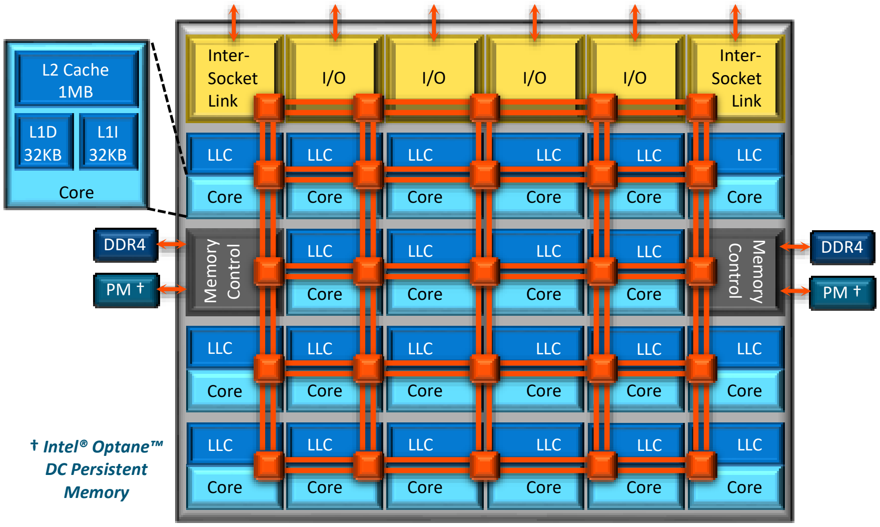
Intel® VTune™ Amplifier Tuning Guide for the Intel® Xeon® Processor Scalable Family, 2nd Gen

How to Use This Guide

This guide is intended for software developers interested in optimizing their application's performance on the Intel® Xeon® processor Scalable Family 2nd Gen, 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.



† Intel® Optane™ DC Persistent Memory

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

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.

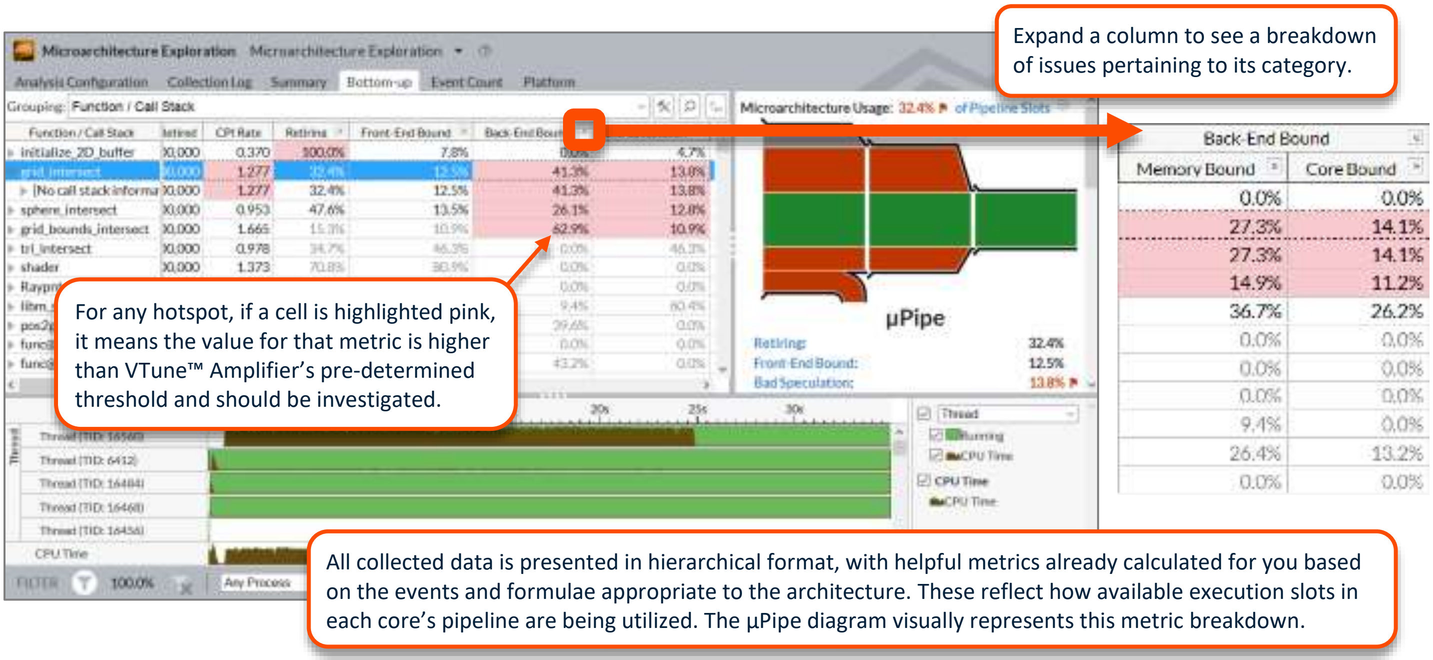
Additionally, some features present in the Intel® Xeon® processor Scalable Family 2nd Gen can have significant effects on performance measurement, and make the process of measuring and interpreting performance data more complex. It may be beneficial to disable Intel® Hyper-Threading Technology and Intel® Turbo Boost 2.0 Technology for the duration of the optimization process, including data collection, and then re-enable them when you are done optimizing. Both of these features can be enabled and disabled through the BIOS on most platforms.

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 (Professional and Cluster Editions only) and Intel® System Studio. It can be run via command line, graphical interface, or integration with Microsoft* Visual Studio*, on Windows* and Linux* operating systems. Data can be viewed on macOS* systems as well. This tool is compatible with multiple languages, including C/C++, Fortran, Java, Assembly, Python, and more.

Intel® VTune™ Amplifier contains several pre-configured analysis types; this guide will focus primarily on the Microarchitecture Exploration analysis (formerly known as General Exploration). No research or familiarity with the hardware events is necessary, as the pre-configured analysis types are already set to collect the appropriate hardware events for your processor. The Microarchitecture Exploration analysis is also pre-programmed with all relevant formulae, and will automatically calculate the appropriate metrics for display in the default view. Some pre-configured profiles will display raw data, but still do not require manual input of the desired hardware events.



Expand a column to see a breakdown of issues pertaining to its category.

Function / Call Stack	...ntired	CPI Rate	Retiring	Front-End Bound	Back-End Bound	
initialize_2D_buffer	)0,000	0.370	100.0%	7.8%	0.0%	4.7%
grid_intersect	)0,000	1.277	32.4%	12.5%	41.3%	13.8%
[No call stack informa	)0,000	1.277	32.4%	12.5%	41.3%	13.8%
sphere_intersect	)0,000	0.953	47.6%	13.5%	26.1%	12.8%
grid_bounds_intersect	)0,000	1.665	15.3%	10.9%	62.9%	10.9%
tri_intersect	)0,000	0.978	34.7%	46.3%	0.0%	46.3%
shader	)0,000	1.373	70.8%	30.9%	0.0%	0.0%
Raypr					0.0%	0.0%
libm_					9.4%	60.4%
pos2p					39.6%	0.0%
func@					0.0%	0.0%
func@					43.2%	0.0%

Microarchitecture Usage: 32.4% of Pipeline Slots

Retiring: 32.4%
Front-End Bound: 12.5%
Bad Speculation: 13.8%

Back-End Bound	
Memory Bound	Core Bound
0.0%	0.0%
27.3%	14.1%
27.3%	14.1%
14.9%	11.2%
36.7%	26.2%
0.0%	0.0%
0.0%	0.0%
0.0%	0.0%
9.4%	0.0%
26.4%	13.2%
0.0%	0.0%

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.

