

Intel® VTune™ Profiler User Guide

Contents

Chapter 1: Intel® VTune™ Profiler User Guide

Introduction	11
Tuning Methodology.....	14
Tutorials and Samples.....	15
Notational Conventions	16
Get Help	17
Product Website and Support.....	21
Related Information.....	21
Install Intel® VTune™ Profiler.....	22
Sampling Drivers	23
Set Up System for GPU Analysis	24
Rebuild and Install the Kernel for GPU Analysis	26
Rebuild and Install Module i915 for GPU Analysis on CentOS*	27
Rebuild and Install Module i915 for GPU Analysis on Ubuntu*	29
Verify Intel® VTune™ Profiler Installation.....	32
Install VTune Profiler Server	32
Set Up Transport Security	35
Configure User Authentication/Authorization.....	36
Security Best Practices.....	38
Open Intel® VTune™ Profiler	39
Get Started with Intel® VTune™ Profiler	40
Intel® VTune™ Profiler Graphical User Interface	42
Web Server Interface.....	44
Microsoft Visual Studio* Integration.....	52
Eclipse* and Intel System Studio IDE Integration	54
Containerization Support.....	55
Run VTune Profiler in a Container	56
Profile Container Targets from the Host	58
Set Up Project	60
WHERE: Analysis System	62
Analysis System Options	63
WHAT: Analysis Target	64
Analysis Target Options.....	66
HOW: Analysis Types	71
Search Directories	72
Search Order.....	74
Set Up Analysis Target	75
Prepare Application for Analysis	78
Windows* Targets	80
Install the Sampling Drivers for Windows* Targets	82
Debug Information for Windows* Application Binaries	83
Compiler Switches for Performance Analysis on Windows* Targets .	84
Debug Information for Windows* System Libraries	86
Add Administrative Privileges.....	88
Linux* Targets	88
Build and Install the Sampling Drivers for Linux* Targets	89

Debug Information for Linux* Application Binaries.....	93
Compiler Switches for Performance Analysis on Linux* Targets.....	95
Enable Linux* Kernel Analysis.....	99
Resolution of Symbol Names for Linux-Loadable Kernel Modules..	101
Analyze Statically Linked Binaries on Linux* Targets	101
Set Up Remote Linux* Target	102
Embedded Linux* Targets.....	113
Configure Yocto Project* and VTune Profiler with the Integration Layer	118
Configure Yocto Project*/Wind River* Linux* and Intel® VTune™ Profiler with the Intel System Studio Integration Layer.....	120
Configure Yocto Project* and Intel® VTune™ Profiler with the Linux* Target Package	122
FreeBSD* Targets	123
Set Up FreeBSD* System.....	126
QNX* Targets	127
Managed Code Targets	128
.NET* Targets.....	129
Windows Store Application Targets	131
Go* Application Targets	132
Android* Targets.....	133
Build and Install Sampling Drivers for Android* Targets.....	135
Set Up Android* System	136
Enable Java* Analysis on Android* System.....	137
Prepare an Android* Application for Analysis	140
Analyze Unplugged Devices.....	141
Search Directories for Android* Targets	142
Intel® Xeon Phi™ Processor Targets	142
Targets in Virtualized Environments	145
Profile Targets on a VMware* Guest System	146
Profile Targets on a Parallels* Guest System	147
Profile Targets on a KVM* Guest System	148
Profile Targets on a Xen* Virtualization Platform.....	157
Profile Targets in the Hyper-V* Environment.....	158
Targets in a Cloud Environment.....	159
Arbitrary Targets	159
Embedded System Targets	161
Analyze Performance	162
User-Mode Sampling and Tracing Collection	163
Hardware Event-based Sampling Collection.....	164
Allow Multiple Runs or Multiplex Events.....	166
Hardware Event-based Sampling Collection with Stacks	167
Performance Snapshot	171
Algorithm Group	174
Hotspots Analysis for CPU Usage Issues	174
Anomaly Detection Analysis (preview)	181
Memory Consumption Analysis	189
Microarchitecture Analysis Group.....	191
Microarchitecture Exploration Analysis for Hardware Issues	191
Memory Access Analysis for Cache Misses and High Bandwidth Issues	200
Parallelism Analysis Group.....	213
Threading Analysis.....	213
HPC Performance Characterization Analysis	222
Input and Output Analysis	237

Analyze Platform Performance	242
Analyze DPDK Applications.....	252
Analyze SPDK Applications	255
Analyze Linux Kernel I/O.....	259
Accelerators Analysis Group.....	262
GPU Offload Analysis	263
GPU Compute/Media Hotspots Analysis (Preview)	273
CPU/FPGA Interaction Analysis.....	295
NPU Exploration Analysis (Preview)	301
Platform Analysis Group	306
System Overview Analysis.....	306
Platform Analysis	321
Hybrid CPU Analysis	327
Source Code Analysis.....	330
Custom Analysis	337
Custom Analysis Options.....	338
Energy Analysis	353
Run Energy Analysis.....	354
View Energy Analysis Data with Intel® VTune™ Profiler	356
Interpret Energy Analysis Data with Intel® VTune™ Profiler	357
Code Profiling Scenarios.....	360
Java* Code Analysis	361
Python* Code Analysis.....	365
Intel® Threading Building Blocks Code Analysis	368
MPI Code Analysis.....	369
OpenSHMEM* Code Analysis with Fabric Profiler.....	375
GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics	381
Frame Data Analysis.....	396
Task Analysis	398
Control Data Collection	402
Finalization	403
Pause Data Collection	405
Limit Data Collection	406
Generate Command Line Configuration from GUI	407
Minimize Collection Overhead	409
Import External Data.....	410
Manage Data Views	423
Switch Viewpoints	424
Control Window Synchronization	426
View Stacks	427
Manage Grid Views.....	433
Manage Timeline View	435
Change Threshold Values	438
Choose Data Format.....	438
Group and Filter Data	440
View Data on Inline Functions.....	443
Analyze Loops	447
Stitch Stacks for Intel® oneAPI Threading Building Blocks or OpenMP* Analysis.....	448
Search for Data	450
Manage Result Files.....	451
VTune Profiler Filenames and Locations	452
Import Results and Traces into VTune Profiler GUI	454
Compare Results.....	456

Intel® VTune™ Profiler Command Line Interface	464
vtune Command Syntax.....	465
vtune Actions	466
Run Command Line Analysis	467
performance-snapshot Command Line Analysis.....	470
hotspots Command Line Analysis.....	471
anomaly-detection Command Line Analysis	472
threading Command Line Analysis	474
memory-consumption Command Line Analysis	474
hpc-performance Command Line Analysis.....	475
uarch-exploration Command Line Analysis	476
memory-access Command Line Analysis.....	477
tsx-exploration Command Line Analysis.....	478
tsx-hotspots Command Line Analysis	479
sgx-hotspots Command Line Analysis	479
gpu-hotspots Command Line Analysis	480
gpu-offload Command Line Analysis.....	483
npu.....	484
graphics-rendering Command Line Analysis	484
fpga-interaction Command Line Analysis	485
io Command Line Analysis.....	486
system-overview Command Line Analysis.....	487
runsa/runss Custom Command Line Analysis	488
Configure Analysis Options from Command Line.....	490
Work with Results from Command Line	499
View Command Line Results in the GUI.....	499
Import Results from Command Line	500
Re-finalize Results from Command Line.....	502
Generate Command Line Reports	503
Summary Report	505
Hotspots Report.....	510
Hardware Events Report	511
Callstacks Report	512
Timeline Report	514
Top-down Report	519
GPU Compute/Media Hotspots Report	520
gprof-cc Report.....	521
Difference Report.....	522
View Source Objects from Command Line	524
Save and Format Command Line Reports	526
Filter and Group Command Line Reports	527
Command Line Usage Scenarios	530
Use VTune Profiler Server in Containers	530
Android* Target Analysis from the Command Line	531
OpenMP* Analysis from the Command Line	535
Java* Code Analysis from the Command Line	539
Command Line Interface Reference	549
Option Descriptions and General Rules.....	549
allow-multiple-runs	550
analyze-kvm-guest	551
analyze-system	552
app-working-dir.....	552
archive.....	553
call-stack-mode	554

collect	555
collect-with	559
column	560
command	561
cpu-mask	562
csv-delimiter	562
cumulative-threshold-percent	563
custom-collector	565
data-limit.....	566
discard-raw-data	566
duration	567
filter.....	568
finalization-mode	570
finalize	571
format.....	572
group-by	573
help	575
import.....	576
inline-mode.....	577
knob	578
kvm-guest-kallsyms	588
kvm-guest-modules	589
limit	590
loop-mode	591
mrte-mode	592
no-follow-child.....	593
no-summary	594
no-unplugged-mode	595
quiet	595
report	596
report-knob.....	598
report-output	600
report-width	601
result-dir	601
resume-after	603
return-app-exitcode	603
ring-buffer	604
search-dir	605
show-as.....	606
sort-asc	607
sort-desc	608
source-object	609
source-search-dir.....	610
stack-size	611
start-paused	612
strategy.....	612
target-install-dir	613
target-system.....	614
target-tmp-dir	616
target-duration-type.....	617
target-pid	618
target-process	619
time-filter	620
trace-mpi.....	620
user-data-dir.....	621

verbose	622
version	622
Report Problems from Command Line	623
API Support.....	624
Instrumentation and Tracing Technology APIs.....	624
Basic Usage and Configuration	624
Instrumentation and Tracing Technology API Reference	633
JIT Profiling API	657
Using JIT Profiling API	660
JIT Profiling API Reference	661
System APIs Supported by Intel® VTune™ Profiler	667
Troubleshooting.....	676
Best Practices: Resolve Intel® VTune™ Profiler BSODs, Crashes, and	
Hangs in Windows* OS	677
Error Message: Application Sets Its Own Handler for Signal	680
Error Message: Cannot Enable Event-Based Sampling Collection.....	680
Error Message: Cannot Collect GPU Hardware Metrics	682
Error Message: Cannot Load Data File.....	683
Error Message: Cannot Locate Debugging Information	684
Error Message: Cannot Open Data.....	684
Error Message: Client Is Not Authorized to Connect to Server.....	685
Error Message: Root Privileges Required for Processor Graphics Events	685
Error Message: No Pre-built Driver Exists for This System.....	685
Error Message: Not All OpenCL™ API Profiling Callbacks Are Received ...	686
Error Message: Problem Accessing the Sampling Driver.....	687
Error Message: Required Key Not Available	687
Error Message: Scope of ptrace System Call Is Limited	688
Error Message: Stack Size Is Too Small	688
Error Message: Symbol File Is Not Found.....	689
Problem: Analysis of the .NET* Application Fails	689
Problem: Cannot Access VTune Profiler Documentation.....	690
Problem: CPU time for Hotspots or Threading Analysis is Too Low	690
Problem: 'Events= Sample After Value (SAV) * Samples' Is Not True If	
Multiple Runs Are Disabled	691
Problem: Guessed Stack Frames	692
Problem: GUI Hangs or Crashes	692
Problem: Inaccurate Sum in the Grid	693
Problem: Information Collected via ITT API Is Not Available When	
Attaching to a Process	693
Problem: No GPU Utilization Data Is Collected	693
Problem: Same Functions Are Compared As Different Instances	694
Problem: Skipped Stack Frames	694
Problem: Stack in the Top-Down Tree Window Is Incorrect.....	695
Problem: Stacks in Call Stack and Bottom-Up Panes Are Different.....	695
Problem: System Functions Appear in the User Functions Only Mode....	696
Problem: VTune Profiler is Slow to Respond When Collecting or	
Displaying Data	696
Problem: VTune Profiler is Slow on X-Servers with SSH Connection	696
Problem: Unexpected Paused Time	697
Problem: {Unknown Timer} in the Platform Power Analysis Viewpoint ..	698
Problem: Unknown Critical Error Due to Disabled Loopback Interface ...	698
Problem: Unknown Frames.....	699
Problem: Unreadable Text on macOS*	699
Problem: Unsupported Microsoft* Windows* OS.....	700
Warnings about Accurate CPU Time Collection	700

