,000	1.000	0.0%	0.0%	100.0%	0.0%
_IO_vfscanf_internal	0.004s	13,200,000	44,000,000	0.300	0.0%	0.0%	0.0%	100.0%
ColorAccum	0.004s	17,600,000	11,000,000	1.600	0.0%	0.0%	100.0%	0.0%
__GI_____strtof_l_internal	0.004s	11,000,000	15,400,000	0.714	0.0%	0.0%	0.0%	100.0%
shade_reflection	0.003s	15,400,000	6,600,000	2.333	0.0%	0.0%	0.0%	100.0%
tri_normal	0.003s	6,600,000	17,600,000	0.375	0.0%	0.0%	100.0%	0.0%
func@0xbd30	0.002s	2,200,00						100.0%
VDot	0.002s	2,200,0						0.0%
__vdso_gettimeofday	0.002s	2,200,0						0.0%
func@0xc660	0.002s							100.0%
ColorAddS	0.002s	13,200,0						100.0%
__pthread_mutex_trylock	0.002s							100.0%
draw_trace	0.002s	6,600,0						0.0%
reset_intersection	0.002s	6,600,0						100.0%
libc_feholdsetround_sse_ctx	0.001s							100.0%
engrid_object	0.001s							100.0%
VSub	0.001s	4,400,000	11,000,000	0.400	0.0%	0.0%	0.0%	100.0%
shadow_intersection	0.001s	6,600,000	13,200,000	0.500	0.0%	100.0%	0.0%	0.0%
char_buffer_add	0.001s	0	0	0.000	0.0%	0.0%	0.0%	100.0%
xcb_poll_for_reply	0.001s	0	0	0.000	0.0%	0.0%	0.0%	100.0%

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

All collected data is presented in hierarchical format, with helpful metrics already calculated for you based on the events and formulae appropriate to the architecture. These reflect how available execution slots in each core's pipeline are being utilized.

Process: Any Process | Thread: Any Thread | Module: Any Module | Call Stack Mode: User functioi | Inline Mode: Show inline


58.83% - Retiring
24.55% - Core Bound
8.35% - Bad
This metric represents how much Core non-memory issues were of a bottleneck. Shortage

µPipe

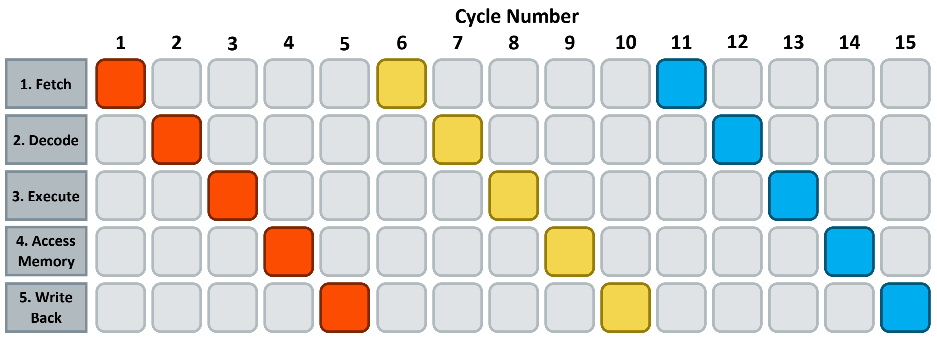
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

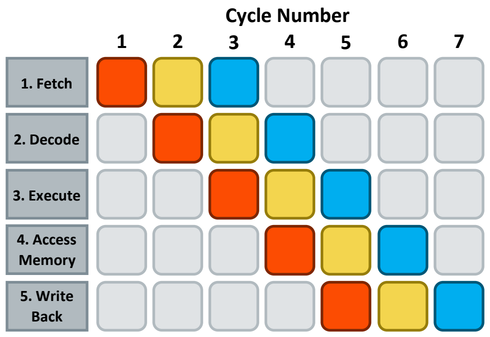
Most screenshots in this guide were taken in Intel® VTune™ Amplifier 2019. They may not necessarily have been taken on the microarchitecture this guide is written for. Screenshots taken in different versions of this tool may have minor differences.

The uOp Pipeline

The tuning methodology described in this guide relies on the concept of uop pipeline slot categorization. A uop (or more properly a µop) is a micro-operation, a low-level instruction such as a single addition, load, or less-than comparison. There are several steps in performing this operation – the uop must be fetched, decoded, executed, etc.

In this simplified example, processing an instruction involves five steps, each of which takes one cycle. Without pipelining, the red instruction must be completely processed before beginning the yellow instruction, which also must be finished before moving on to the blue instruction. To process all three in this fashion takes fifteen cycles.

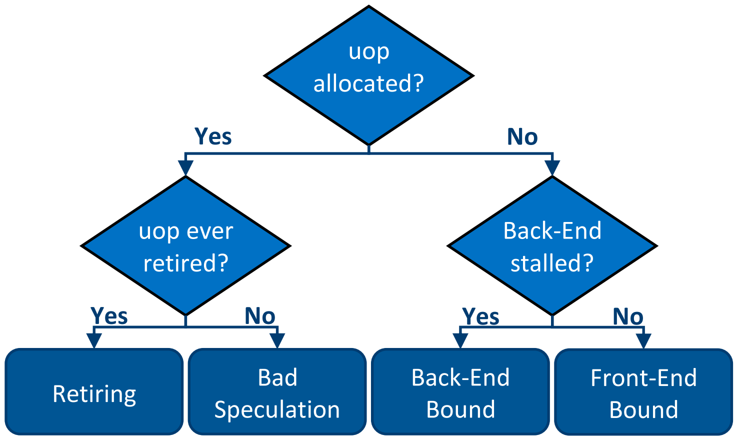




To improve efficiency, modern computers pipeline the uops: because the different steps in processing an instruction are handled by different sections of hardware, they can process multiple instructions at once. For instance, in cycle 3 in this diagram, it's fetching the blue instruction, decoding the yellow instruction, and executing the red one. All three are done in seven cycles.

This may be compared to washing a second load of laundry while the previous load is in the dryer. The part of the CPU which fetches and decodes is referred to as the Front-End, and the part which executes and retires the instruction is called the Back-End.

The pipeline slot is an abstract concept representing the hardware resources required to process one uop. Because the Front-End and Back-End can only process so many uops in a given amount of time, there's a fixed number of available pipeline slots. On this architecture, there are four pipeline slots available per cycle on each core. Each slot can be classified into one of four categories on any given cycle by what happens to the uop in that slot.