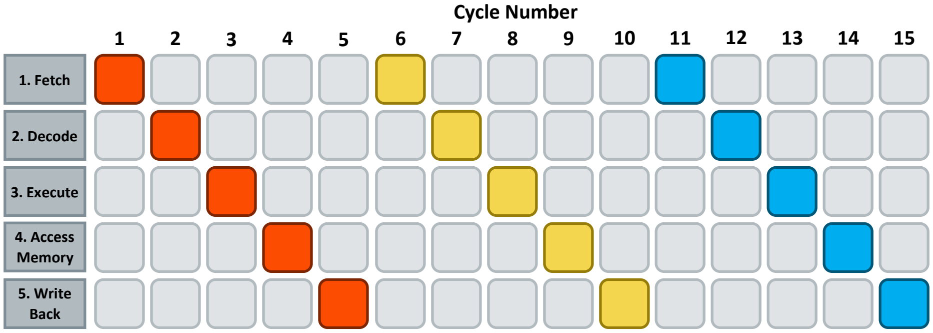
All collected data is presented in hierarchical format, with helpful metrics already calculated for you based on the events and formulae appropriate to the architecture. These reflect how available execution slots in each core's pipeline are being utilized. The μPipe diagram visually represents this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

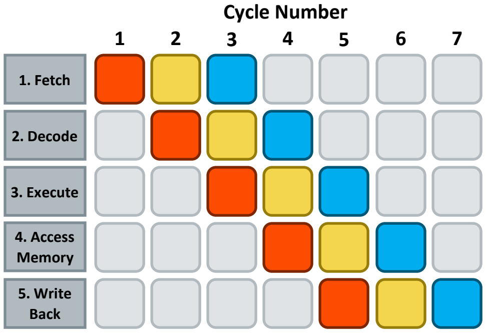
Most screenshots in this guide were taken in Intel® VTune™ Amplifier 2017 Update 3. They may not necessarily have been taken on the microarchitecture this guide is written for. Screenshots taken in different versions of this tool may have minor differences.

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.

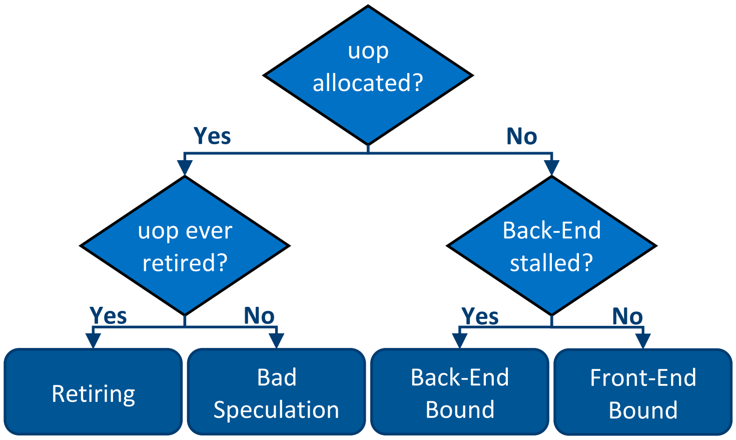




To improve efficiency, modern computers pipeline the uops: because the different steps in processing an instruction are handled by different sections of hardware, they can process multiple instructions at once. For instance, in cycle 3 in this diagram, it's fetching the blue instruction, decoding the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

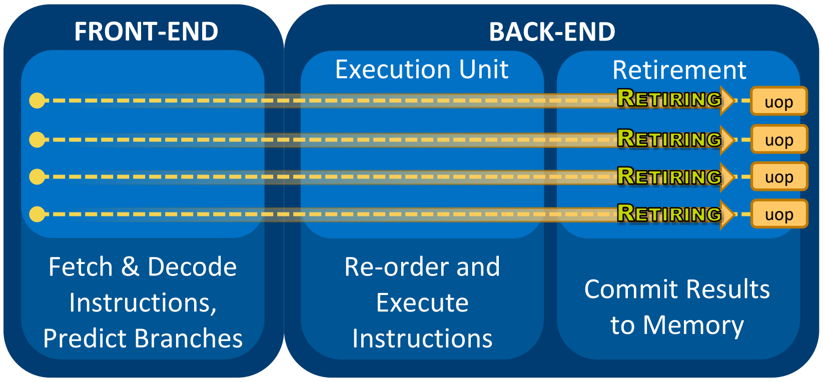


Each pipeline slot category is expected to fall within a particular percentage range for a well-tuned application of a given type. These ranges are detailed in the table below.

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%

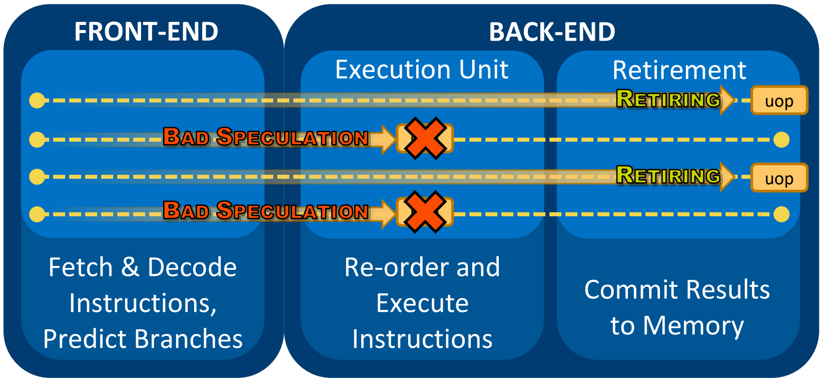
Retiring



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.

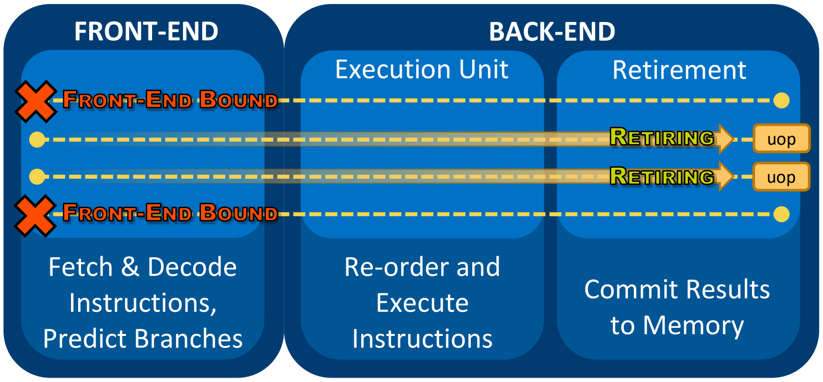
Bad Speculation



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.