Reference.....	701
User Interface	701
Context Menu: Grid.....	703
Context Menus: Call Stack Pane.....	704
Context Menus: Project Navigator	704
Context Menus: Source/Assembly Window	706
Dialog Box: Binary/Symbol Search	707
Dialog Box: Source Search.....	708
Hot Keys.....	709
Menu: Customize Grouping	710
Menu: Intel VTune Profiler.....	710
Pane: Call Stack	712
Pane: Options - General.....	714
Pane: Options - Result Location	715
Pane: Options - Source/Assembly	716
Project Navigator	717
Pane: Timeline	718
Toolbar: Configure Analysis	721
Toolbar: Filter.....	723
Toolbar: Source/Assembly	725
Toolbar: Intel VTune Profiler	726
Window: Bandwidth - Platform Power Analysis	728
Window: Bottom-up	730
Window: Caller/Callee	733
Window: Cannot Find <file type> File	733
Window: Collection Log.....	734
Window: Compare Results.....	735
Window: Configure Analysis	735
Window: Core Wake-ups - Platform Power Analysis.....	736
Window: Correlate Metrics - Platform Power Analysis	739
Window: CPU C/P States - Platform Power Analysis	742
Window: Debug	745
Window: Event Count - Hardware Events	745
Window: Flame Graph	746
Window: Graphics - GPU Compute/Media Hotspots	749
Window: Graphics C/P States - Platform Power Analysis	751
Window: NC Device States - Platform Power Analysis.....	753
Window: Platform	756
Window: Platform Power Analysis.....	761
Window: Sample Count - Hardware Events.....	763
Window: SC Device States - Platform Power Analysis.....	765
Window: Summary	767
Window: System Sleep States - Platform Power Analysis	805
Window: Temperature/Thermal Sample - Platform Power Analysis	807
Window: Timer Resolution - Platform Power Analysis	809
Window: Top-down Tree.....	811
Window: Uncore Event Count - Hardware Events.....	812
Window: Wakelocks - Platform Power Analysis	813
CPU Metrics Reference	816
GPU Metrics Reference	864
ALU0 Active	865
ALU0 Instructions	865
ALU1 Active	865
ALU1 Instructions	866
ALU2 Active	866

ALU2 Instructions	866
ALU0 and ALU1 Active	866
ALU0 and ALU2 Active	866
Average Time	866
Computing Threads Started	867
Computing Threads Started, Threads/sec	867
CPU Time	867
EU 2 FPU Pipelines Active	867
EU Array Active	867
EU Array Idle	868
EU Array Stalled/Idle	868
EU Array Stalled	868
EU IPC Rate	868
EU Send pipeline active	869
EU Threads Occupancy	869
Host to GPU Memory Read Bandwidth	869
Host-to-GPU Memory Write Bandwidth	869
Global	869
GPU EU Array Usage	869
GPU L3 Bound	870
GPU L3 Miss Ratio	870
GPU L3 Misses	870
GPU L3 Misses, Misses/sec	870
GPU Memory Read Bandwidth, GB/sec	870
GPU Memory Texture Read Bandwidth, GB/sec	871
GPU Memory Write Bandwidth, GB/sec	871
GPU Texel Quads Count, Count/sec	871
GPU Utilization	871
Instance Count	872
L3 Read Bandwidth	872
L3 Write Bandwidth	873
L3 Sampler Bandwidth, GB/sec	873
L3 Shader Bandwidth, GB/sec	873
LLC Miss Rate due GPU Lookups	873
LLC Miss Ratio due GPU Lookups	873
Local	874
Maximum GPU Utilization	874
Occupancy	874
PS EU Active %	874
PS EU Stall %	875
Ratio to Max Bandwidth, %	875
Ratio to Max Bandwidth, %	875
Ratio to Max Bandwidth, %	876
Render/GPGPU Command Streamer Loaded	876
Samples Blended	876
Samples Killed in PS, pixels	876
Samples Written	876
Sampler Busy	877
Sampler Is Bottleneck	877
Shared Local Memory Read Bandwidth, GB/sec	877
Shared Local Memory Write Bandwidth, GB/sec	877
SIMD Width	878
Stack-to-stack Incoming Bandwidth	878
Stack-to-stack Outgoing Bandwidth	878
System Memory Read Bandwidth	878

System Memory Write Bandwidth	878
Size	879
Total, GB/sec.....	879
Total Time.....	879
Typed Memory Read Bandwidth, GB/sec.....	879
Typed Memory Write Bandwidth, GB/sec	879
Typed Reads Coalescence.....	880
Typed Writes Coalescence	880
Untyped Memory Read Bandwidth, GB/sec	880
Untyped Memory Write Bandwidth, GB/sec.....	880
Untyped Reads Coalescence	880
Untyped Writes Coalescence	881
VS EU Active	881
VS EU Stall	881
OpenCL™ Kernel Analysis Metrics Reference.....	882
Computing Task Total Time.....	882
Instance Count.....	882
SIMD Width	882
SIMD Utilization.....	883
Work Size	883
Energy Analysis Metrics Reference.....	883
Available Core Time.....	883
C-State.....	883
D0ix States	884
DRAM Self Refresh	884
Energy Consumed (mJ)	884
Idle Wake-ups	885
P-State.....	885
S0ix States	885
Temperature	886
Timer Resolution.....	886
Total Time in C0 State	886
Total Time in Non-C0 States	886
Total Time in S0 State	887
Total Wake-up Count	887
Wake-ups	887
Wake-ups/sec per Core.....	887
Intel Processor Events Reference.....	887
Notices and Disclaimers.....	888

Intel® VTune™ Profiler User Guide

1

This document explains how you use Intel VTune Profiler to profile serial and multithreaded applications on CPU, GPU, and FPGA platforms. You can run Intel VTune Profiler to profile local and remote application targets on Windows*, Linux*, and Android* platforms.

Download Here

Download VTune Profiler from these sources:

- [Standalone version](#)
- [As part of Intel® oneAPI Base Toolkit](#)

NOTE

You can download older versions of documentation for VTune Profiler from the [documentation archive](#).

Start Here

- [Introduction](#)
- [Get Started](#)
- [Tutorials and Samples](#)
- [Performance Analysis Cookbook](#)
- [Intel VTune Profiler Installation Guide](#)

Introduction

Intel® VTune™ Profiler is a performance analysis tool for serial and multithreaded applications. Use VTune Profiler to analyze your choice of algorithm. Identify potential benefits for your application from available hardware resources.

Use VTune Profiler to locate or determine:

- The most time-consuming (hot) functions in your application and/or on the whole system
- Sections of code that do not effectively utilize available processor time
- The best sections of code to optimize for sequential performance and for threaded performance
- Synchronization objects that affect the application performance
- Whether, where, and why your application spends time on input/output operations
- Whether your application is CPU or GPU bound and how effectively it offloads code to the GPU
- The performance impact of different synchronization methods, different numbers of threads, or different algorithms
- Thread activity and transitions
- Hardware-related issues in your code such as data sharing, cache misses, branch misprediction, and others

Usage Models

- [Install](#) VTune Profiler on Windows* or Linux* platforms and use it to analyze local and remote target systems.
- Use the GUI or run analyses from the command line interface (`vtune`) to collect data and perform regression testing.
- Use VTune Profiler as a [web server](#). This is an optimal solution for multi-user environments.

- Install the standalone GUI client or integrate VTune Profiler into IDEs, such as Microsoft Visual Studio* or Eclipse*.

NOTE

Documentation for versions of VTune Profiler prior to the 2021 release are available for download only. For a list of available documentation downloads by product version, see these pages:

- [Download Documentation for Intel Parallel Studio XE](#)
- [Download Documentation for Intel System Studio](#)

Key Features

This table summarizes the availability of important analysis types per host and remote target platform using VTune Profiler:

Analysis	Windows Target	Linux Target	Android Target	FreeBSD* Target
Hotspots analysis	+	+	+	
Threading analysis	+	+		
Remote analysis	+	+	+	+
Analysis in/from containers		+		
IDE (Eclipse*/Visual Studio*)	+	+		
HPC Performance Characterization analysis	+	+		
Microarchitecture Exploration	+	+	+	+
Memory Access analysis	+	+		
Memory Consumption analysis		+		
Input and Output analysis		+		+ ⁴
System Overview analysis		+		
Custom analysis	+	+	+	+
GPU analysis	+	+ ²	+	
VTune Profiler-Platform Profiler analysis	+ ¹	+ ¹		
OpenCL™ kernel analysis	+	+ ²		
Intel Media SDK program analysis		+ ²		
Java* code analysis	+	+	+	
.NET* code analysis	+			
Python* code analysis	+	+		
Go* application analysis	+ ³	+ ³		
OpenMP* analysis	+	+		
MPI analysis	+	+		

Analysis	Windows Target	Linux Target	Android Target	FreeBSD* Target
KVM Guest OS analysis		+		
Ftrace* events analysis		+	+	
Atrace* events analysis			+	
Energy analysis (visualization only)	+	+	+	

¹Preview only; ²Intel HD Graphics and Intel Iris® Graphics only; ³EBS analysis only; ⁴Hardware event-based metrics only, excl. MMIO accesses, DPDK, SPDK

VTune Profiler provides features that facilitate the analysis and interpretation of the results:

- **Top-down tree analysis:** Use to understand which execution flow in your application is more performance-critical.
- **Timeline analysis:** Analyze thread activity and the transitions between threads.
- **ITT API analysis:** Use the ITT API to mark significant transition points in your code and analyze performance per frame, task, and so on.
- **Architecture diagram:** Analyze GPU OpenCL™ applications by exploring the GPU hardware metrics per GPU architecture blocks.
- **Source analysis:** View source with performance data attributed per source line to explore possible causes of an issue.
- **Comparison analysis:** Compare performance analysis results for several application runs to localize the performance changes you got after optimization.
- **Start data collection paused mode:** Click the **Start Paused** button on the command bar to start the application without collecting performance data and click the **Resume** button to enable the collection at the right moment.
- **Grouping:** Group your data by different granularity in the grid view to analyze the problem from different angles.
- **Viewpoints:** Choose among preset configurations of windows and panes available for the analysis result. This helps focus on particular performance problems.
- **Hot keys to start and stop the analysis:** Use a batch file to create hot keys to start and stop a particular analysis.