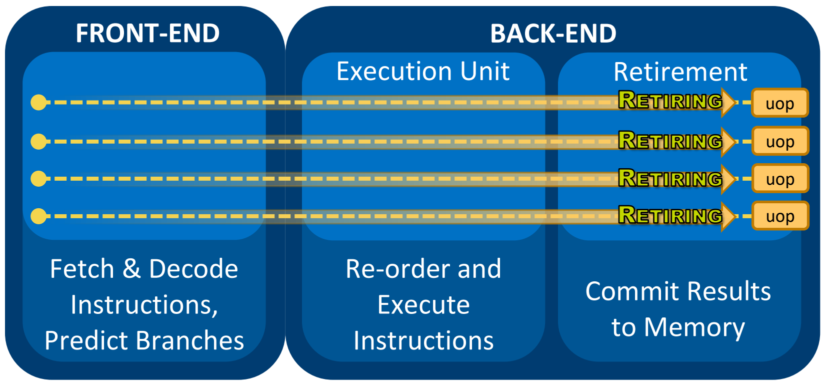


Each pipeline slot category is expected to fall within a particular percentage range for a well-tuned application of a given type. These ranges are detailed in the table below.

Note that for all categories but Retiring, lower numbers are better, and for Retiring, higher numbers are welcome. These values are simply the normal ranges one can expect for a well-tuned application based on its type.

	Application Type		
Category	**Client/Desktop**	**Server/Database/Distributed**	**High Performance Computing**
Retiring	20-50%	10-30%	30-70%
Bad Speculation	5-10%	5-10%	1-5%
Front-End Bound	5-10%	10-25%	5-10%
Back-End Bound	20-40%	20-60%	20-40%

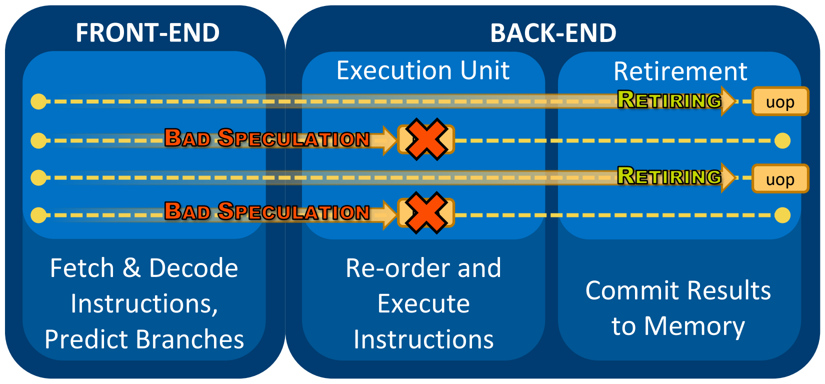
Retiring



This category represents pipeline slots filled with uops that successfully finish executing and retire. In general, it is desirable to have as many pipeline slots retiring per cycle as possible. However, there are still possible inefficiencies in this category, mostly concerning doing more work than is actually necessary.

See the section on tuning Retiring for more information.

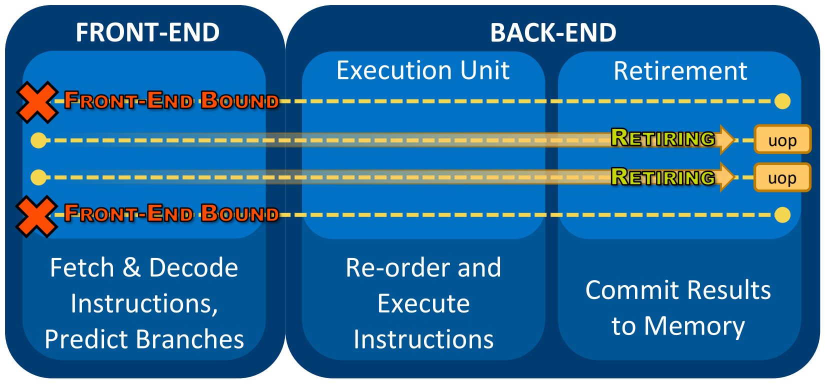
Bad Speculation



This category represents uops being removed from the Back-End without retiring. This effectively means that the uop is cancelled, and any time spent processing it has been wasted. This happens most often when a branch is mispredicted, and the partially-processed uops from the incorrect branch must be thrown out.

See the section on tuning Bad Speculation for more information.

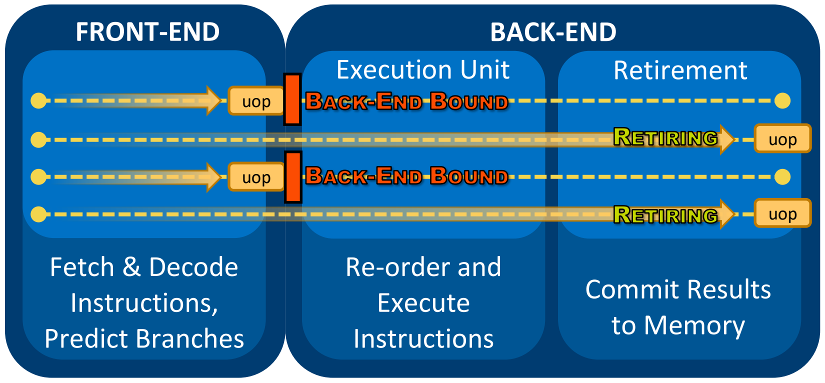
Front-End Bound



This category refers to cycles on which the Front-End could not deliver uops to a pipeline slot or slots, even though the Back-End was able to take them. This often occurs due to delays in fetching or decoding instructions. Using the laundry metaphor, the dryer is empty but the washer isn't finished yet.

See the section on tuning Front-End Bound for more information.

Back-End Bound

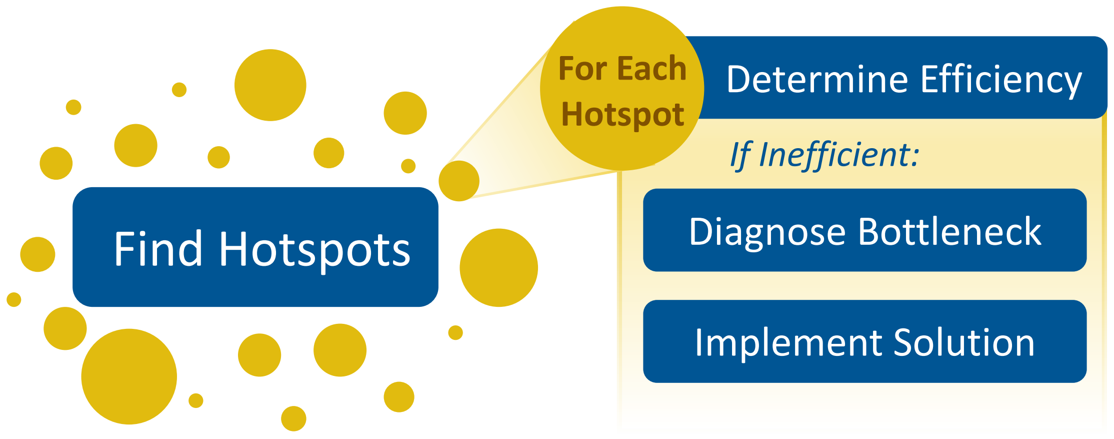


This category refers to cycles on which the Back-End couldn't accept uops in a pipeline slot or slots. This usually occurs because the Back-End is already occupied by uops waiting on data or taking longer to execute. Using the laundry metaphor, the washer is done, but the dryer is still running and can't accept a new load yet.