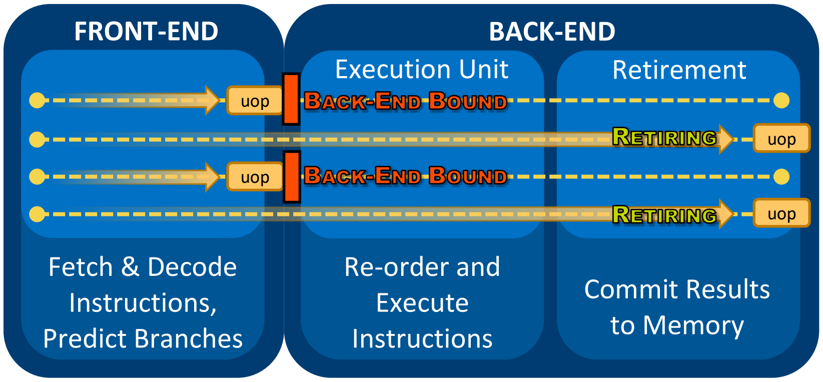
Front-End Bound



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

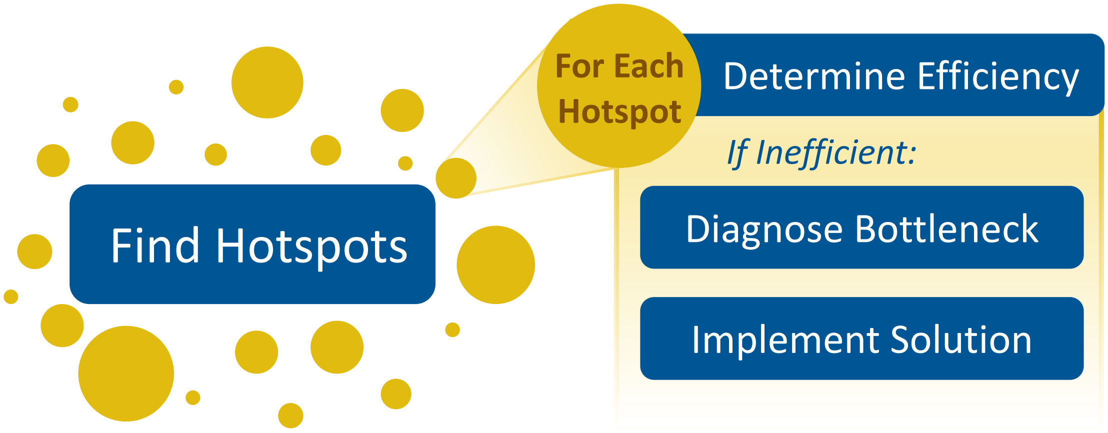


This category refers to cycles on which the Back-End couldn't accept uops in a pipeline slot or slots. This usually occurs because the Back-End is already occupied by uops waiting on data or taking longer to execute. Using the laundry metaphor, the washer is done, but the dryer is still running and can't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

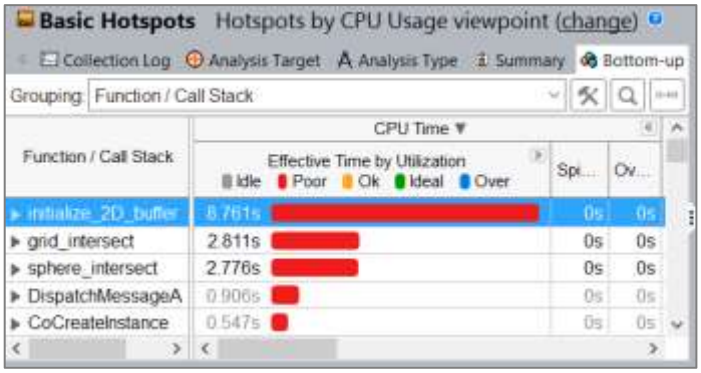
See the section on tuning Back-End Bound for more information.

Software-on-Hardware Tuning

The Software-on-Hardware 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



Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼ Effective Time by Utilization ▮ Idle ▮ Poor ▮ Ok ▮ Ideal ▮ Over	Spi...	Ov...
▸ initialize_2D_buffer	8.761s	0s	0s
▸ grid_intersect	2.811s	0s	0s
▸ sphere_intersect	2.776s	0s	0s
▸ DispatchMessageA	0.906s	0s	0s
▸ CoCreateInstance	0.547s	0s	0s

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 Hotspots 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.

NOTE: Because it counts at the reference frequency for the CPU, the `CPU_CLK_UNHALTED.REF_TSC` counter should not increase or decrease as a result of frequency changes due to Turbo Mode 2.0 or Speedstep Technology. This makes it useful for removing the variance introduced by these technologies when comparing multiple analyses.

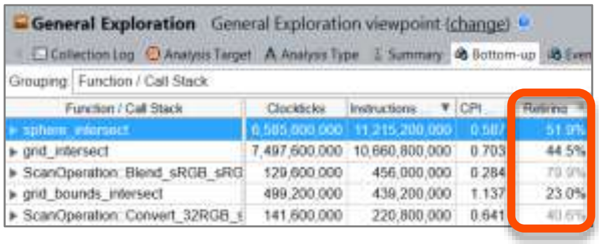
Once you have identified your hotspots, you can proceed through the rest of the process for each one: determine whether it is inefficient, and if so, determine the bottleneck, identify the cause, and optimize the code.

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. To do this, run a Microarchitecture Exploration analysis. 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

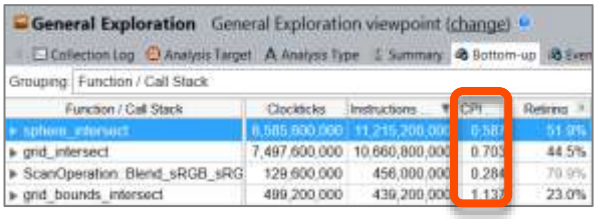
Function / Call Stack	Clockticks	Instructions	CPI	Retiring
sphere_intersect	6,585,600,000	11,215,200,000	0.587	51.9%
grid_intersect	7,497,600,000	10,660,800,000	0.703	44.5%
ScanOperation::Blend_sRGB_sRG	129,600,000	456,000,000	0.284	70.9%
grid_bounds_intersect	499,200,000	439,200,000	1.137	23.0%
ScanOperation::Convert_32RGB_s	141,600,000	220,800,000	0.641	40.6%

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

NOTE: CPI may be affected by Intel® Hyper-Threading. In a serial workload without Intel® Hyper-Threading, the theoretical best CPI on a hardware thread is 0.25, because the core can allocate and retire four instructions per cycle. With Intel® Hyper-Threading enabled, the theoretical best CPI on a hardware thread would be 0.5, because the hardware threads share the allocation and retirement resources.



Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions	CPI	Retiring
sphere_intersect	6,585,600,000	11,215,200,000	0.587	51.9%
grid_intersect	7,497,600,000	10,660,800,000	0.703	44.5%
ScanOperation::Blend_sRGB_sRG	129,600,000	456,000,000	0.284	70.9%
grid_bounds_intersect	499,200,000	439,200,000	1.137	23.0%

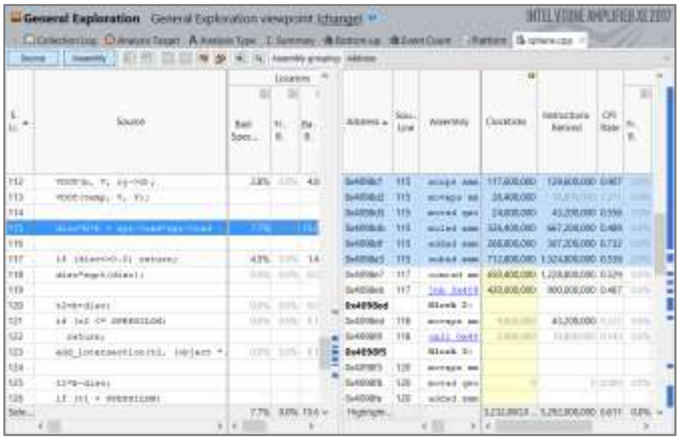
Another measurement of performance is Cycles Per Instruction (CPI), the average time instructions in your workload take to execute. CPI is a general efficiency metric, most useful for comparing sets of data, and is not a robust indicator of inefficiency in and of itself. The Intel® VTune™ Amplifier interface 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.



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, although you should be aware that not all code can be vectorized. Note that some older vector instructions may be present among more recent instructions, and this is not a problem. However, if newer instructions are absent entirely, you may be compiling without the proper flags for your architecture.

- Intel Compiler: `/QxCORE-AVX512` for Windows*, `-xCORE-AVX512` for Linux*
- GCC: `-march=skylake-avx512`

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.
AVX and AVX2	AVX and AVX2 instructions both use ymm registers. AVX2 adds additional functionality to AVX, so AVX2 instructions often coexist with AVX instructions. These instructions are prefixed with a v.
AVX-512	AVX-512 instructions use the zmm registers. They are also prefixed with a v.

NOTE: Intel also provides a powerful analysis tool called Intel® Advisor, which includes a Vectorization Workflow for evaluating and tuning your use of SIMD instructions. You can read more about Intel® Advisor on its official product webpage.

Fused Multiply-Add Instructions

Fused multiply-add (FMA) instructions have the same latency as a floating point multiply, and essentially perform a multiply and an addition as a single operation. This allows for higher peak FLOPs/cycle. Examine your source code for operations of the format `r=±(x*y)±z`. Check the corresponding assembly to see if FMA instructions are being used. FMA instruction names begin with VFM or VFNM.

If you are not using FMAs, you may not be using the proper compiler switches.

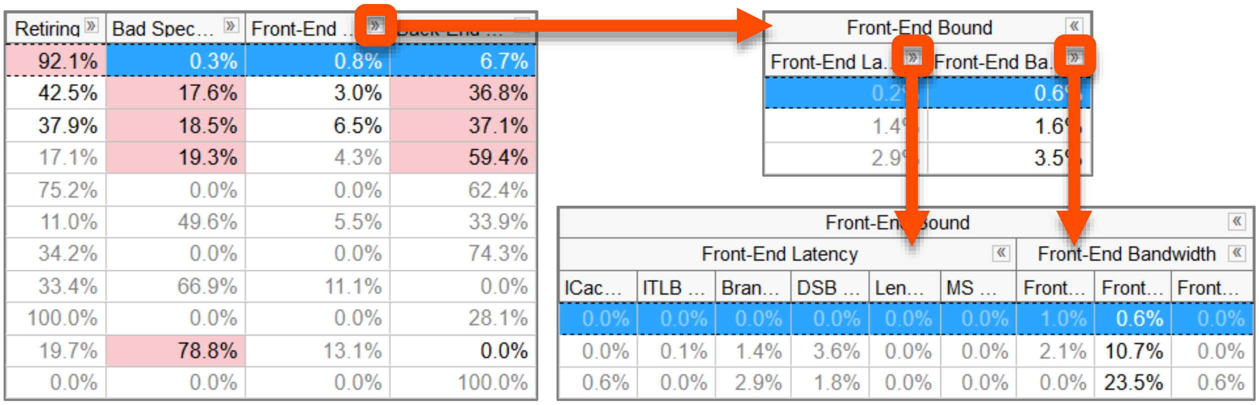
- Intel Compiler:
  - Linux: `-fma` along with either the `-x` or `-march` option set to `CORE-AVX2` or higher.
  - Windows: `/Qfma` along with either the `/Qx` or `/arch` option set to `CORE-AVX2` or higher.
- GCC: `-xfma` or `-march=skylake-avx512`

WARNING! Replacing separate multiply and add instructions with a fused multiply-add instruction may produce slightly different results. This is because a multiply followed by an add has two rounding steps, one after the multiply and one after the add, while a fused multiply-add only rounds once, at the end of the operation.
More information is available in the Intel® 64 and IA-32 Architectures Software Developer's Manual.

Diagnose and Optimize the Bottleneck

Front-End Bound

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



Retiring	Bad Spec...	Front-End ...	Back-End ...
92.1%	0.3%	0.8%	6.7%
42.5%	17.6%	3.0%	36.8%
37.9%	18.5%	6.5%	37.1%
17.1%	19.3%	4.3%	59.4%
75.2%	0.0%	0.0%	62.4%
11.0%	49.6%	5.5%	33.9%
34.2%	0.0%	0.0%	74.3%
33.4%	66.9%	11.1%	0.0%
100.0%	0.0%	0.0%	28.1%
19.7%	78.8%	13.1%	0.0%
0.0%	0.0%	0.0%	100.0%

Front-End Bound	
Front-End La...	Front-End Ba...
0.2	0.6
1.4	1.6
2.9	3.5

Front-End Bound								
Front-End Latency						Front-End Bandwidth		
ICac...	ITLB ...	Bran...	DSB ...	Len...	MS ...	Front...	Front...	Front...
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.0%	0.6%	0.0%
0.0%	0.1%	1.4%	3.6%	0.0%	0.0%	2.1%	10.7%	0.0%
0.6%	0.0%	2.9%	1.8%	0.0%	0.0%	0.0%	23.5%	0.6%

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

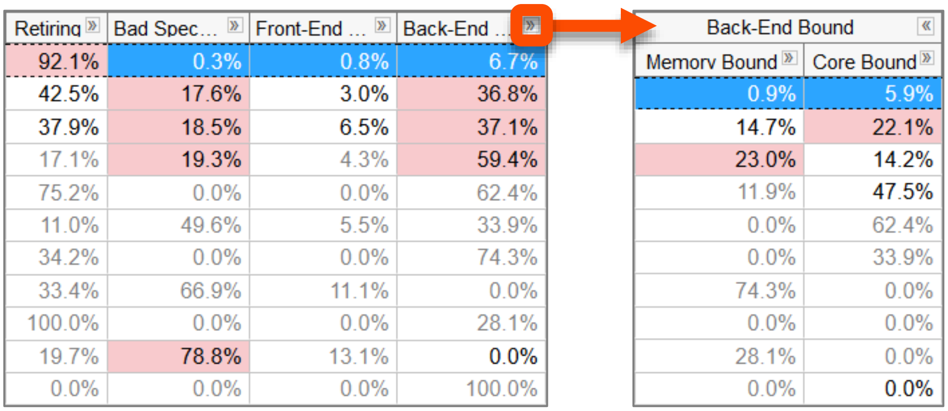
Front-End Latency indicates that you may have a problem with inefficient code layout or generation.