Caution

Because VTune Profiler requires specific knowledge of assembly-level instructions, its analysis may not operate correctly if a program (target) is compiled to generate non-Intel architecture instructions. In this case, run the analysis with a target executable compiled to generate only Intel instructions. After you finish using VTune Profiler, you can use optimizing compiler options that generate non-Intel architecture instructions.

See Also

[Get Started with Intel® VTune™ Profiler](#)

[Install Intel® VTune™ Profiler](#)

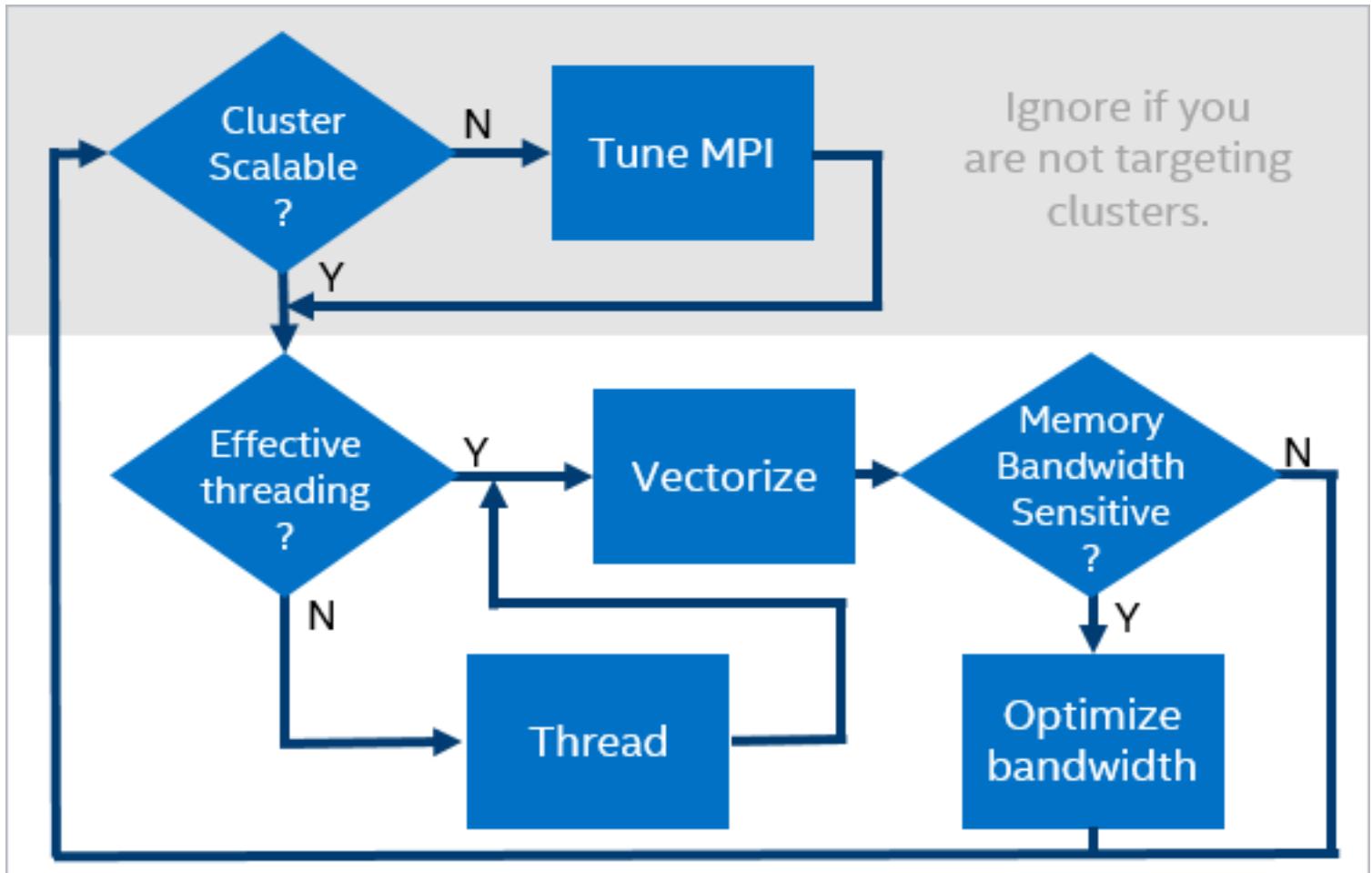
[Microsoft Visual Studio* Integration](#)

[Intel® VTune™ Profiler Graphical User Interface](#)

[Intel® VTune™ Profiler Command Line Interface](#)

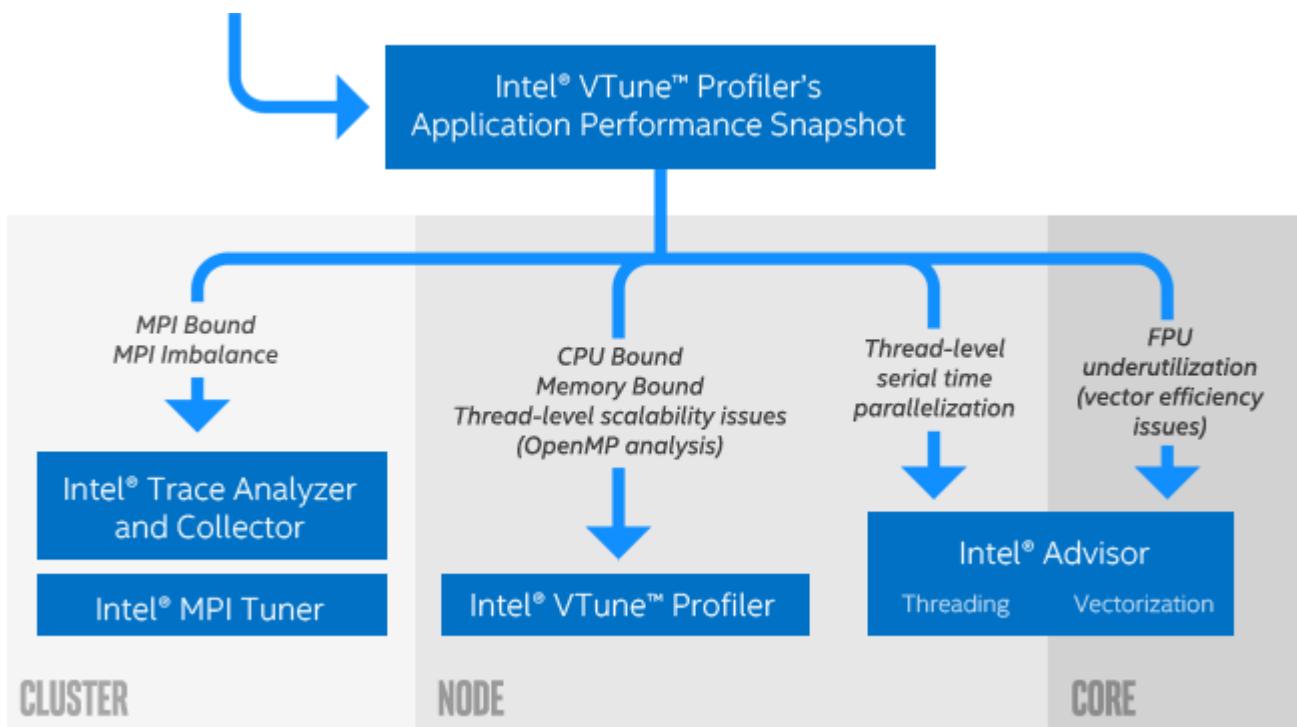
Tuning Methodology

When optimizing your code for parallel hardware, consider using the following iterative approach:



Ignore the top two elements if you are not running on a cluster. There is not a recommended start point what to optimize first as this may vary. Pop up a level, look at all the potential optimizations and see where you can get the biggest gain for the least work. That is where you want to start.

Use these Intel performance analysis tools for the performance optimization workflow:



Explore available performance analysis and tuning scenarios with VTune Profiler provided in:

- [Tutorials](#)
- [Performance Analysis Cookbook](#)
- [Profiling Scenarios](#) for managed code and applications using Intel® runtime libraries
- [Tuning Guides](#)

Tutorials and Samples

Intel® VTune™ Profiler provides web tutorials using sample code for step-by-step instructions on configuring and interpreting performance analysis.

Analyze Common Performance Bottlenecks - C++ Sample Code

Linux* Tutorial: [HTML](#)

Windows* Tutorial: [HTML](#)

Sample: pre-built matrix C++ matrix multiplication application. The pre-built application is available from the **Project Navigator** when you first launch Intel VTune Profiler. You can access the sample code from:

- Linux: <`install-dir`>/samples/en/C++/matrix
- Windows: <`install-dir`>\samples\en\C++\matrix

Learning Objective:

- **Demonstrates:** Iterative application optimization with VTune Profiler, finding algorithmic and hardware utilization bottlenecks
- **Performance issues:** memory access, vectorization
- **Analyses used:** Performance Snapshot, Hotspots, Memory Access, HPC Performance Characterization, Microarchitecture Exploration

Analyzing an OpenMP* and MPI Application - C++ Sample Code

Linux* Tutorial: [HTML](#)

Sample: heart_demo C++ application that simulates electrophysiological heart activity. You can access the sample code at https://github.com/AlexeyMalkhanov/Cardiac_demo.