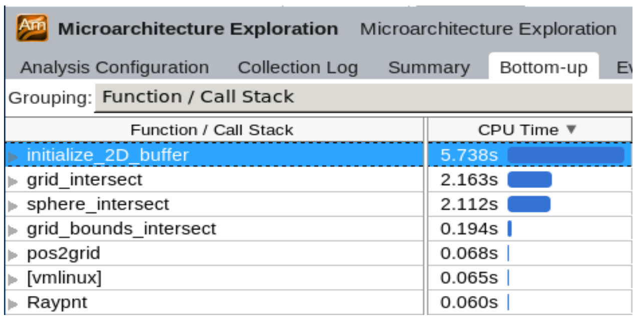
See the section on tuning Back-End Bound for more information.

Tuning Software for Specific Hardware Microarchitecture

The tuning process makes use of pipeline slot categorizations to focus optimizations on the bottlenecks with the greatest impact, as measured on the particular hardware architecture the software is intended for.



Find hotspots



Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼
initialize_2D_buffer	5.738s
grid_intersect	2.163s
sphere_intersect	2.112s
grid_bounds_intersect	0.194s
pos2grid	0.068s
[vmlinux]	0.065s
Raypnt	0.060s

The first step of the tuning process is to identify the hotspots – the sections of code your application spends the most time in.

The more time a function or loop takes, the more impact optimization in that chunk of code will have, in accordance with Amdahl's Law, which states that the total speedup of a task due to an improvement is limited by the proportion of that task which is actually affected by the improvements being made.

To find your hotspots using Intel® VTune™ Amplifier, run a Hotspot or Microarchitecture analysis.

Hotspots are generally defined in terms of clockticks. On this processor family, the `CPU_CLK_UNHALTED.REF_TSC` counter measures unhalted clockticks on a per-hardware-thread basis, at reference frequency. This allows you to see where cycles are being spent on each individual hardware thread. There is no per-core clocktick counter available on this processor, unlike some earlier processors.

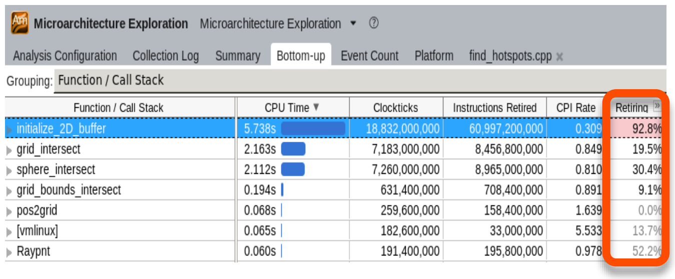
Once you have identified your hotspots, you can proceed through the rest of the process for each one: determine whether it is inefficient, and if so, determine the bottleneck, identify the cause, and optimize the code.

Determine Efficiency

A hotspot is defined in terms of the proportion of time the program spends in it, and may not necessarily indicate an inefficiency. Sometimes a hotspot is as well-optimized as it can be, but due to the nature of the algorithm, spending much of the program's time there is simply inevitable. Therefore, it is critical to not only identify the hotspots but evaluate whether they are efficient or not. There are several methods for determining a hotspot's efficiency.

Method 1: Retiring Slots

One of the simplest methods of determining efficiency is to check the percentage of pipeline slots that are retiring. Check the Retiring metric for your hotspot. If more than 70% of the pipeline slots are retiring, it may be beneficial to examine the code for evidence of performing unnecessary work, as described in Method 3.



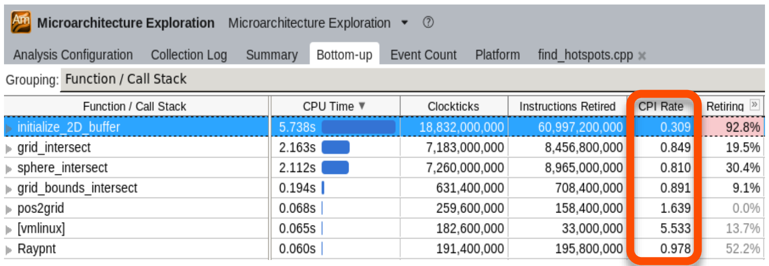
Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring
initialize_2D_buffer	5.738s	18,832,000,000	60,997,200,000	0.309	92.8%
grid_intersect	2.163s	7,183,000,000	8,456,800,000	0.849	19.5%
sphere_intersect	2.112s	7,260,000,000	8,965,000,000	0.810	30.4%
grid_bounds_intersect	0.194s	631,400,000	708,400,000	0.891	9.1%
pos2grid	0.068s	259,600,000	158,400,000	1.639	0.0%
[vmlinux]	0.065s	182,600,000	33,000,000	5.533	13.7%
Raypnt	0.060s	191,400,000	195,800,000	0.978	52.2%

Percent Retiring by Application Type		
Client/Desktop	Server/Database/ Distributed	High Performance Computing
20-50%	10-30%	30-70%

Otherwise, compare the observed value with the expected range for Retiring slots in your application type. If the hotspot is below the expected range, it is likely inefficient.