You may want to reduce your code size with switches like `/O1` or `/Os`, use linker ordering techniques (using `/ORDER` on the Microsoft* 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.



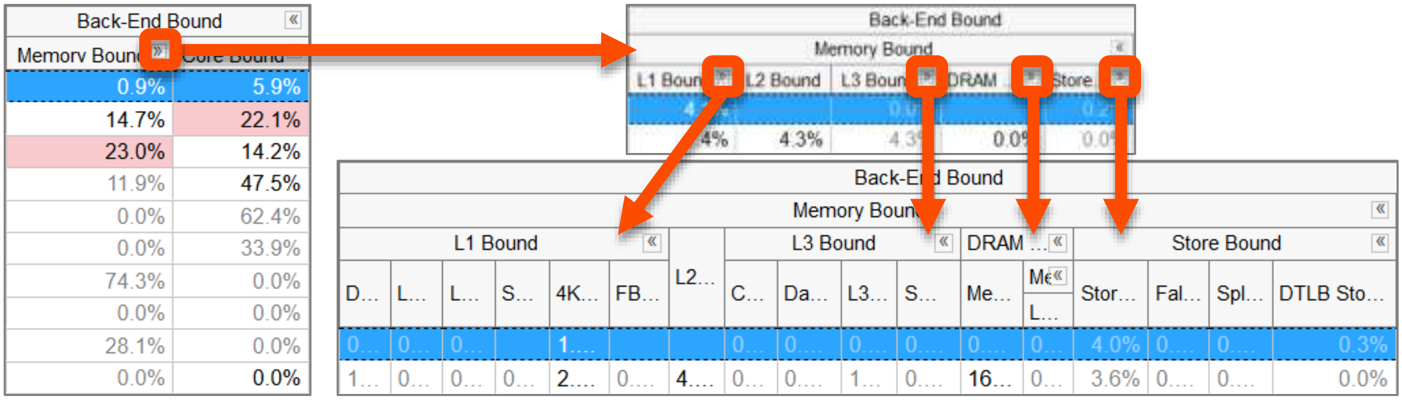
Retiring	Bad Spec…	Front-End …	Back-End …
92.1%	0.3%	0.8%	6.7%
42.5%	17.6%	3.0%	36.8%
37.9%	18.5%	6.5%	37.1%
17.1%	19.3%	4.3%	59.4%
75.2%	0.0%	0.0%	62.4%
11.0%	49.6%	5.5%	33.9%
34.2%	0.0%	0.0%	74.3%
33.4%	66.9%	11.1%	0.0%
100.0%	0.0%	0.0%	28.1%
19.7%	78.8%	13.1%	0.0%
0.0%	0.0%	0.0%	100.0%

Back-End Bound	
Memory Bound	Core Bound
0.9%	5.9%
14.7%	22.1%
23.0%	14.2%
11.9%	47.5%
0.0%	62.4%
0.0%	33.9%
74.3%	0.0%
0.0%	0.0%
28.1%	0.0%
0.0%	0.0%

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
0.9%	5.9%
14.7%	22.1%
23.0%	14.2%
11.9%	47.5%
0.0%	62.4%
0.0%	33.9%
74.3%	0.0%
0.0%	0.0%
28.1%	0.0%
0.0%	0.0%

Back-End Bound				
Memory Bound				
L1 Boun	L2 Bound	L3 Boun	DRAM	Store
4		0.0		0.2
4%	4.3%	4.3	0.0	0.0

Back-End Bound																
Memory Bound																
L1 Bound						L2…	L3 Bound				DRAM …		Store Bound			
D…	L…	L…	S…	4K…	FB…		C…	Da…	L3…	S…	Me…	Me L…	Stor…	Fal…	Spl…	DTLB Sto…
0…	0…	0…		1….			0…	0….	0….	0….	0….	0…	4.0%	0….	0….	0.3%
1…	0…	0…	0…	2….	0….	4….	0…	0….	1…	0….	16…	0…	3.6%	0….	0….	0.0%

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

Most of these issues are due to inefficiencies rather than space concerns. If you believe that your application suffers from insufficient memory, you should assess whether you could benefit from Intel® Optane™ DC Persistent Memory.

Cache Misses

WHY OPTIMIZE THIS?

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

ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  ├ L3 Bound
  │ └ L3 Latency
  └ DRAM Bound

When optimizing applications with cache misses as a bottleneck, focus on the longer-latency accesses from higher-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 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 outlined in section B.5.4.2 of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

SUB-NUMA CLUSTER MODE

Sub-NUMA Cluster Mode (SNC) is a system setting that can be configured in the BIOS, which associates LLC slices with the nearest memory controller. This allows applications to use NUMA primitives to achieve lower LLC/memory latency. Besides optimizing for NUMA, no additional changes to the code are necessary to use SNC.

Previous architecture generations supported Clusters-On-Die (COD). SNC, previously available on Intel® Xeon Phi™ processors (codenamed Knights Landing), is similar to COD but lacks some of its drawbacks.

- Only one UPI caching agent is required, even in 2-SNC mode.
- Latency for memory access in a remote cluster is smaller.
- Last Level Cache capacity is used more efficiently in 2-cluster mode, with no duplication of lines in the LLC.

You may want to test whether your code's performance increases with SNC if your application…

- has a very large dataset.
- is latency-sensitive.
- does not frequently share data across many threads.
- benefited from Cluster-On-Die (COD) on previous generations.
- benefits from SNC on Intel® Xeon Phi™ processors.

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 Threading analysis should also indicate a problem. However, the Threading 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 `HITM`s by viewing the source.

1. Look for the `MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM_PS` event, which will tag to the instruction that generated the `HITM`.
2. 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.

Data Sharing (a.k.a. Read Sharing)

WHY OPTIMIZE THIS?

Sharing clean data among cores at L2 level has a penalty the first time due to cache coherency.

ASSOCIATED METRICS

Back-End Bound
└ Memory Bound
  └ Data Sharing

This metric measures read sharing, or sharing of “clean” data, across L2 caches within one CPU socket. The L3 cache has a set of “core valid” bits that indicate whether each cache line 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.

The Data Sharing metric measures the impact of that additional access time, when the cache line 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 safe to serve. Thus the performance impact only occurs the first time a cache line is requested for reading by a second L2 after it has already been placed in the L3 cache.

The method for addressing this issue is similar to that for Contested Accesses, but uses a different event. If this issue is highlighted for your hotspot, locate the source code line(s) generating `HIT`s by viewing the source.

1. Look for the `MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT_PS` event, which will tag to the instruction that generated the `HIT`.
2. 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.