Learning Objective:

- **Demonstrates:** Identifying issues in a hybrid OpenMP and MPI application.
- **Analysis/tools used:** Application Performance Snapshot (APS), Intel Trace Analyzer and Collector, and VTune Profiler's HPC Performance Characterization analysis

Performance Analysis Cookbook

For end-to-end tuning and configuration use cases, explore the *VTune Profiler Performance Analysis Cookbook* that introduces such recipes as:

- Tuning Recipes:
 - Frequent DRAM Accesses
 - Remote Socket Accesses
 - OpenMP* Imbalance and Scheduling Overhead
- Configuration Recipes:
 - Profiling in a Docker* Container
 - Profiling a .NET* Core App
 - Profiling JavaScript* Code in Node.js*

See [more recipes here](#).

To install and set up the VTune Profiler sample code:

1. Copy the archive file from the installation directory to a writable directory or share on your system.
2. Extract the sample from the archive.

NOTE

- Samples are non-deterministic. Your screens may vary from the screen shots shown throughout these tutorials.
 - Samples are designed only to illustrate the VTune Profiler features and do not represent best practices for tuning any particular code. Results may vary depending on the nature of the analysis and the code to which it is applied.
-

See Also

[Getting Help](#)

[Video and Articles](#)

[Microsoft Visual Studio* Integration](#)

Notational Conventions

The following conventions may be used in this document.

Convention	Explanation	Example
<i>Italic</i>	Used for introducing new terms, denotation of terms, placeholders, or titles of manuals.	The filename consists of the <i>basename</i> and the <i>extension</i> . For more information, refer to the <i>Intel® Linker Manual</i> .

Convention	Explanation	Example
Bold	Denotes GUI elements	Click Cancel .
>	Indicates a menu item inside a menu.	File > Close indicates to select Close from the File menu.
Monospace	Indicates directory paths and filenames, or text that can be part of source code.	ippsapi.h \alt\include Use the <code>okCreateObjs()</code> function to... <code>printf("hello, world\n");</code>
*	An asterisk at the end of a word or name indicates it is a third-party product trademark.	OpenMP*

Get Help

Use these documents and resources to better understand functionality in Intel® VTune™ Profiler:

- [Installation Guides](#)
- [Get Started Guide](#)
- [User Guide](#)
- [Tutorials and Cookbook](#)
- [Articles, Webinars, and Videos](#)
- [Intel Processor Event Reference](#)
- [Release Notes](#)

NOTE

All documentation for VTune Profiler is available online in the [Intel Software Documentation Library](#) on Intel Developer Zone (IDZ). You can also [download an offline version](#) of the VTune Profiler documentation.

Access Documentation

Access product documentation through one of these ways:

- For the cross-platform standalone user interface of the VTune Profiler: Click the



menu button and select **Help > documentation_format** or click the



Help button on the product toolbar.

- Windows* only: For the VTune Profiler integrated into the Visual Studio user interface, select **Intel VTune Profiler version > documentation_format** from the **Help** menu or click the product icon on the toolbar.

NOTE

- VTune Profiler is shipped as a standalone version and as part of Intel oneAPI Base Toolkit. Access to VTune Profiler documentation may vary depending on the product shipment.
- You need an internet connection to access all VTune Profiler documentation formats listed in the menu.
- Google* Chrome* is the recommended browser to view a downloaded copy of the VTune Profiler documentation. If you use Microsoft* Internet Explorer* or Microsoft Edge* browser, you may encounter these issues:
 - Internet Explorer 11: No help topics show up when you select them in the TOC pane.
Solution: Add `http://localhost` to the list of trusted sites in the **Tools > Internet Options > Security** tab. You can remove the site when you finish viewing the documentation.
 - Microsoft Edge: Help panes are truncated and a proper style sheet is not applied.
Solution: Click the Menu <...> and select **Open with Internet Explorer**.

Installation Guides

[Installation Guides](#) contain installation instructions for installing the product and post-installation configuration steps.

Get Started Guide

VTune Profiler provides a [Get Started guide](#) that includes a brief product introduction, provides a basic usage flow and links to additional resources, like Tutorials using a variety of tuning scenarios for sample applications. This guide automatically opens after product installation. You can also access this document through the **Help** menu/toolbar button or Get Started link on the Welcome page.

VTune Profiler User Guide

VTune Profiler User Guide documents concepts, procedures, and reference information required to successfully work with the product. The User Guide is available from the [Intel Software Documentation Library](#) on the web and accessible via the **Help** menu or the



Help toolbar button.

Context-Sensitive Help

Access help topics on active GUI elements through context-sensitive help configured in VTune Profiler. These features are available on a product-specific basis:

- **Learn more | F1** button |



Context Help button provide help for an active dialog box, property page, pane, or window.

- **What's This Column:** In the grid, right-click a performance metric column and select the **What's This Column** entry from the context menu to open a help topic for this particular metric. You can also view a lightweight metric description in the pop-up window when hovering over the column name.

The screenshot shows a table with performance metrics. The columns are: DTLB Overhead, Loads Blocked by Store Forwarding, Lock Latency, Split Loads, 4K Aliasing, and L2. The first row has values: 0.055, 0.000, 0.000, 0.000, 0.000, and L2. A mouse cursor is hovering over the 'What's This Column?' button in the 'Loads Blocked by Store Forwarding' cell. An overlay window titled 'Loads Blocked by Store Forwarding' provides a metric description. The description explains that to streamline memory operations in the pipeline, a load can avoid waiting for memory if a prior store, still in flight, is writing the data that the load wants to read (store forwarding process). However, in some cases, when the prior store is writing a smaller region than the load is reading, the load is blocked for a significant time pending the store forward. This metric measures the performance penalty of such blocked loads.

Help Tour

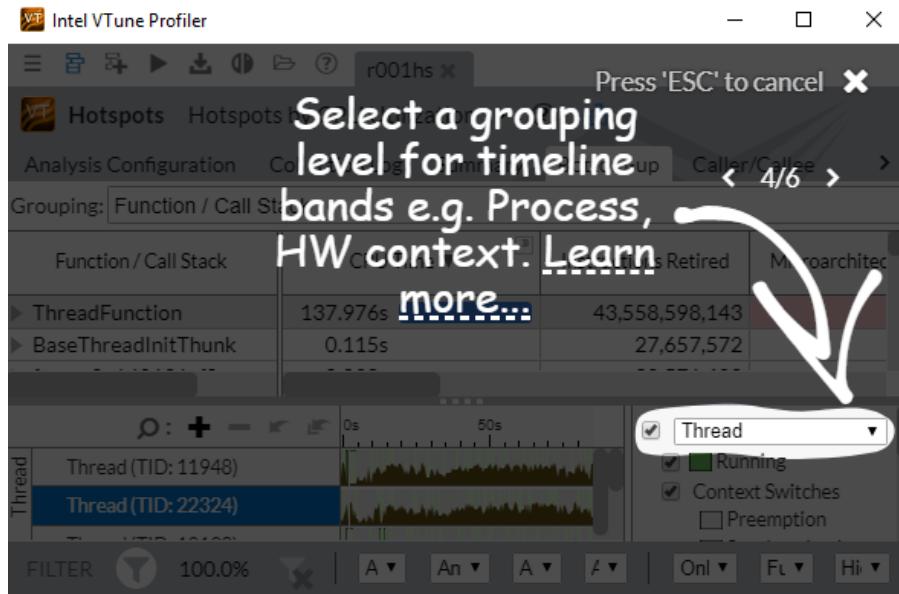
Use the **Help Tour** on the Welcome page to get started with Intel® VTune™ Profiler and understand its interface. The tour uses a sample project to guide you through a typical workflow.

Overlays

In some windows, an overlay outlines useful tips to manage analysis data and enhance your experience. Where available, click the



icon for a tour of useful features in the analysis window.



Tutorials and Cookbook

VTune Profiler provides 15-minute tutorials that show you how to use basic or advanced product features with a short sample. The tutorials provide an excellent foundation before you read the VTune Profiler help. For details, see the [Tutorials and Samples](#) topic.

For featured tuning and configuration scenarios, explore the [Intel® VTune™ Profiler Performance Analysis Cookbook](#).

Command Line Interface Cheat Sheet

Use the Command Line Interface [Cheat Sheet PDF](#) for quick reference on VTune Profiler CLI.

Articles, Webinars, and Videos

Access a library of articles and video content that can help you complete specific tasks with VTune Profiler.

- [Articles](#)
- [Webinars](#) - Detailed video content that illustrate workflows and methodologies.
- [How-to Videos](#) - Short instructional videos to guide you with common tasks with VTune Profiler

Intel Processor Event Reference

VTune Profiler documentation includes [Reference for Intel processor events](#). To access the Reference for a particular Intel processor/microarchitecture, select **Intel Processor Event Reference** option from the **Help** menu and choose the required microarchitecture/processor.

You can also find it useful to explore [Tuning Guides for Intel microarchitecture](#) created by Intel architects and available on the web.

Release Notes

[VTune Profiler Release Notes](#) provide the most up-to-date information about the product, including a product description, technical support, system requirements, and known limitations and issues.

See Also

[Tutorials and Samples](#)

Related Information

Product Website and Support

These links provide information and support on Intel® VTune™ Profiler.

Website	Description
https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.htm	The Intel VTune Profiler product page contains links to the product downloads (standalone or with the Intel® oneAPI Base Toolkit), documentation, forums and other support resources.
https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html	The Intel® oneAPI Base Toolkit support page has links to support forums, startup help, knowledge base, and a video on getting started with the toolkit.
https://www.intel.com/content/www/us/en/developer/get-help/overview.html	Find technical support information to register your product or contact Intel.
https://www.intel.com/content/www/us/en/developer/articles/guide/processor-specific-performance-analysis-papers.html	This site features a collection of tuning guides and performance analysis papers.
https://community.intel.com/t5/Analyzers/bd-p/analyzers	Discuss your experience with Intel VTune Profiler in the Analyzers Developer Forum .

For additional support information, see the Technical Support section of your Release Notes.

System Requirements

To understand hardware and software requirements for the use of Intel® VTune™ Profiler, see the Intel® VTune™ Profiler [System Requirements](#). For the information on the new features and known issues, see the [Release Notes](#).

Related Information

For better understanding of the performance data provided by the Intel® VTune™ Profiler, you are highly recommended to explore additional resources on the web.

Intel® Processor Information

For the most updates, errata, and the latest information on Intel processors, explore the resources available at <https://www.intel.com/content/www/us/en/develop/articles/intel-sdm.html>. The following sections describe processor manuals for Intel 64, IA-32 architecture processors and for Intel Itanium® processors.