Method 2: CPI Changes



Microarchitecture Exploration — Bottom-up — Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Retiring
initialize_2D_buffer	5.738s	18,832,000,000	60,997,200,000	0.309	92.8%
grid_intersect	2.163s	7,183,000,000	8,456,800,000	0.849	19.5%
sphere_intersect	2.112s	7,260,000,000	8,965,000,000	0.810	30.4%
grid_bounds_intersect	0.194s	631,400,000	708,400,000	0.891	9.1%
pos2grid	0.068s	259,600,000	158,400,000	1.639	0.0%
[vmlinux]	0.065s	182,600,000	33,000,000	5.533	13.7%
Raypnt	0.060s	191,400,000	195,800,000	0.978	52.2%

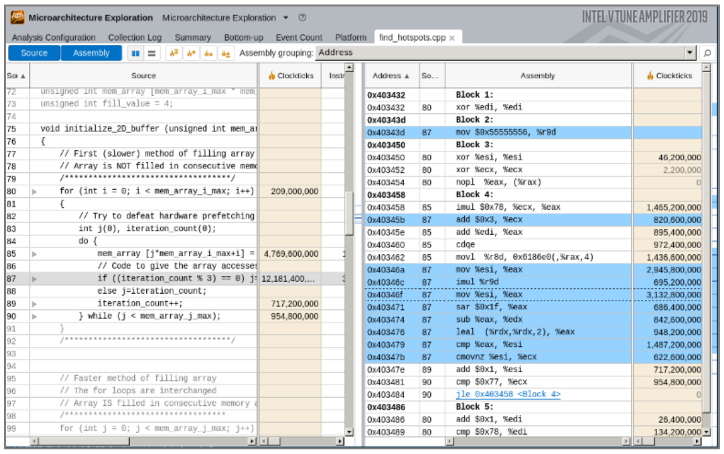
Another measurement of performance is Cycles Per Instruction (CPI), the average time instructions in your workload take to execute. CPI is a general efficiency metric, most useful for comparing sets of data, and is not a robust indicator of inefficiency in and of itself. The Intel® VTune™ Amplifier interface highlights CPI if it exceeds 1, as some well-tuned applications achieve CPIs of 1 or below, but many applications naturally have CPIs exceeding 1. It is highly dependent on workload and platform.

Because of this, changes in CPI between runs are often more useful as (very general) indicators than the CPI values themselves. Usually, optimizations lowering CPI are good and those raising it are bad, but there are exceptions. Because CPI is a ratio of cycles per instruction, it will change when the code size changes. For the same reason, it is possible to have a very low CPI and still be inefficient because more work is being done than is actually needed.

Relatedly, the presence of AVX instructions may increase the CPI and the stall percentage, but still improve the performance, because one vector instruction takes longer to execute than one scalar instruction, but does far more useful work in that time. This is discussed in more detail in Method 3.

Method 3: Code Study

While Methods 1 and 2 measure how long it takes for instructions to execute, that is not the only type of inefficiency. Code can also be inefficient if it does unnecessary work. This can often result from failure to make use of modern instructions. Method 3 makes use of Intel® VTune™ Amplifier's capability as a source and disassembly viewer allows you to check your code for this form of inefficiency. The source/disassembly viewer can be accessed from any analysis type by double clicking on a function name. This will open a code view tab already scrolled to the appropriate location in the code. Source and Assembly can be toggled independently using buttons in the upper left.



Microarchitecture Exploration — find_hotspots.cpp — Assembly grouping: Address

So...	Source	Clockticks
72	unsigned int mem_array [mem_array_i_max * mem	
73	unsigned int fill_value = 4;	
74		
75	void initialize_2D_buffer (unsigned int mem_ar	
76	{	
77	// First (slower) method of filling array	
78	// Array is NOT filled in consecutive mem	
79	/**********************************/	
80	for (int i = 0; i < mem_array_i_max; i++)	209,000,000
81	{	
82	// Try to defeat hardware prefetching	
83	int j(0), iteration_count(0);	
84	do {	
85	mem_array [j*mem_array_i_max+i] =	4,769,600,000
86	// Code to give the array accesse	
87	if ((iteration_count % 3) == 0) j=	12,181,400,...
88	else j=iteration_count;	
89	iteration_count++;	717,200,000
90	} while (j < mem_array_j_max);	954,800,000
91	}	
92	/**********************************/	
93		
94		
95	// Faster method of filling array	
96	// The for loops are interchanged	
97	// Array IS filled in consecutive memory a	
98	/**********************************	
99	for (int j = 0; j < mem_array_j_max; j++)	

Address	So...	Assembly	Clockticks
0x403432		Block 1:	
0x403432	80	xor %edi, %edi	
0x40343d		Block 2:	
0x40343d	87	mov $0x55555556, %r9d	
0x403450		Block 3:	
0x403450	80	xor %esi, %esi	46,200,000
0x403452	80	xor %ecx, %ecx	2,200,000
0x403454	80	nopl %eax, (%rax)	0
0x403458		Block 4:	
0x403458	85	imul $0x78, %ecx, %eax	1,465,200,000
0x40345b	87	add $0x3, %ecx	820,600,000
0x40345e	85	add %edi, %eax	895,400,000
0x403460	85	cdqe	972,400,000
0x403462	85	movl %r8d, 0x6186e0(,%rax,4)	1,436,600,000
0x40346a	87	mov %esi, %eax	2,945,800,000
0x40346c	87	imul %r9d	695,200,000
0x40346f	87	mov %esi, %eax	3,132,800,000
0x403471	87	sar $0x1f, %eax	686,400,000
0x403474	87	sub %eax, %edx	842,600,000
0x403476	87	leal (%rdx,%rdx,2), %eax	948,200,000
0x403479	87	cmp %eax, %esi	1,487,200,000
0x40347b	87	cmovnz %esi, %ecx	622,600,000
0x40347e	89	add $0x1, %esi	717,200,000
0x403481	90	cmp $0x77, %ecx	954,800,000
0x403484	90	jle 0x403458 <Block 4>	0
0x403486		Block 5:	
0x403486	80	add $0x1, %edi	26,400,000
0x403489	80	cmp $0x78, %edi	134,200,000

There are two particular types of modern instruction that are commonly missed out on. These are the latest vector instructions and fused multiply-add instructions.

Vector Instructions

Vector (or SIMD: Single Instruction Multiple Data) instructions can greatly increase performance by allowing multiple operations of the same type to be done at once – for instance, adding four numbers to four other numbers, instead of performing four separate add instructions. Over time, the available SIMD instructions have been expanded with new sets of instructions. If you're not making use of the latest set of SIMD instructions available on your architecture, you're missing out on the performance benefits they bring.

When looking through your code's assembly, especially in areas containing loops, look for instructions that are non-SIMD, or which are using outdated SIMD instructions. Not all code can be vectorized and it's not necessarily a problem when some older vector instructions are present among more recent instructions.