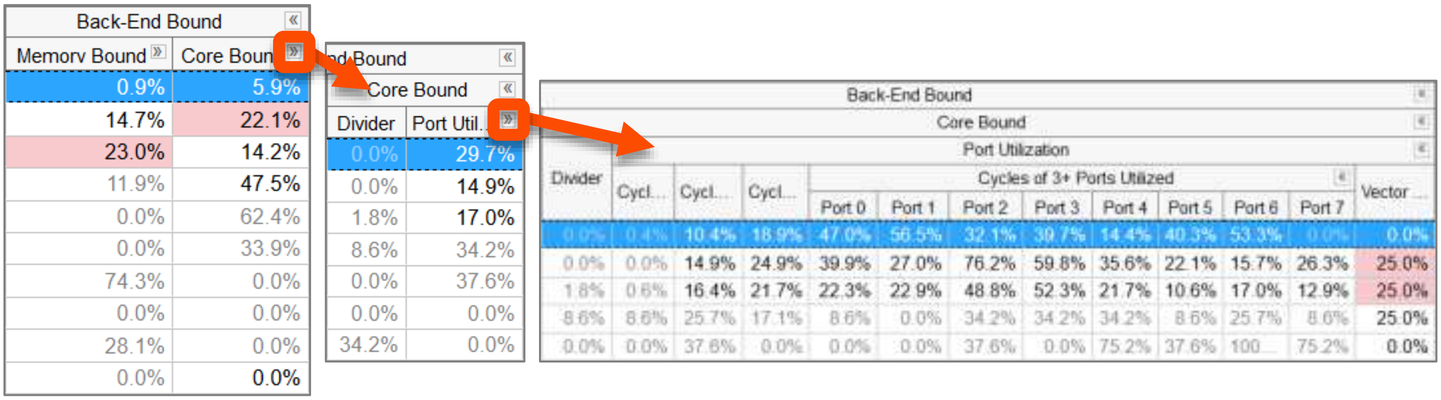
Associated Metrics

Back-End Bound
└ Memory Bound
  └ DTLB Overhead

DTLB (Data Translation Lookaside Buffer) misses are more likely to occur with server applications, or 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 Boun
0.9%	5.9%
14.7%	22.1%
23.0%	14.2%
11.9%	47.5%
0.0%	62.4%
0.0%	33.9%
74.3%	0.0%
0.0%	0.0%
28.1%	0.0%
0.0%	0.0%

nd Bound	
Core Bound	
Divider	Port Util.
0.0%	29.7%
0.0%	14.9%
1.8%	17.0%
8.6%	34.2%
0.0%	37.6%
0.0%	0.0%
34.2%	0.0%

Back-End Bound												
Core Bound												
	Port Utilization											
Divider	Cycl...	Cycl...	Cycl...	Cycles of 3+ Ports Utilized								Vector ...
				Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	
0.0%	0.4%	10.4%	18.9%	47.0%	56.5%	32.1%	30.7%	14.4%	40.3%	53.3%	0.0%	0.0%
0.0%	0.0%	14.9%	24.9%	39.9%	27.0%	76.2%	59.8%	35.6%	22.1%	15.7%	26.3%	25.0%
1.8%	0.6%	16.4%	21.7%	22.3%	22.9%	48.8%	52.3%	21.7%	10.6%	17.0%	12.9%	25.0%
8.6%	8.6%	25.7%	17.1%	8.6%	0.0%	34.2%	34.2%	34.2%	8.6%	25.7%	8.6%	25.0%
0.0%	0.0%	37.6%	0.0%	0.0%	0.0%	37.6%	0.0%	75.2%	37.6%	100...	75.2%	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

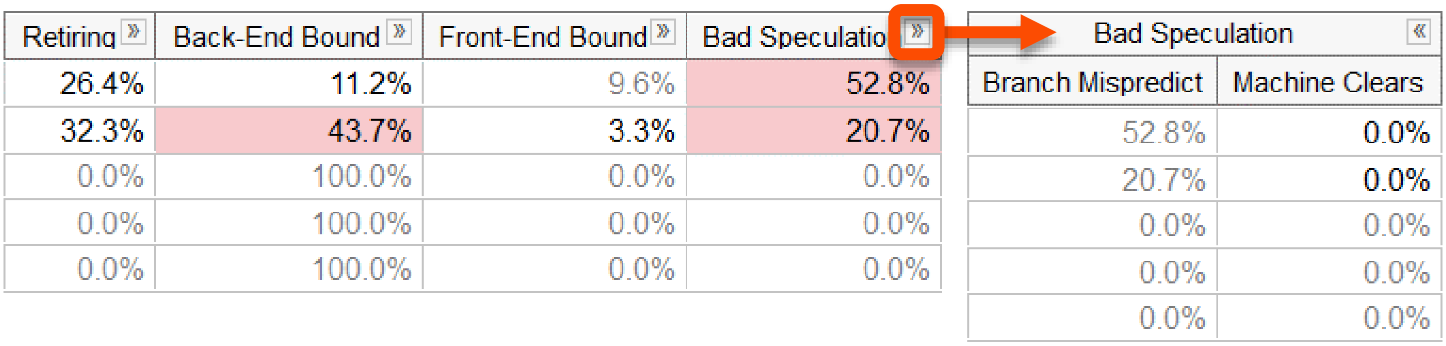
Divide instructions are more expensive than other arithmetic instructions, so should be avoided where possible.

Locate the code line(s) generating divide instructions by viewing the source and looking for the `ARITH.DIVIDER_ACTIVE` event.

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	Back-End Bound	Front-End Bound	Bad Speculatio
26.4%	11.2%	9.6%	52.8%
32.3%	43.7%	3.3%	20.7%
0.0%	100.0%	0.0%	0.0%
0.0%	100.0%	0.0%	0.0%
0.0%	100.0%	0.0%	0.0%

Bad Speculation	
Branch Mispredict	Machine Clears
52.8%	0.0%
20.7%	0.0%
0.0%	0.0%
0.0%	0.0%
0.0%	0.0%

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

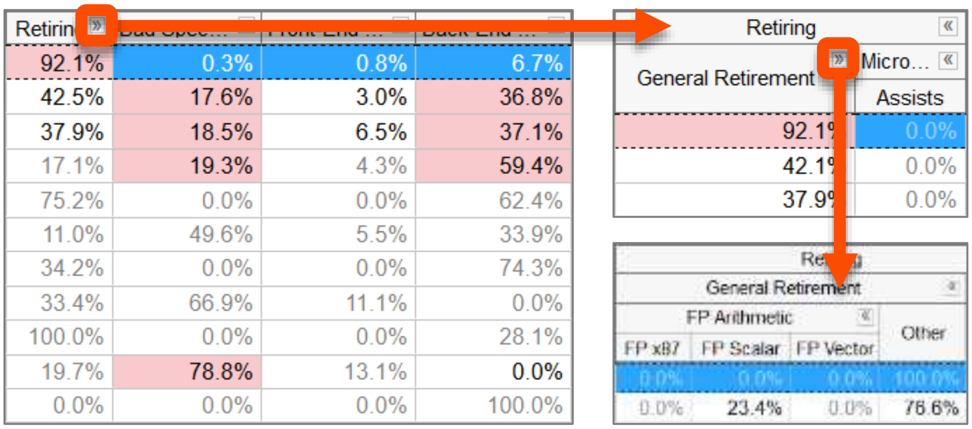
Machine clears are much rarer than branch mispredicts. These are generally caused by contention on a lock, failed memory disambiguation from 4K aliasing, or self-modifying code.