Intel 64 and IA-32 Architectures Manuals

The Intel 64 and IA-32 Architectures Software Developer's Manual consists of the following volumes that describe the architecture and programming environment of all Intel 64 and IA-32 architecture processors:

- **Volume 1** describes the architecture and programming environment of processors supporting IA-32 and Intel 64 architectures.
- **Volume 2** includes the full Instruction Set Reference, A-Z, in one volume. Describes the format of the instruction and provides reference pages for instructions.

- **Volume 3** includes the full System Programming Guide, Parts 1, 2, and 3, in one volume. Describes the operating-system support environment of Intel 64 and IA-32 Architectures, including: memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, VMX instructions, and Intel Virtualization Technology (Intel VT).
- **Intel 64 and IA-32 Architectures Software Developer's Manual Documentation Changes** section describes bug fixes made to the Intel 64 and IA-32 Software Developer's Manual between versions.

NOTE

This Change Document applies to all Intel 64 and IA-32 Software Developer's Manual sets (combined volume set, 3 volume set and 7 volume set).

Please refer to all volumes when evaluating your design needs.

For more information on processor-specific performance analysis, explore articles and tuning guides available for download at <http://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers/>.

Multithreading

You are strongly encouraged to read the following books for in-depth understanding of threading. Each book discusses general concepts of parallel programming by explaining a particular programming technology:

Technology	Resource
Intel Threading Building Blocks	Reinders, James. <i>Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism</i> . O'Reilly, July 2007 (http://oreilly.com/catalog/9780596514808/)
OpenMP* technology	Chapman, Barbara, Gabriele Jost, Ruud van der Pas, and David J. Kuck (foreword). <i>Using OpenMP: Portable Shared Memory Parallel Programming</i> . MIT Press, October 2007 (http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11387)
Microsoft Win32* Threading	Akhter, Shameem, and Jason Roberts. <i>Multi-Core Programming: Increasing Performance through Software Multithreading</i> , Intel Press, April 2006 (http://www.intel.com/intelpress/sum_mcp.htm).

Intel Analyzers

Explore more profiling and optimization opportunities with Intel performance analysis tools:

- [Intel Advisor](#) to design your code performance on Intel hardware with the roofline methodology and explore potential for vectorization, threading, and offload optimizations.
- [Intel Inspector](#) to analyze your code for threading, memory, and persistent memory errors.
- [Intel Graphics Performance Analyzers](#) to analyze performance of your game applications (system, frame, and trace analysis).

Install Intel® VTune™ Profiler

Download and install Intel® VTune™ Profiler on your system to gather performance data, either on your native system or on a remote system. You can install the application on Linux*, Windows*, or macOS* host systems but you can collect performance data on remote Windows or Linux target systems only.

System Requirements

To verify hardware and software requirements for your VTune Profiler download, see [Intel® VTune™ Profiler System Requirements](#).

Download Intel VTune Profiler

Download VTune Profiler from these sources:

- [Standalone version](#)
- [As part of Intel® oneAPI Base Toolkit](#)

NOTE

You can download older versions of documentation for VTune Profiler from the [documentation archive](#).

Installation Information

Whether you downloaded Intel® VTune™ Profiler as a standalone component or with the Intel® oneAPI Base Toolkit, the default path for your <install-dir> is:

Operating System	Path to <install-dir>
Windows* OS	<ul style="list-style-type: none"> • C:\Program Files (x86)\Intel\oneAPI\ • C:\Program Files\Intel\oneAPI\ <p>(in certain systems)</p>
Linux* OS	<ul style="list-style-type: none"> • /opt/intel/oneapi/ for root users • \$HOME/intel/oneapi/ for non-root users
macOS*	/opt/intel/oneapi/

For OS-specific installation instructions, refer to the [VTune Profiler Installation Guide](#).

See Also

[Sampling Drivers](#)

[Cookbook: Profiling Hardware Without Drivers](#)

Sampling Drivers

Intel® VTune™ Profiler uses kernel drivers to enable the [hardware event-based sampling](#). VTune Profiler installer automatically uses the Sampling Driver Kit to build drivers for your kernel with the default installation options. If the drivers were not built and set up during installation (for example, lack of privileges, missing kernel development RPM, and so on), VTune Profiler provides an error message and, on Linux* and Android* systems, enables [driverless sampling data collection](#) based on the Linux Perf* tool functionality, which has some analysis limitations for a non-root user. VTune Profiler also automatically uses the driverless mode on Linux when hardware event-based sampling collection is run with stack analysis, for example, for Hotspots or Threading analysis types.

If not used by default, you may still enable a driver-based sampling data collection by building/installing the sampling drivers for your target system:

- [Windows* targets](#): Verify the sampling driver is installed correctly. If required, install the driver.
- [Linux* targets](#):
 - Make sure the driver is installed.

- Build the driver, if required.
- Install the driver, if required.
- Verify the driver configuration.
- **Android* targets:** Verify the sampling driver is installed. If required, build and install the driver.

NOTE

- You may need kernel header sources and other additional software to build and load the kernel drivers on Linux. For details, see the `README.txt` file in the `sepdk/src` directory.
 - A Linux kernel update can lead to incompatibility with VTune Profiler drivers set up on the system for event-based sampling (EBS) analysis. If the system has installed VTune Profiler boot scripts to load the drivers into the kernel each time the system is rebooted, the drivers will be automatically re-built by the boot scripts at system boot time. Kernel development sources required for driver rebuild should correspond to the Linux kernel update.
 - If you loaded the drivers but do not use them and no collection is happening, there is no execution time overhead of having the drivers loaded. The memory overhead is also minimal. You can let the drivers be loaded at boot time (for example, via the `install-boot-script`, which is used by default) and not worry about it. Unless data is being collected by the VTune Profiler, there will be no latency impact on system performance.
-

See Also

[Cookbook: Profiling Hardware without Sampling Drivers](#)

[Embedded Linux* Targets](#)

[Configure Yocto Project* and Intel® VTune™ Profiler with the Linux* Target Package](#)

[Error Message: No Pre-built Driver Exists for This System](#)

Set Up System for GPU Analysis

To analyze Intel HD and Intel Iris Graphics (further: Intel Graphics) hardware events on a GPU,

- Your system must have the Intel Metric Discovery (MD) API library installed on it.
- You need relevant permissions.

Install Intel Metric Discovery API Library on Windows* OS

On Windows, Intel Metric Discovery API library is part of the official Intel Graphics driver package. You can install a driver for your system from <https://www.intel.com/content/www/us/en/download-center/home.html>.

NOTE

If you run GPU analysis via a Remote Desktop connection, make sure your software fits these requirements:

- Intel® Graphics driver version 15.36.14.64.4080, or higher
 - target analysis application runnable via RDC
-

Install Intel Metrics Discovery API Library on Linux* OS

Intel Metrics Discovery API library is supported on Linux operating systems with kernel version 4.14 or newer. If VTune Profiler [cannot collect GPU hardware metrics](#) and provides a corresponding error message, make sure you have installed the API library correctly.

You can download Intel Metrics Discovery API library from <https://github.com/intel/metrics-discovery>.

Enable Permissions

Typically, you should run the GPU Offload and GPU Compute/Media Hotspots analyses with root privileges on Linux or as an Administrator on Windows.

If you lack root permissions on Linux, enable collecting GPU hardware metrics for non-privileged users. Follow these steps:

- Add your username to the `video` and `render` groups.

To check whether your username is part of the `video` group, enter: `groups | grep video`.

To add your username to the `video` group, enter: `sudo usermod -a -G video <username>`.

- Set the value of `dev.i915.perf_stream_paranoid` option to 0 as follows:

```
sysctl -w dev.i915.perf_stream_paranoid=0
```

This command makes a temporary change that is lost after reboot. To make a permanent change, enter:

```
echo dev.i915.perf_stream_paranoid=0 > /etc/sysctl.d/60-mdapi.conf
```

- Since GPU analysis relies on the Ftrace* technology, use the [prepare_debugfs.sh](#) script that sets read/write permissions to debugFS.

Enable GPU utilization events (i915 ftrace events)

If you are only looking to see high level information about GPU utilization, you do not need to reconfigure the kernel.

To analyze detailed GPU utilization metrics on Linux, you may need to rebuild the kernel. Because the i915 driver has to provide low-level tracing events, for kernels 4.14 and newer, enable tracing events using these kernel configuration options:

```
CONFIG_EXPERT=y
```

```
CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=y
```

To check the current state of the `CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS` option, enter:

```
grep CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS /boot/config-$(uname -r)
```

If the option is disabled, you need to [rebuild the i915 driver or the whole kernel](#).

Use the `./install/bin64/prepare-gpu-hardware-metrics.sh` script to automatically enable permissions for non-privileged users.

See Also

[Rebuild and Install the Kernel for GPU Analysis](#)

[GPU Architecture Terminology for Intel® Xe Graphics](#)

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

[Problem: No GPU Utilization Data Is Collected](#)

Rebuild and Install the Kernel for GPU Analysis

To collect i915 ftrace events that are required for a detailed analysis of [GPU utilization](#), your Linux kernel should be properly configured.

If VTune Profiler cannot start an analysis and you see an error message (*Collection of GPU usage events cannot be enabled. i915 ftrace events are not available*), you must rebuild and install the re-configured module i915.

NOTE Rebuilding the Linux kernel is only required if you need to see detailed information about [GPU utilization](#). You can run GPU analyses and see high level information about GPU utilization without rebuilding your Linux kernel.

For kernel versions 4.14 and newer, enable these settings:

- CONFIG_EXPERT=y
- CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=y

If you update the kernel rarely, it is sufficient to configure and rebuild only module i915.

If you update the kernel often, build the special kernel for GPU analysis. Follow this procedure.

NOTE

Installing the kernel requires root permissions.

1. Add source package repositories for your Ubuntu version.

For example, on Ubuntu Bionic Beaver* add:

```
sudo add-apt-repository -s "deb http://ru.archive.ubuntu.com/ubuntu/ bionic main restricted"
```

2. Install build dependencies:

```
sudo apt -y build-dep linux linux-image-$(uname -r)
sudo apt -y install libncurses-dev flex bison openssl libssl-dev dkms libelf-dev libudev-dev
libpci-dev libiberty-dev autoconf
```

3. Install kernel headers:

```
sudo apt -y install linux-headers-$(uname -r)
```

4. Create a folder for kernel source:

```
mkdir -p /tmp/kernel
cd !$
```

5. Download kernel sources:

```
apt -y source linux
cd linux-*
```

If you have a custom kernel, you need to find the corresponding source code the kernel belongs to.