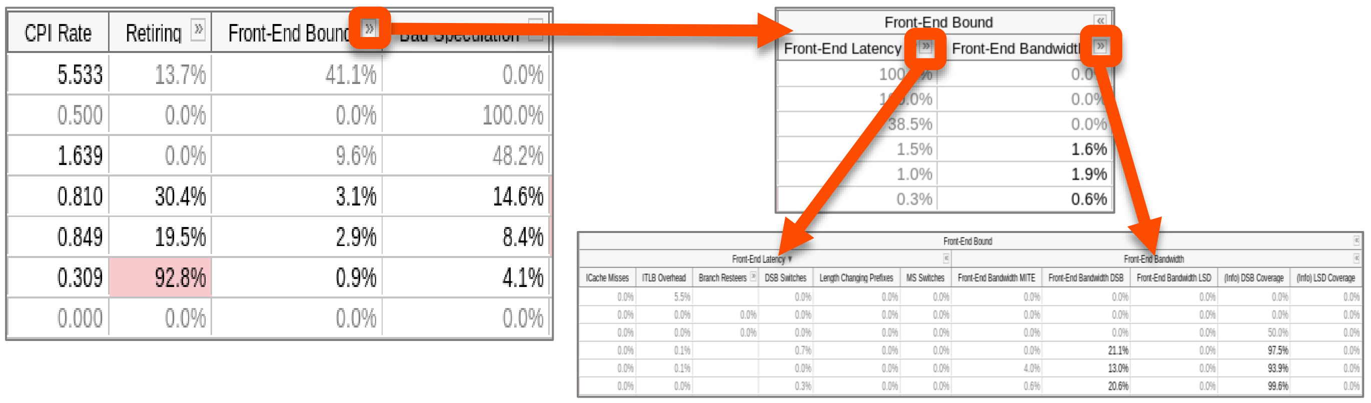
SIMD instructions can be recognized by their names. The table below lists them from oldest to newest.

Instruction Set	Identifiers
MMX	MMX instructions can be identified by the fact that they use the mmx registers. MMX instructions only operate on integers.
SSE	SSE instructions can be recognized by the two-character tag at the end of the instruction name. The second character is s, while the first character indicates whether it is scalar (non-SIMD) or packed (SIMD). For instance, addss is a scalar SSE add instruction, while the packed equivalent is addps. SSE instructions use the xmm registers.

Diagnose and Optimize the Bottleneck

Front-End Bound

For a conceptual explanation of the Front-End Bound category, see the appropriate uOp Pipeline entry.



CPI Rate	Retiring	Front-End Bound	Bad Speculation
5.533	13.7%	41.1%	0.0%
0.500	0.0%	0.0%	100.0%
1.639	0.0%	9.6%	48.2%
0.810	30.4%	3.1%	14.6%
0.849	19.5%	2.9%	8.4%
0.309	92.8%	0.9%	4.1%
0.000	0.0%	0.0%	0.0%

Front-End Bound	
Front-End Latency	Front-End Bandwidth
100[illegible]%	0.0[illegible]
1[illegible].0%	0.0%
38.5%	0.0%
1.5%	1.6%
1.0%	1.9%
0.3%	0.6%

Front-End Bound										
Front-End Latency						Front-End Bandwidth				
ICache Misses	ITLB Overhead	Branch Resteers	DSB Switches	Length Changing Prefixes	MS Switches	Front-End Bandwidth MITE	Front-End Bandwidth DSB	Front-End Bandwidth LSD	(Info) DSB Coverage	(Info) LSD Coverage
0.0%	5.5%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	50.0%	0.0%
0.0%	0.1%		0.7%	0.0%	0.0%	0.0%	21.1%	0.0%	97.5%	0.0%
0.0%	0.1%		0.0%	0.0%	0.0%	4.0%	13.0%	0.0%	93.9%	0.0%
0.0%	0.0%		0.3%	0.0%	0.0%	0.6%	20.6%	0.0%	99.6%	0.0%

The Front-End Bound category in VTune™ Amplifier expands into the Front-End Latency and Front-End Bandwidth categories, which display the percentage of Front-End Bound slots that fall into these sub-categories. Front-End Bound slots are counted toward Latency on cycles when no uops are being delivered at all, and toward Bandwidth on the cycles when uops are delivered in some slots, but not all.

If Front-End Bound is the primary bottleneck in your application, you should focus on Front-End Latency.

Front-End Latency

WHY OPTIMIZE THIS?

Front-End Latency can cause the Back-End to suffer from instruction starvation: not having enough uOps to execute.

ASSOCIATED METRICS

Front-End Bound
  └ Front-End Latency
    └ All Sub-Metrics

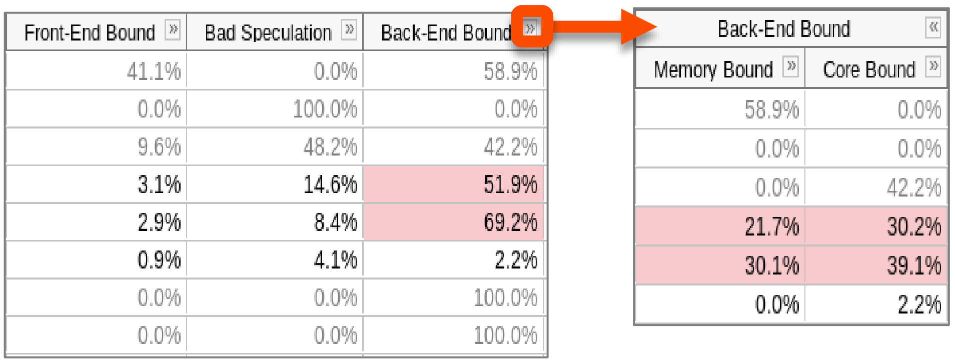
Front-End Latency indicates that you may have a problem with inefficient code layout or generation.

You may want to reduce your code size with switches like `/O1` or `/Os`, use linker ordering techniques (using `/ORDER` on Microsoft*'s linker or a linker script for gcc). You can also try Profile-Guided Optimizations (PGO) with your compiler.

For dynamically-generated code, try co-locating hot code, reducing code size, and avoiding indirect calls.

Back-End Bound

For a conceptual explanation of the Back-End Bound category, see the appropriate uOp Pipeline entry.