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.



Retirin	Bad Spec...	Front-End ...	Back-End ...
92.1%	0.3%	0.8%	6.7%
42.5%	17.6%	3.0%	36.8%
37.9%	18.5%	6.5%	37.1%
17.1%	19.3%	4.3%	59.4%
75.2%	0.0%	0.0%	62.4%
11.0%	49.6%	5.5%	33.9%
34.2%	0.0%	0.0%	74.3%
33.4%	66.9%	11.1%	0.0%
100.0%	0.0%	0.0%	28.1%
19.7%	78.8%	13.1%	0.0%
0.0%	0.0%	0.0%	100.0%

Retiring	
General Retirement	Micro... Assists
92.1%	0.0%
42.1%	0.0%
37.9%	0.0%

Retiring			
General Retirement			
FP Arithmetic			Other
FP x87	FP Scalar	FP Vector	
0.0%	0.0%	0.0%	100.0%
0.0%	23.4%	0.0%	76.6%

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.

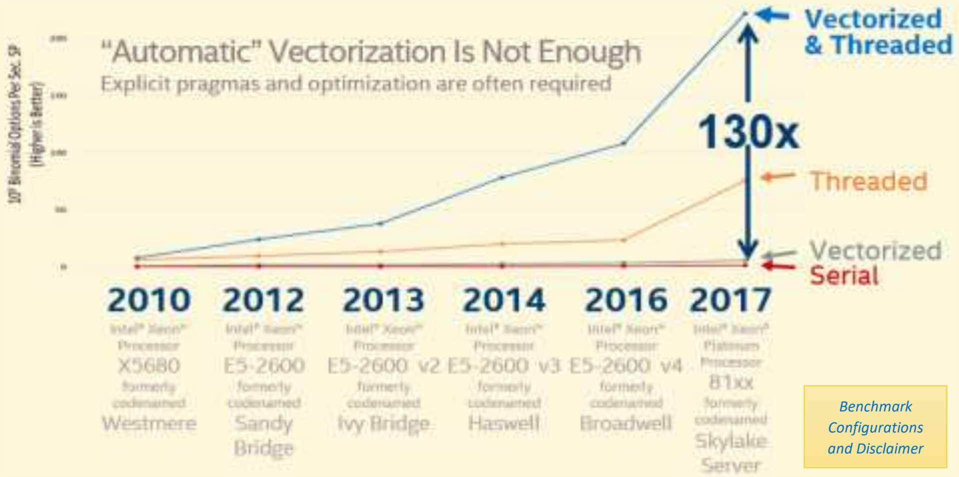
Vectorization is a particularly good way to avoid doing unnecessary work. Why perform eight operations when you can do the same calculation with one? If FP x87 and FP Scalar are significant metrics, try to increase the FP Vector percentage by improving vectorization.

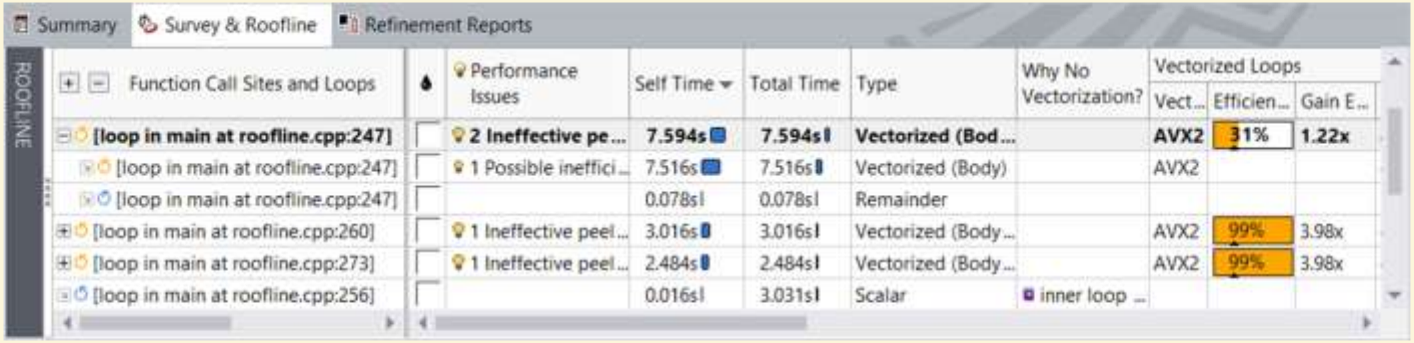
Intel® Advisor

For deeper vectorization optimization, consider using Intel® Advisor, a specialized tool for thread prototyping and vectorization tuning.

Intel® Advisor is available on its own or as part of the Intel® Parallel Studio XE (Profession and Cluster Editions only) and Intel® System Studio suites.

You can find out more or download a free copy at the Intel® Advisor product site.





Function Call Sites and Loops		Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops		
							Vect...	Efficien...	Gain E...
[loop in main at roofline.cpp:247]	□	2 Ineffective pe...	7.594s	7.594s	Vectorized (Bod...		AVX2	31%	1.22x
[loop in main at roofline.cpp:247]	□	1 Possible ineffici...	7.516s	7.516s	Vectorized (Body)		AVX2		
[loop in main at roofline.cpp:247]	□		0.078s	0.078s	Remainder				
[loop in main at roofline.cpp:260]	□	1 Ineffective peel...	3.016s	3.016s	Vectorized (Body...		AVX2	99%	3.98x
[loop in main at roofline.cpp:273]	□	1 Ineffective peel...	2.484s	2.484s	Vectorized (Body...		AVX2	99%	3.98x
[loop in main at roofline.cpp:256]	□		0.016s	3.031s	Scalar	inner loop ...			

Microcode Assists

WHY OPTIMIZE THIS?

Assists from the microcode sequencer can have long latency penalties.

ASSOCIATED METRICS

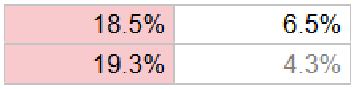
Retiring
└ Microcode Assists

Many instructions can cause assists without causing a performance problem. If this metric is highlighted for your hotspot, re-sample using the assist events to determine the cause.

If `FP_ASSIST.ANY/INST_RETIRED.ANY` is significant, check for denormals. To fix this problem, enable FTZ and/or DAZ if using SSE/AVX instructions, or scale your results up or down depending on the problem.

Additional Topics

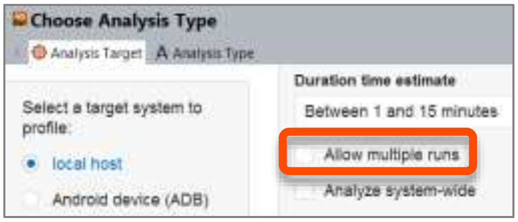
Metric Reliability



18.5%	6.5%
19.3%	4.3%

The 4.3% in the lower right is grayed out as unreliable.