6. Create a .config file with the same configuration you have for your running kernel:

```
cp /boot/config-$(uname -r) .config
make olddefconfig
```

7. In the new .config file, make sure the following settings are enabled:

```
CONFIG_EXPERT=y
```

```
CONFIG_FTRACE=y
```

```
CONFIG_DEBUG_FS=y
```

`CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=y`

Update the file, if required, and save.

8. Create a full `.config` file for the kernel:

```
make olddefconfig
```

9. Build `objtool`. This tool is required for building the sampling driver.

```
make -C tools/ objtool
```

10. Build the kernel with the new `.config` file:

```
make -j `getconf _NPROCESSORS_ONLN` deb-pkg
```

If you are using a custom kernel, use this command instead:

```
make LOCALVERSION= -j `getconf _NPROCESSORS_ONLN` deb-pkg
```

11. Install the kernel and kernel modules:

```
sudo dpkg -i linux-*.deb
```

12. Reboot the machine with the new kernel.

See Also

[Rebuild and Install Module i915 for GPU Analysis on CentOS*](#)

[Rebuild and Install Module i915 for GPU Analysis on Ubuntu*](#)

[GPU Compute/Media Hotspots Analysis \(Preview\)](#)

[Error Message: Cannot Collect GPU Hardware Metrics](#)

Rebuild and Install Module i915 for GPU Analysis on CentOS*

NOTE Profiling support for CentOS* 7 is deprecated and will be removed in a future release.

To collect i915 ftrace events required to analyze the [GPU utilization](#), your Linux kernel should be properly configured. If the Intel® VTune™ Profiler cannot start an analysis and provides an error message: *Collection of GPU usage events cannot be enabled. i915 ftrace events are not available*. You need to rebuild and install the re-configured i915 module. For example, for kernel 4.14 and higher, these settings should be enabled:

`CONFIG_EXPERT=y` and `CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=y`.

If you update the kernel often, make sure to [build the special kernel for GPU analysis](#).

NOTE

Installing the kernel requires root permissions.

On CentOS* systems, if you update the kernel rarely, you can configure and rebuild only module i915 as follows:

1. Install build dependencies:

```
sudo yum install flex bison elfutils-libelf-devel
```

2. Create a folder for kernel source:

```
mkdir -p /tmp/kernel
cd !$
```

3. Get your kernel version:

```
uname -r
```

This is an example of the command output:

```
4.18.0-80.11.2.el8_0.x86_64
```

4. Get source code for the kernel:

```
wget http://vault.centos.org/8.0.1905/BaseOS/Source/SRPMS/kernel-4.18.0-80.11.2.el8_0.src.rpm
```

```
rpm --define "_topdir /tmp/kernel/rpmbuild" -i kernel-4.18.0-80.11.2.el8_0.src.rpm
```

```
tar -xf ./rpmbuild/SOURCES/linux-4.18.0-80.11.2.el8_0.tar.xz
```

5. Change the current directory:

```
cd linux-*
```

6. Configure the kernel modules:

```
cp /usr/src/kernels/$(uname -r)/.config ./
```

```
cp /usr/src/kernels/$(uname -r)/Module.symvers ./
```

7. Update the version in Makefile in the current directory.

The version value must be the same as in the `uname -r` command output. For example, if `uname -r` prints `4.18.0-80.11.2.el8_0.x86_64`, the values in the Makefile should be:

```
VERSION = 4
PATCHLEVEL = 18
SUBLEVEL = 0
EXTRAVERSION = -80.11.2.el8_0.x86_64
```

Update the file, if required, and save it.

8. Make sure the kernel version is set correctly in the Makefile:

```
make kernelversion
```

The command output for the example above is the following:

```
4.18.0-80.11.2.el8_0.x86_64
```

9. In the new .config file, make sure the following settings are enabled:

```
CONFIG_EXPERT=y
CONFIG_FTRACE=y
CONFIG_DEBUG_FS=y
CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=y
```

Update the file, if required, and save it.

10. Create a full config file for the kernel:

```
make olddefconfig
```

11. Build module i915:

```
make -j$(getconf _NPROCESSORS_ONLN) modules_prepare
```

```
make -j$(getconf _NPROCESSORS_ONLN) M=../drivers/gpu/drm/i915 modules
```

If you get the following error:

```
LD [M] drivers/gpu/drm/i915/i915.o
ld: no input files
```

you need to replace the following lines in `scripts/Makefile.build`:

```
link_multi_deps = \
$(filter $(addprefix $(obj)/, \
$($($subst $(obj)///,$(@:.o=-objs)) \
$($($subst $(obj)///,$(@:.o=-y))) \
$($($subst $(obj)///,$(@:.o=-m))))), $^)
```

with the line:

```
link multi deps = $(filter %.o,$^)
```

NOTE

See the patch <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=69ea912fda74a673d330d23595385e5b73e3a2b9> for more information.

12. Install the new module:

```
sudo make M=../drivers/gpu/drm/i915 modules install
```

13. Make sure the folder with the new driver is present in /etc/depmod.d/* files, or just add it:

```
echo "search extradrivers" | sudo tee /etc/modprobe.d/00-extra.conf
```

14. Update initramfs:

```
sudo depmod  
sudo dracut --force
```

15. Reboot the machine:

```
sudo reboot
```

16. Make sure the new driver is loaded:

```
modinfo i915 | grep filename
```

The command output should be the following:

filename: /lib/modules/4.18.0-80.11.2.el8_0.x86_64/extradrivers/gpu/drm/i915/i915.ko

To roll back the changes and load the original module i915:

1. Remove the folder with the new driver from /etc/depmod.d/* files:

```
sudo rm /etc/depmod.d/00-extra.conf
```

2. Update initramfs:

```
sudo depmod  
sudo update-initramfs -u
```

3. Reboot the machine:

`sudo reboot`

GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics

Rebuild and Install Module i915 for GPU Analysis on Ubuntu*

To collect i915 ftrace events required to analyze the [GPU utilization](#), your Linux kernel should be properly configured. If the Intel® VTune™ Profiler cannot start an analysis and provides an error message: *Collection of GPU usage events cannot be enabled. i915 ftrace events are not available.* You need to rebuild and install the re-configured module i915. For example, for kernel 4.14 and higher, these settings should be enabled:

CONFIG_EXPERT=v and CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=v.

If you update the kernel often, make sure to [build the special kernel for GPU analysis](#).

On Ubuntu* systems, if you update the kernel rarely, you can configure and rebuild only module i915 as follows:

NOTE

Installing the kernel requires root permissions.

- 1.** Add source package repositories for your Ubuntu* version.

For example, on Ubuntu Bionic Beaver* add:

```
sudo add-apt-repository -s "deb http://ru.archive.ubuntu.com/ubuntu/ bionic main restricted"
```

- 2.** Install build dependencies:

```
sudo apt -y build-dep linux linux-image-$(uname -r)
sudo apt -y install libncurses-dev flex bison openssl libssl-dev dkms libelf-dev libudev-dev
libpci-dev libiberty-dev autoconf
```

- 3.** Install kernel headers:

```
sudo apt -y install linux-headers-$(uname -r)
```

- 4.** Create a folder for kernel source:

```
mkdir -p /tmp/kernel
cd !$
```

- 5.** Download kernel sources:

```
apt -y source linux
cd linux-*
```

- 6.** Configure the kernel modules:

```
cp /usr/src/linux-headers-$(uname -r)/.config .
cp /usr/src/linux-headers-$(uname -r)/Module.symvers .
```

- 7.** Update the version in Makefile in the current directory.

The version value should be the same as in the `uname -r` command output. For example, if `uname -r` prints `4.15.0-20-generic`, the values in the Makefile must be:

```
VERSION = 4
PATCHLEVEL = 15
SUBLEVEL = 0
EXTRAVERSION = -20-generic
```

Update the file, if required, and save it.

- 8.** Make sure the kernel version is set correctly in the Makefile:

```
make kernelversion
```

The command output for the example above must be:

```
4.15.0-20-generic
```

- 9.** Update the new `.config` file, if required, and save it.

Make sure the following settings in the file are enabled:

```
CONFIG_EXPERT=y
CONFIG_FTRACE=y
CONFIG_DEBUG_FS=y
CONFIG_DRM_I915_LOW_LEVEL_TRACEPOINTS=y
```

- 10.** Create a full config file for the kernel:

```
make olddefconfig
```

- 11.** Build module i915:

```
make -j$(getconf _NPROCESSORS_ONLN) modules_prepare
make -j$(getconf _NPROCESSORS_ONLN) M=../drivers/gpu/drm/i915 modules
```

If you get the following error:

```
LD [M] drivers/gpu/drm/i915/i915.o
ld: no input files
```

you need to replace the following lines in `scripts/Makefile.build`:

```
link_multi_deps = \
$(filter $(addprefix $(obj)/, \
$($($subst $(obj)/,,${@:.o=-objs})) \
$($($subst $(obj)/,,${@:.o=-y})) \
$($($subst $(obj)/,,${@:.o=-m}))), $^)
```

with the line:

```
link_multi_deps = $(filter %.o,$^)
```

NOTE

See the patch <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=69ea912fda74a673d330d23595385e5b73e3a2b9> for more information.

12. Install the new module:

```
sudo make M=./drivers/gpu/drm/i915 modules_install
```

13. Make sure the folder with the new driver is present in `/etc/depmod.d/*` files, or just add it:

```
echo "search extradrivers" | sudo tee /etc/depmod.d/00-extra.conf
```

14. Update initramfs:

```
sudo depmod
sudo update-initramfs -u
```

15. Reboot the machine:

```
sudo reboot
```

16. Make sure the new driver is loaded:

```
modinfo i915 | grep filename
```

The expected command output is the following:

```
filename: /lib/modules/4.15.0-20-generic/extradrivers/gpu/drm/i915/i915.ko
```

To roll back the changes and load the original module i915:

1. Remove the folder with new driver from `/etc/depmod.d/*` files:

```
sudo rm /etc/depmod.d/00-extra.conf
```

2. Update initramfs:

```
sudo depmod
sudo update-initramfs -u
```

3. Reboot the machine:

```
sudo reboot
```

GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics

Verify Intel® VTune™ Profiler Installation

A self-check script is available to validate that appropriate drivers are installed and the system is set up properly to collect performance data. The script can be run on individual systems or on a cluster environment.