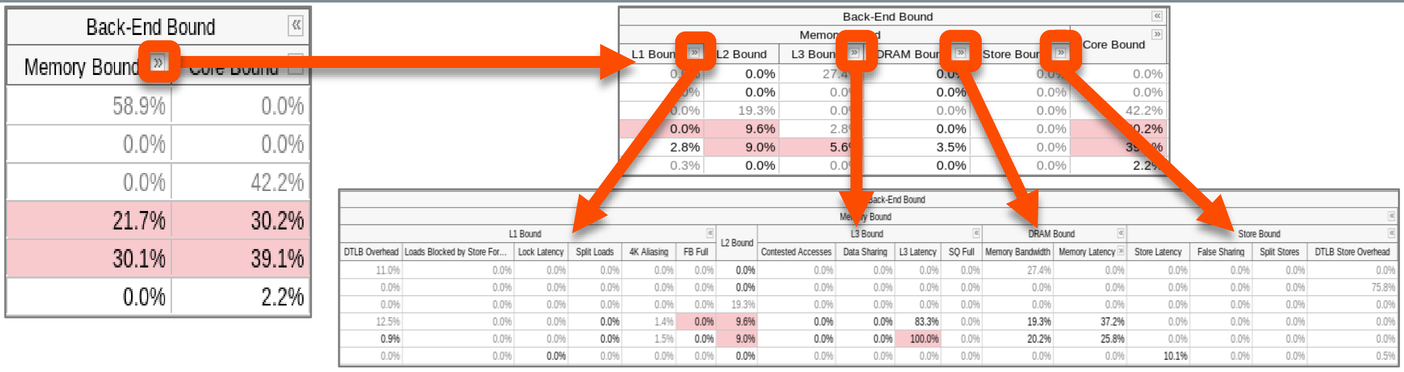
Front-End Bound	Bad Speculation	Back-End Bound
41.1%	0.0%	58.9%
0.0%	100.0%	0.0%
9.6%	48.2%	42.2%
3.1%	14.6%	51.9%
2.9%	8.4%	69.2%
0.9%	4.1%	2.2%
0.0%	0.0%	100.0%
0.0%	0.0%	100.0%

Back-End Bound	
Memory Bound	Core Bound
58.9%	0.0%
0.0%	0.0%
0.0%	42.2%
21.7%	30.2%
30.1%	39.1%
0.0%	2.2%

Expand the Back-End Bound category to see the Memory Bound and Core Bound sub-metric categories.

Memory Bound refers to cases where the Back-End could not accept new uops due to outstanding memory operations, while Core Bound refers to those where the issue is saturated execution ports.

Memory Bound



Back-End Bound	
Memory Bound	Core Bound
58.9%	0.0%
0.0%	0.0%
0.0%	42.2%
21.7%	30.2%
30.1%	39.1%
0.0%	2.2%

Back-End Bound					
Memory Bound					Core Bound
L1 Boun	L2 Bound	L3 Boun	DRAM Bour	Store Bour	
[illegible]	0.0%	27.4	0.0	0.0	0.0%
[illegible]	0.0%	0.0	0.0%	0.0%	0.0%
0.0%	19.3%	0.0	0.0%	0.0%	42.2%
0.0%	9.6%	2.8	0.0%	0.0%	0.2%
2.8%	9.0%	5.6	3.5%	0.0%	[illegible]
0.3%	0.0%	0.0	0.0%	0.0%	2.2

Back-End Bound																	
Memory Bound																	
L1 Bound						L2 Bound	L3 Bound				DRAM Bound		Store Bound				
DTLB Overhead	Loads Blocked by Store For...	Lock Latency	Split Loads	4K Aliasing	FB Full		Contested Accesses	Data Sharing	L3 Latency	SQ Full	Memory Bandwidth	Memory Latency	Store Latency	False Sharing	Split Stores	DTLB Store Overhead	
11.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	27.4%	0.0%	0.0%	0.0%	0.0%	0.0%	
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	75.8%	
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	19.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	
12.5%	0.0%	0.0%	0.0%	1.4%	0.0%	9.6%	0.0%	0.0%	83.3%	0.0%	19.3%	37.2%	0.0%	0.0%	0.0%	0.0%	
0.9%	0.0%	0.0%	0.0%	1.5%	0.0%	9.0%	0.0%	0.0%	100.0%	0.0%	20.2%	25.8%	0.0%	0.0%	0.0%	0.0%	
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	10.1%	0.0%	0.0%	0.5%	

The Memory Bound sub-category metrics indicate issues related to the various levels of the memory hierarchy.

Cache Misses

Why Optimize This?

Cache misses, especially higher-level misses, raise the CPI of an application.

Associated Metrics

Back-End Bound
 └ Memory Bound
  └ DRAM Bound

When optimizing applications with cache misses as a bottleneck, focus on the longer-latency accesses from last-level caches first.

First check for sharing issues, as these can cause cache misses. See the Contested Accesses section for more details. If the cache misses do not result from sharing issues, you may want to block your data accesses so that they fit into the cache, or change the algorithm to reduce data storage.

Under normal circumstances, writing to memory causes memory to be read as well. When a lot of data is being written and will not be used again soon, it can be beneficial to bypass the cache entirely using Non-temporal or Streaming Stores. When a lot of data is being read, software prefetches may be useful to ensure that data has already been loaded by the time it is actually needed, thus preventing the delay that results from a cache miss.

When vectorizing, align the data appropriately if possible, and include the appropriate clauses to inform the compiler.

Finally, you may wish to try the techniques like Cache-line Replacement Analysis outlined in section B.5.4.2 of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

Remote Memory Accesses

WHY OPTIMIZE THIS?

With Non-Uniform Memory Access (NUMA) architecture, remote loads have higher latency.

ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  └ DRAM Bound
    └ Memory Latency
      └ Remote DRAM

This metric indicates you need to improve your NUMA affinity. Note that it only measures remote *memory* (DRAM) accesses, and does not include data found in the cache in the remote socket. Also note that `Malloc()` and `VirtualAlloc()` do not touch memory. The operating system 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.

It's best to ensure that memory is first touched (accessed) by the thread that will be using it. If thread migration is a problem, try pinning or affinitizing threads to cores. For OpenMP*, you can use the affinity environment variable.

If possible, use NUMA-aware options for supporting applications (e.g. softnuma for SQL Server*), and use NUMA-efficient thread schedulers (such as Intel® Threading Building Blocks).

Contested Accesses (a.k.a. Write Sharing)

WHY OPTIMIZE THIS?

Sharing modified data among cores at L2 level can raise the latency of data access.

ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  └ Contested Accesses

This issue 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 bouncing back and forth 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 two 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, VTune™ Amplifier's Locks and Waits analysis should also indicate a problem. However, the Locks and Waits analysis will also detect other cases, such as false sharing or write sharing on a hot data structure.

If this metric is highlighted for your hotspot, locate the source code line(s) generating load uops retired where the cache line containing the data was in the modified state of another core or module's cache (HITM) by viewing the source. Use your knowledge of the code to determine whether real or false sharing is taking place.

- For real sharing, reduce sharing requirements.
- For false sharing, pad variables to cache line boundaries.

Blocked Loads Due to No Store Forwarding

WHY OPTIMIZE THIS?

If it is not possible to forward the result of a store through the pipeline, dependent loads may be blocked.

ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  └ Loads Blocked by Store Forwarding

Store forwarding occurs when two memory instructions, a store followed by a load from the same address, exist within the pipeline at the same time. Instead of waiting for the data to be stored to cache, it is usually "forwarded" through the pipeline directly to the load instruction. This prevents the load from having to wait for the memory to be written to the cache. However, in some cases, the store cannot be forwarded, and the load becomes blocked waiting for it to write to the cache and then load it.

If this metric is highlighted for your hotspot, view the source and look for the `LD_BLOCKS.STORE_FORWARD` event. This event usually tags to the next instruction after the attempted load that was blocked. Locate that load, and then try to find the store that cannot forward – usually this is within the prior 10 to 15 instructions. The most common case is that the store is to a smaller memory space than the load. In this case, the problem can be corrected by storing to the same size or a larger space as the load.

4K Aliasing

WHY OPTIMIZE THIS?

Aliasing conflicts result in having to re-issue loads.

ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  └ 4K Aliasing

If a load is issued after a store, and their memory addresses are offset by 4K, the address of the load will match the previous store in the pipeline, as the full address is not used at this point. The pipeline will try to forward the results of the store, but later, when the address of the load is fully resolved, it will no longer match. The load then has to be re-issued from a later point in the pipeline. This tends to have about a 7-cycle penalty, but in certain situations (such as with unaligned loads spanning two cache lines), it can be worse.

This issue can easily be resolved by changing the alignment of the load. Methods of correction include aligning data to 32 bytes, changing the offset between the input and output buffers if possible, or using 16-byte memory accesses on memory that is not 32-byte aligned.

DTLB Misses

WHY OPTIMIZE THIS?

First-level DTLB load misses incur a latency penalty. Second-level misses require a page walk that can affect application performance.

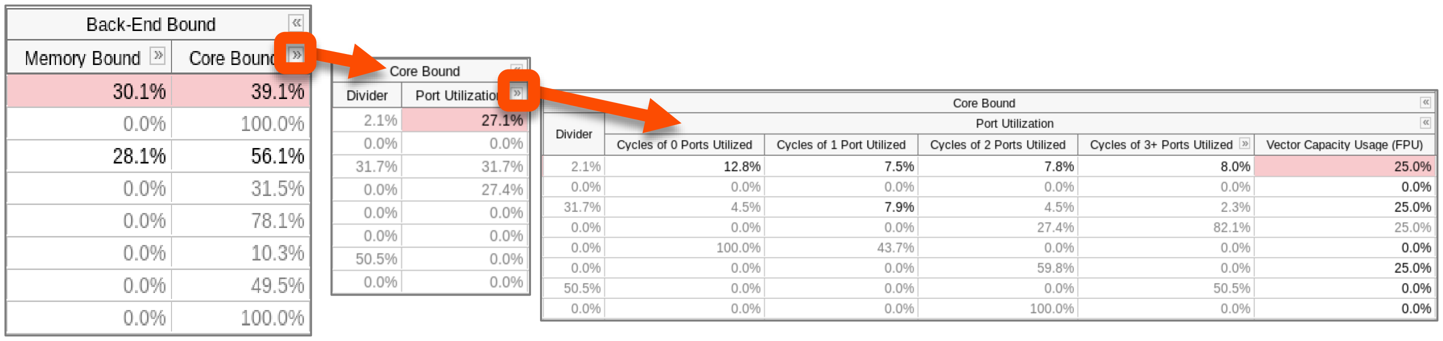
ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  └ DTLB Overhead

DTLB (Data Translation Lookaside Buffer) misses are more likely to occur with applications with a large random dataset.

To address this issue on database or server applications, try using large pages. On virtualized systems, use Extended Page Tables (EPT). You can also try to target data locality to the Translation Lookaside Buffer (TLB) size by blocking data and minimizing random access patterns. Finally, you can increase data locality by using Profile Guided Optimization (PGO) or better memory allocation.

Core Bound



Back-End Bound	
Memory Bound	Core Bound
30.1%	39.1%
0.0%	100.0%
28.1%	56.1%
0.0%	31.5%
0.0%	78.1%
0.0%	10.3%
0.0%	49.5%
0.0%	100.0%

Core Bound	
Divider	Port Utilization
2.1%	27.1%
0.0%	0.0%
31.7%	31.7%
0.0%	27.4%
0.0%	0.0%
0.0%	0.0%
50.5%	0.0%
0.0%	0.0%

Core Bound					
Divider	Port Utilization				
	Cycles of 0 Ports Utilized	Cycles of 1 Port Utilized	Cycles of 2 Ports Utilized	Cycles of 3+ Ports Utilized	Vector Capacity Usage (FPU)
2.1%	12.8%	7.5%	7.8%	8.0%	25.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
31.7%	4.5%	7.9%	4.5%	2.3%	25.0%
0.0%	0.0%	0.0%	27.4%	82.1%	25.0%
0.0%	100.0%	43.7%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	59.8%	0.0%	25.0%
50.5%	0.0%	0.0%	0.0%	50.5%	0.0%
0.0%	0.0%	0.0%	100.0%	0.0%	0.0%

The Core Bound category contains information relating to the execution core, including a breakdown of port utilization.

Divider

WHY OPTIMIZE THIS?

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

ASSOCIATED METRICS

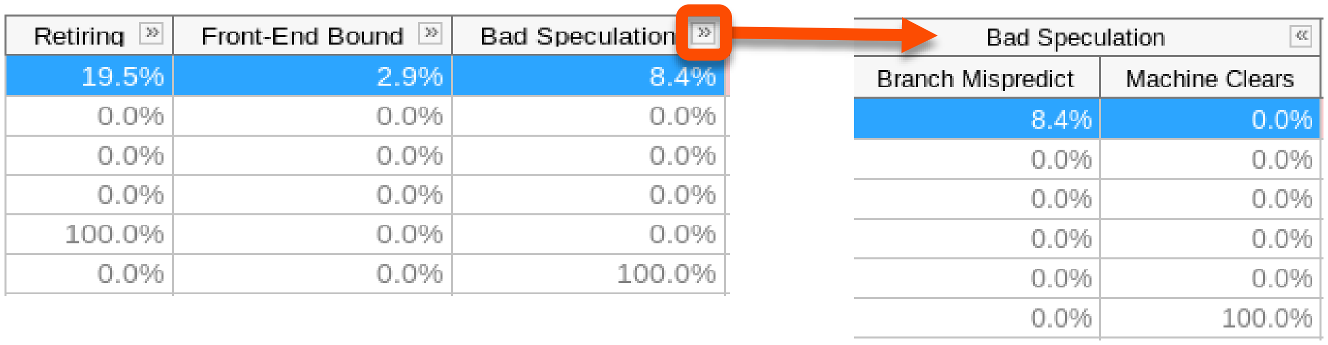
Back-End Bound
└ Core Bound
  └ Divider

Divide instructions are more expensive than other arithmetic instructions, so should be avoided where possible.