The Microarchitecture Exploration analysis type (and all hardware event based analysis types) multiplexes hardware events during collection, which can result in imprecise results if too few samples are collected. More information about how hardware event based data collection works can be found in the VTune™ Amplifier Help document.



VTune™ Amplifier's GUI will gray out metrics if their reliability is too low based on the number of samples collected. This is calculated per-metric rather than per-row (as in some older versions of the product), and is calculated independently for each sub-metric.

If a metric is grayed out for your area of interest, consider increasing the runtime of the analysis, or allowing multiple runs via a checkbox in the advanced section when configuring the analysis.

Memory Bandwidth

Memory bandwidth bottlenecks can increase the latency incurred by cache misses. In addition to the Microarchitecture Exploration analysis, VTune™ Amplifier has several specialized analysis types designed to investigate specific types of problems. If you suspect you have a memory bandwidth issue, try running the Memory Access analysis.

Begin by computing the maximum theoretical memory bandwidth per socket for your platform in GB/s, based on the megatransfers per second and number of channels, using the formula to the right. For example, a processor with DDR 1600 and 4 channels has a maximum theoretical bandwidth of 51.2 GB/s.

$$\text{GB/s} = \frac{(MT/s) * 8\ \text{Bytes/Clock} * (channels)}{1000}$$

If the total bandwidth per socket is greater than 75% of the maximum theoretical bandwidth, your application may be experiencing higher latencies. If appropriate, make system tuning adjustments, such as upgrading/balancing DIMMs or disabling hardware prefetchers. Also try to reduce bandwidth usage; remove ineffective software prefetches, make algorithmic changes to reduce data storage/sharing, reduce data updates, and balance memory access across sockets.

Memory Consumption and Intel® Optane™ DC Persistent Memory

The Intel® Xeon® processor Scalable Family 2nd Gen is the first to support Intel® Optane™ DC Persistent Memory, a new tier of storage between DRAM and the disk. You can determine whether your application could benefit from this technology even if you don't yet have the hardware for it using the Memory Consumption analysis type in VTune™ Amplifier. This analysis reveals your application's peak memory footprint. If this value is 90% or more of your available DRAM, you may benefit from using persistent memory.

NOTE: Memory Consumption analysis is not currently available on Windows* operating systems. However, you can get a sense of how well your data fits into DRAM from the DRAM Cache Hits and DRAM Cache Misses metrics provided by a Memory Access analysis. These metrics are only available on machines with Memory Mode enabled.

The next step is to identify your application's working set using the Memory Access analysis type with "analyze dynamic memory objects" enabled. Identify the objects with the most accesses and sum up their sizes to find your working set size. If this size is small enough to fit into DRAM, you may wish to try enabling Intel® Optane™ DC Persistent Memory in Memory Mode. In this mode, the additional memory is not actually persistent, but rather acts as an extension of the cache hierarchy, and requires no special programming. Ideally, the most frequently accessed objects will stay in DRAM, while the rest of the application's memory footprint resides in persistent memory instead of on the disk.

If your working set is too big for DRAM, or you don't see the desired results in Memory Mode, you may wish to try App Direct Mode. There are two versions of App Direct Mode available: Volatile (not persistent) and Non-Volatile (persistent). In both cases, this mode requires more than simply enabling persistent memory, as you have to use an API to manually control allocation of memory.

For effective use of App Direct Mode, the objects with the most LLC misses and store-heavy objects should be allocated in DRAM, while the rest of the objects should be allocated in persistent memory. The goal here is to take advantage of two properties: first, that DRAM is faster than persistent memory, so more frequently accessed objects should reside in DRAM; and second, that loading from persistent memory is much faster than storing to it, so store-heavy objects are not well suited to persistent memory. Balancing the importance of these two factors in deciding where to allocate a given object is left to your discretion.

INTEL® INSPECTOR – PERSISTENCE INSPECTOR

Persistent memory comes with its own challenges; for instance, data does not actually persist until it has been properly flushed out of cache and into the persistent memory system. Because of how modern processor out-of-order execution and caching work, the order that data becomes persistent in may not be the same as the order it is stored in, and programming errors can lead to persistent memory not behaving as expected.

Intel also provides Intel® Inspector – Persistence Inspector, a tool that can be used to check for these errors and ensure that your code works as intended by detecting:

- Cache flush misses
- Redundant cache flushes and memory fences
- Out-of-order persistent memory stores
- Incorrect Persistent Memory Development Kit (PMDK) undo logging

Discovering persistent memory errors early in your application's development cycle is critical to ensure that they can be corrected before they become a problem. Intel® Inspector is available on its own or as part of Intel® Parallel Studio XE (Professional and Cluster editions only) and Intel® System Studio. You can find out more on its official web page.

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
- Compiler Options Guide
- Intel® Advisor Product Page
- Intel® Inspector Product Page

Legal Disclaimer & 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

CONFIGURATIONS FOR 2010-2017 BENCHMARKS

Performance results are based on testing as of Sept. 12 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

Platform	Intel® Xeon™ X5680 Processor	Intel® Xeon™ E5 2690 Processor	Intel® Xeon™ E5 2697v2 Processor	Intel® Xeon™ E5 2600v3 Processor	Intel® Xeon™ E5 2600v4 Processor	Intel® Xeon™ E5 2600v4 Processor	Intel® Xeon™ Platinum 81xx Processor
Code name	WSM	SNB	IVB	HSW	BDW	BDW	SKX
Unscaled Core Frequency	3.33 GHZ	2.9 GHZ	2.7 GHZ	2.2 GHz	2.3 GHz	2.2 GHz	2.5 GHz
Cores/Socket	6	8	12	18	18	22	28
Num Sockets	2	2	2	2	2	2	2
L1 Data Cache	32K	32K	32K	32K	32K	32K	32K
L2 Cache	256K	256K	256K	256K	256K	256K	1024K
L3 Cache	12MB	20MB	30MB	46MB	46MB	56MB	40MB
Memory	48MB	64GB	64GB	128GB	256GB	128GB	192GB
Memory Frequency	1333 MHz	1600 MHz	1867 MHz	2133 MHz	2400 MHz	2133 MHz	2666 MHz
Memory Access	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
H/W Prefetchers Enabled	Yes	Yes	Yes	Yes	Yes	Yes	Yes
HT Enabled	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Turbo Enabled	Yes	Yes	Yes	Yes	Yes	Yes	Yes
C States	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
O/S Name	Fedora 20	Fedora 20	RHEL 7.1	Fedora 20	RHEL 7.0	CentOS 7.2	CentOS 7.3
Operating System	3.11.10-301.fc20	3.11.10-301.fc20	3.10.0-229.el7.x86_64	3.15.10-200.fc20.x86_64	3.10.0-123.el7.x86_64	3.10.0-327.el7.x86_64	3.10.0-514.10.2.el7.x86_64
Compiler Version	icc version 17.0.2	icc version 17.0.2	icc version 17.0.2	icc version 17.0.2	icc version 17.0.2	icc version 17.0.2	icc version 17.0.2

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, Intel Core, Intel Optane, Intel Xeon Phi, VTune, and Xeon 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.