The `vtune-self-checker` script is available from `<install-dir>/bin64`[Installation Information](#) on the Windows or Linux system on which you installed VTune Profiler. The script runs several representative analysis types on a sample with reliable hotspots. After the script completes, it produces a log file and gives diagnostics on the success or failure of the checks. The analysis types that are launched cover:

- Software sampling and tracing collection (Hotspots and Threading in the user-mode sampling)
- Core event-based sampling collection (Hotspots in the hardware event-based sampling mode with and without stacks)
- HPC Performance Characterization
- Microarchitecture Exploration analysis
- Memory Access analysis with uncore events
- Threading with hardware event-based sampling
- Performance Snapshot
- GPU Compute/Media Hotspots (source analysis and characterization modes)

The result of the self-check provides these details:

- Analyses that passed the check
- Analyses that failed the check
- Possible collection limitations
- Steps to overcome collection limitations
- Information about missing permissions or outdated drivers

Use the `--log-dir` option when running the script to specify a location for the log file to be stored. This option is useful when running the script on a compute node through a job scheduler.

Install VTune Profiler Server

Set up Intel® VTune™ Profiler as a web server, using a lightweight deployment intended for personal use or a full-scale corporate deployment supporting multi-user environment.

VTune Profiler Server Deployment

Deployment of the VTune Profiler server depends on your usage mode and purpose:

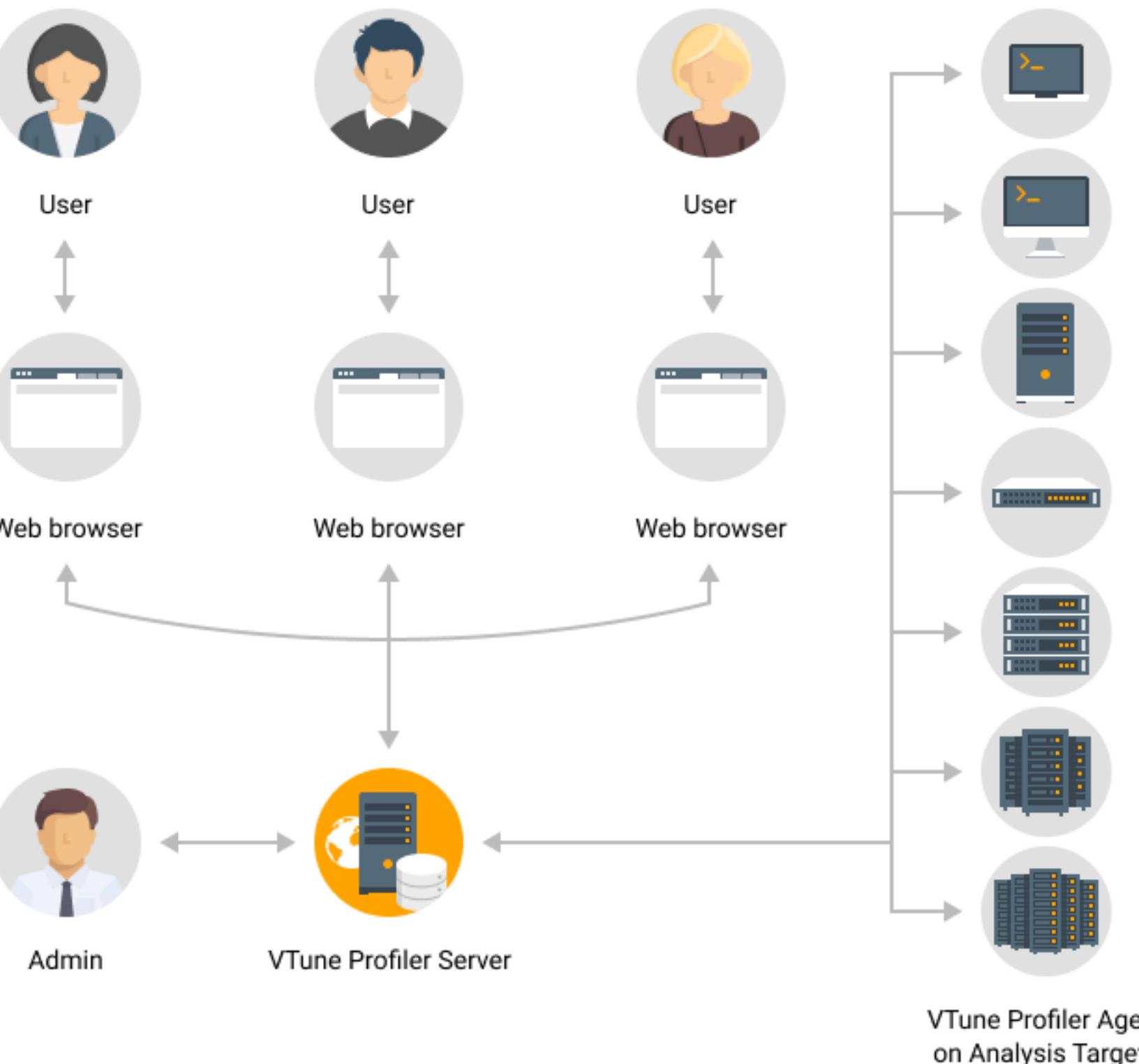
Deployment Mode	Benefits	Limitations
Personal use/evaluation	<ul style="list-style-type: none"> • No host platform setup. VTune Profiler Server is installed as part of the VTune Profiler GUI. • Quick on-boarding experience with self-signed TLS certificates 	<ul style="list-style-type: none"> • Single user mode • Medium security level
Integration with SAML Single-Sign-On (SSO)	<ul style="list-style-type: none"> • Automatic authentication with company accounts 	<ul style="list-style-type: none"> • Mandatory company IT support to register VTune Profiler Server in IT SAML SSO infrastructure

Deployment Mode	Benefits	Limitations
Deployment behind a reverse proxy (NGINX*, Apache* web server, IIS, etc.)	<ul style="list-style-type: none">Support for multi-user environmentHigh security levelSupport for selective access to VTune Profiler Server (for example, per user network group) <ul style="list-style-type: none">Reuse of existing IT web hosting infrastructure (including transport security and user authentication)High security levelSupport for multi-user environment	<ul style="list-style-type: none">DevOps expertise required

Depending on your choice, you can proceed with the next steps:

- [Set up transport security.](#)
- [Configure user authentication/authorization.](#)

How It Works



1. (Reverse proxy and SAML SSO modes) Admin installs a VTune Profiler Server instance in a lab.

2. (Reverse proxy and SAML SSO modes) Admin emails the URL of the installed VTune Profiler Server to the User(s).
3. User accesses the VTune Profiler via a supported web browser, configures and runs analysis on an arbitrary target system.
VTune Profiler Server can be accessed from any client machine.
4. When analysis is initiated, the VTune Profiler Server installs a VTune Profiler Agent on the specified target system. This agent performs collection and uploads results to the VTune Profiler Server for analysis and storage.

Use this glossary of terms for your reference:

VTune Profiler Server	VTune Profiler started as a web server and serving a web site to access the VTune Profiler GUI from remote client machines using a web browser.
User	User of the VTune Profiler Server.
User client system	A machine that the User is logged to and used to access the VTune Profiler Server via a web browser.
Target system	A machine, local or remote, that is profiled with the VTune Profiler.
VTune Profiler Agent	A piece of VTune Profiler software that runs on a target system.

System Requirements

VTune Profiler Server System

- 64-bit Linux* or Windows* OS
- Same system requirements and supported operating system distributions as specified for VTune Profiler command line tool in the [Release Notes](#)

Client System

- Chrome, Firefox or Safari (recent versions)

VTune Profiler Server is tested with the latest versions of supported browsers at the time of each release.

Target System

- 32- or 64-bit Linux or Windows OS
- Same system requirements and supported operating system distributions as specified for VTune Profiler target systems in the [Release Notes](#)

NOTE

VTune Profiler Server currently does not support cross-platform profiling. If the VTune Profiler Server is hosted on a Linux system, then it supports data collection on Linux target systems only. The same is applicable to Windows systems.

See Also

[Web Server Interface](#)

Set Up Transport Security

VTune Profiler Server web site is accessible via encrypted HTTPS connection. HTTPS requires a Transport Layer Security (TLS) certificate. Depending on your [deployment mode](#), you can use different types of TLS certificates.

Self-Signed TLS Certificate

The self-signed certificate is automatically generated when the VTune Profiler Server is started. No additional actions are required from the user who starts the server, but the web browser will provide a warning that the server certificate is not trusted and will ask for a confirmation to proceed.

Signed TLS Certificate

You are recommended to use properly signed TLS certificates so that web browsers automatically validate authenticity of the VTune Profiler Server. Such certificate should be provisioned by your company IT department.

To set up the transport security, the Admin should follow these steps:

1. Provide the signed TLS certificate to users of the VTune Profiler Server.

Make sure to include the VTune Profiler Server DNS name to either **Common Name** or **Alternative Domain Names**.

For example, if the URL to access the VTune Profiler Server is `https://vtune.lab01.myorg.com`, the TLS certificate **Common Name** should be `vtune.lab01.myorg.com`, or `vtune.lab01.myorg.com` should be included into **Alternative Domain Names**.

2. Start the VTune Profiler Server as follows:

```
vtune-backend --tls-certificate /path/to/vtune.lab01.myorg.com.pfx --tls-certificate-password-path /path/to/cert_password.txt
```

You can also enter the certificate password interactively by using the `--tls-certificate-password` option instead of `--tls-certificate-password-path`. In this case, the VTune Profiler Server will prompt to enter the password:

```
vtune-backend --tls-certificate /path/to/vtune.lab01.myorg.com.pfx --tls-certificate-password  
Certificate password:
```

If the certificate private key is stored in a separate file, use the `--tls-certificate-key` option:

```
vtune-backend --tls-certificate /path/to/vtune.lab01.myorg.com.crt --tls-certificate-key /path/to/vtune.lab01.myorg.com.key
```

See Also

[Web Server Interface](#)

Configure User Authentication/Authorization

Use the default passphrase authentication to run the VTune Profiler Server, or benefit from your company solutions with reverse proxy or SAML authentication.

User authentication and authorization for VTune Profiler Server is controlled by a configuration file stored in `<vtune-install-dir>/backend/config.yml`. This configuration file uses YAML format and comes with brief inline documentation describing available configuration options.

Passphrase Authentication

In the default personal use mode, VTune Profiler Server is configured to use passphrase authentication/authorization. When you [start the server](#), you can specify a passphrase:



There are no usernames involved: if the passphrase is shared between multiple users, then they are treated as the same user.

VTune Profiler persists the hash of the passphrase. The browser also persists a secure HTTPS cookie so that you do not enter the passphrase each time. Cookie expiration time is configurable, default value is 365 days. When you access the VTune Profiler Server from a different machine or use a different browser, or if the browser cookies are cleaned / expired, then you are prompted to enter the passphrase again.