Ensure that the code is being compiled with optimizations turned on, vectorize divide instructions if you can, and if possible, use reciprocal multiplication (e.g. multiply by 0.5 instead of dividing by 2).

Bad Speculation

For a conceptual explanation of the Bad Speculation category, see the appropriate uOp Pipeline entry.



Retiring	Front-End Bound	Bad Speculation
19.5%	2.9%	8.4%
0.0%	0.0%	0.0%
0.0%	0.0%	0.0%
0.0%	0.0%	0.0%
100.0%	0.0%	0.0%
0.0%	0.0%	100.0%

Bad Speculation	
Branch Mispredict	Machine Clears
8.4%	0.0%
0.0%	0.0%
0.0%	0.0%
0.0%	0.0%
0.0%	0.0%
0.0%	100.0%

Speculation allows uops to begin executing before it is known whether that operation will retire. This allows the pipeline to continue working by making an educated guess rather than stalling and waiting until the correct path forward is known. Sometimes the speculated path turns out to be incorrect, and the speculated operations need to be cancelled.

This does not cause program incorrectness, as the incorrect instructions never complete, but it can cause inefficiency as time is wasted when the incorrect instructions are discarded and the pipeline starts over with the correct ones.

Branch Mispredicts

WHY OPTIMIZE THIS?

Mispredicted branches cause pipeline inefficiency due to wasted work and/or instruction starvation while waiting for the correct instructions to be fetched.

ASSOCIATED METRICS

Bad Speculation
└ Branch Mispredict

All applications that branch will have some branch mispredicts, so do not be alarmed when you see them in your application. Branch mispredicts are only a problem when they have a considerable performance impact.

Locating the origin of the branch mispredicts may be difficult, as the event normally tags to the first instruction in the correct path, rather than the abandoned incorrect path.

Methods of tuning include using compiler options and/or Profile Guided Optimization (PGO) to improve code generation, or hand-tuning branch statements, which can include techniques like hoisting the most popular targets. As branch misprediction requires a branch to (mis)predict, avoid unnecessary branching.

Machine Clears

Why Optimize This?
Machine clears flush the pipeline and empty store buffers, causing a significant latency penalty.
Associated Metrics
Bad Speculation
└ Machine Clears

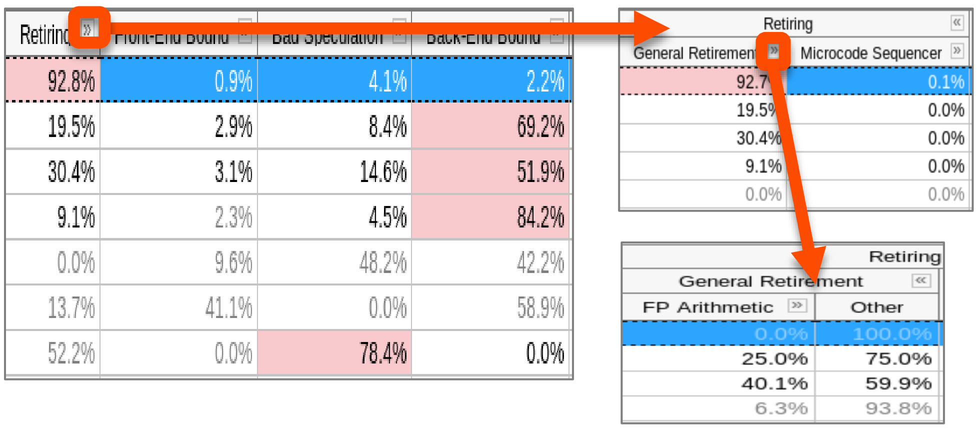
Machine clears are much rarer than branch mispredicts. These are generally caused by contention on a lock, failed memory disambiguation from 4K aliasing, or self-modifying code.

Try to identify the cause in your hotspot by looking for specific events.

`MACHINE_CLEARS.MEMORY_ORDERING` may indicate 4K aliasing conflicts or lock contention. `MACHINE_CLEARS.SMC` indicates the cause is self-modifying code, which should be avoided.

Retiring

For a conceptual explanation of the Retiring category, see the appropriate uOp Pipeline entry.



Retiring	Front-End Bound	Bad Speculation	Back-End Bound
92.8%	0.9%	4.1%	2.2%
19.5%	2.9%	8.4%	69.2%
30.4%	3.1%	14.6%	51.9%
9.1%	2.3%	4.5%	84.2%
0.0%	9.6%	48.2%	42.2%
13.7%	41.1%	0.0%	58.9%
52.2%	0.0%	78.4%	0.0%

Retiring	
General Retirement	Microcode Sequencer
92.7	0.1%
19.5%	0.0%
30.4%	0.0%
9.1%	0.0%
0.0%	0.0%

Retiring	
General Retirement	
FP Arithmetic	Other
0.0%	100.0%
25.0%	75.0%
40.1%	59.9%
6.3%	93.8%

Fixing performance issues often increases the portion of uops classified as General Retirement, which is the best case.

The other sub-category, Microcode Sequencer, indicates the uops retired were generated from the microcode sequencer.

While it is the best category, Retiring uops can still be inefficient.

FP Arithmetic

Why Optimize This?
Floating Point Arithmetic can be expensive if done inefficiently.
Associated Metrics
Retiring
└ General Retirement
  └ FP Arithmetic
    └ All Sub-metrics

These metrics represent the breakdown of each type of instruction as a percentage of all retired uops. It doesn't matter how efficiently instructions are being retired if those instructions don't need to be executed in the first place.

Vectorization is a particularly good way to avoid doing unnecessary work. Why perform eight operations when you can do the same calculation with one? If FP x87 and FP Scalar are significant metrics, try to increase the FP Vector percentage by improving vectorization.

Conclusion

The Top-Down Method and its availability in VTune Amplifier represent a new direction for performance tuning using PMUs. The goal of the Top-Down Method is to identify the dominant bottlenecks in an application performance. The goal of VTune Amplifier's Microarchitecture Exploration analysis and visualization features is to give you actionable information for improving your applications. Together, these capabilities can significantly boost not only application performance, but also the productivity of your optimizations.

Useful References

- VTune™ Amplifier Product Page
- VTune™ Amplifier Training Resources
- VTune™ Amplifier User Forums
- VTune™ Amplifier User's Guide
- Intel® 64 and IA-32 Architecture Software Developer's Manuals
- VTune™ Amplifier Tuning Guides for Other Microarchitectures

Legal Disclaimer