If you forget the passphrase, you can reset it by re-running the VTune Profiler Server using the `--reset-passphrase` option. The server provides an outcome URL with a one-time token to reset the passphrase:

```
vtune-backend --reset-passphrase
Serving GUI at https://127.0.0.1:65417?one-time-token=e2ed7c1365c972ec1024ac4e53179a08
```

When you open this URL in a web browser, you are prompted to set a new passphrase.

Reverse Proxy Authentication

VTune Profiler Server can be deployed behind a reverse proxy, which is a web server that forwards all requests to the VTune Profiler Server and serves its responses back to the user. With this type of setup, the system administrator can configure arbitrary user authentication and authorization in the reverse proxy. Reverse proxy is configured to pass authenticated user ID to the VTune Profiler Server, while the VTune Profiler Server is configured to trust this user ID.

To enable the reverse proxy authentication, the administrator needs to follow these steps:

1. Change the authentication type in the `<vtune-install-dir>/backend/config.yml` to `reverse-proxy` and specify the `header`, which is an HTTP header that reverse proxy uses to pass authenticated user ID.
2. Start the VTune Profiler Server as follows:
 - **If VTune Profiler Server and reverse proxy are on the same host:** start the VTune Profiler Server without the `--allow-remote-ui` option to prevent remote connections to be accepted by the VTune Profiler Agent:

```
vtune-backend --web-port=8080
Serving GUI at https://127.0.0.1:8080
warn: Server access is limited to localhost only. To enable remote access, restart with --allow-remote-ui.
```

- **If VTune Profiler Server and reverse proxy are on different hosts:** configure the reverse proxy to use a client certificate authentication when calling the VTune Profiler Server. Provide the VTune Profiler Server with the path to the public part of the reverse proxy client certificate :

```
vtune-backend --allow-remote-ui --client-certificate /path/to/public/reverse/proxy/cert.crt
```

NOTE

You are recommended to use the client certificate authentication even when VTune Profiler Server and the reverse proxy are on the same host to prevent an unauthorized access from the host system.

SAML SSO Authentication

VTune Profiler Server supports SAML 2.0 Single Sign On (SSO) for user authentication.

To enable the SAML SSO authentication, the Admin needs to follow these steps:

1. Change the authentication type in the `<vtune-install-dir>/backend/config.yml` to `saml` and specify the `rootUrl` and the `entityID`.

2. Request the IT service to register the VTune Profiler Server into the SAML SSO infrastructure. The request should include the entity ID, consume URL (`rootUrl + consumePath`), and the name of a network user group to be provided with an access to the VTune Profiler Server.
In response, the IT service provides the entry point for SAML Identity Provider and its public certificate.
3. Enter the data provided with the IT service to the `entryPoint` and `cert` fields in the `config.yml` file.
4. Start the VTune Profiler Server.

See Also

[Web Server Interface](#)

Security Best Practices

Performance profiling is an activity that may involve making important security decisions. Learn about some important security considerations that arise when installing and using Intel® VTune™ Profiler.

Due to the inherent nature of performance profiling, Intel® VTune™ Profiler requires certain levels of access to deliver some of the more advanced features. It is important that you are aware of these implications to enable you to make informed security decisions.

Administrator and Root Privileges

VTune Profiler requires administrator or root privileges for performing specific types of analyses. On Windows* OS, this means starting VTune Profiler as Administrator, and on Linux* systems, this requires sudo privileges.

It is recommended to only start VTune Profiler with elevated privileges if a specific analysis requires these privileges. Avoid staying in elevated mode for viewing collected results.

Controlling Sampling Driver Access (Linux* OS)

By default, on Linux OS, VTune Profiler installer creates a `vtune` user group, which is given access to the Sampling Driver through the Linux* I/O Control. It is recommended to not alter the default settings, for example, by creating a broad user group. Since the driver runs on the kernel level, exposing the driver to a large group of users can make your system vulnerable. Additionally, any user that has access to the driver can potentially obtain sensitive information by collecting performance metrics from the system.

Though VTune Profiler takes preemptive measures by validating all user input, it is recommended that you follow the principle of least required privilege when allowing access to the sampling driver.

Security Implications of Setting `perf_event_paranoid` (Linux* OS)

On Linux OS, the `perf_event_paranoid` setting controls the access levels for unprivileged users of `perf`. VTune Profiler may recommend that you set this value to 0 to perform a specific analysis. At this level, the collected data includes per-process and system-wide performance monitoring data, including CPU and system events both from the user space and the kernel. This may create a potential for sensitive data leaks.

For more information on the usage of `perf` with VTune Profiler and possible limitations, see the [Profiling Hardware Without Intel Sampling Drivers](#) Cookbook recipe.

VTune Profiler Server Authentication Security

Though all network traffic of VTune Profiler Server is encrypted, it is important to select the appropriate authentication scheme when installing VTune Profiler Server. While passphrase authentication is a viable option for some use cases, such as personal use, it is recommended to use other authentication schemes offered when using VTune Profiler Server in broader environments. Detailed information on configuring secure user access channels is available in the [Install VTune Profiler Server](#) section of the User Guide.

Open Intel® VTune™ Profiler

Open Intel® VTune™ Profiler with the graphical user interface (vtune-gui) or command-line interface (vtune).

Once you have downloaded Intel® VTune™ Profiler, follow these steps to run the application:

1. Locate the installation directory.
2. Set environment variables.
3. Open Intel® VTune™ Profiler
 - From the GUI
 - From the command line

Default Installation Paths

Whether you downloaded Intel® VTune™ Profiler as a standalone component or with the Intel® oneAPI Base Toolkit, the default path for your <install-dir> is:

Operating System	Path to <install-dir>
Windows* OS	<ul style="list-style-type: none"> • C:\Program Files (x86)\Intel\oneAPI\ • C:\Program Files\Intel\oneAPI\ <p>(in certain systems)</p>
Linux* OS	<ul style="list-style-type: none"> • /opt/intel/oneapi/ for root users • \$HOME/intel/oneapi/ for non-root users
mac* OS	/opt/intel/oneapi/

NOTE Profiling support for macOS is deprecated and will be removed in a future release.

Set Environment Variables

To set up environment variables for VTune Profiler, run the `setvars` script:

Linux* OS: `source <install-dir>/setvars.sh`

Windows* OS: `<install-dir>\setvars.bat`

When you run this script, it displays the product name and the build number. You can now use the `vtune` and `vtune-gui` commands.

Open VTune Profiler from the GUI

On Windows* OS, use the **Search** menu or locate VTune Profiler from the **Start** menu to run the standalone GUI client.

For the version of VTune Profiler that is integrated into Microsoft* Visual Studio* IDE on Windows OS, do one of the following:

- Select **Intel VTune Profiler** from the **Tools** menu of Visual Studio.
- Click the



Configure Analysis with VTune Profiler toolbar button.

On a macOS* system, start **Intel VTune Profiler version** from the Launchpad.

NOTE

You can also [launch the VTune Profiler from the Eclipse* IDE](#).

Open VTune Profiler from the Command Line

To launch the VTune Profiler from the command line, run the following scripts from the `<install-dir>/bin64` directory:

- `vtune-gui` for the [standalone graphical interface](#)
- `vtune` for the [command line interface](#)

To open a specific VTune Profiler project or a result file, enter:

```
> vtune-gui <path>
```

where `<path>` is one of the following:

- full path to a result file (`*.vtune`)
- full path to a project file (`*.vtuneproj`)
- full path to a project directory. If the project file does not exist in the directory, the **New Project** dialog box opens and prompts you to [create a new project](#) in the given directory.

For example, to open the `matrix` project in the VTune Profiler GUI on Linux, run:

```
vtune-gui /root/intel/vtune/projects/matrix/matrix.vtuneproj
```

See Also

[Web Server Interface](#)

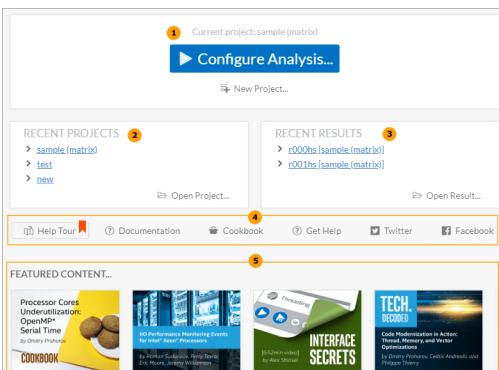
[Install Intel® VTune™ Profiler](#)

[Get Started with Intel® VTune™ Profiler](#)

[Set Up Project](#)

Get Started with Intel® VTune™ Profiler

When you start Intel® VTune™ Profiler, a Welcome page opens with several links to product news, resources, and directions for your next steps.



- 1 To start with VTune Profiler, you need to have a project that specifies a target to analyze.
To [create a new project](#), click the **New Project...** link. If a project is open, its name shows up on the Welcome page as the **Current project**.
To configure and run a new analysis for the current project, click **Configure Analysis...** on the Welcome screen. You also use this selection to configure [target](#) and [analysis](#) settings for a project that is currently open.
The **Configure Analysis** link opens the [Performance Snapshot](#) analysis type by default. This snapshot gives you a quick overview of issues affecting your application performance.
For other analysis types, click the analysis header to open the **Analysis Tree** which displays all available analyses.
- 2 For quick and easy access to an existing project used recently, click the required project name in the **Recent Projects** list. Hover over a project name in the list to see the full path to the project file.
Click **Open Project...** to open an existing project (*.vtuneproj).
- 3 To open a recently collected result, click the required item in the **Recent Results** list. By default, each [result name](#) has an identifier of its analysis type (last two letters in the result name); for example, **tr** stands for Threading analysis. Hover over a result name in the list to see the full path to the result file.
Click **Open Result...** to open a result file (*.vtune).
- 4 Use the link bar to access additional informational resources such as [Performance Analysis Cookbook](#), online product documentation or social media channels. Consider getting started with the product by running the **Help Tour** that guides you through the interface using a sample project.
- 5 Review the latest **Featured Content** that typically includes performance tuning scenarios and tuning methodology articles.

Use the Get Started document to get up and running with a basic Hotspots analysis using your own application on your host system.

- [Windows*](#)
- [Linux*](#)
- [macOS*](#)

NOTE

From a macOS host, you can launch a collection on a remote Linux* system or on an Android* system and view the data collection result on the host. VTune Profiler does not support local analysis on a macOS host.

See Also

[Introduction and Key Features](#)

[Set Up Remote Linux* Target](#)

[Android* Targets](#)

[Intel® VTune™ Profiler Graphical User Interface](#)

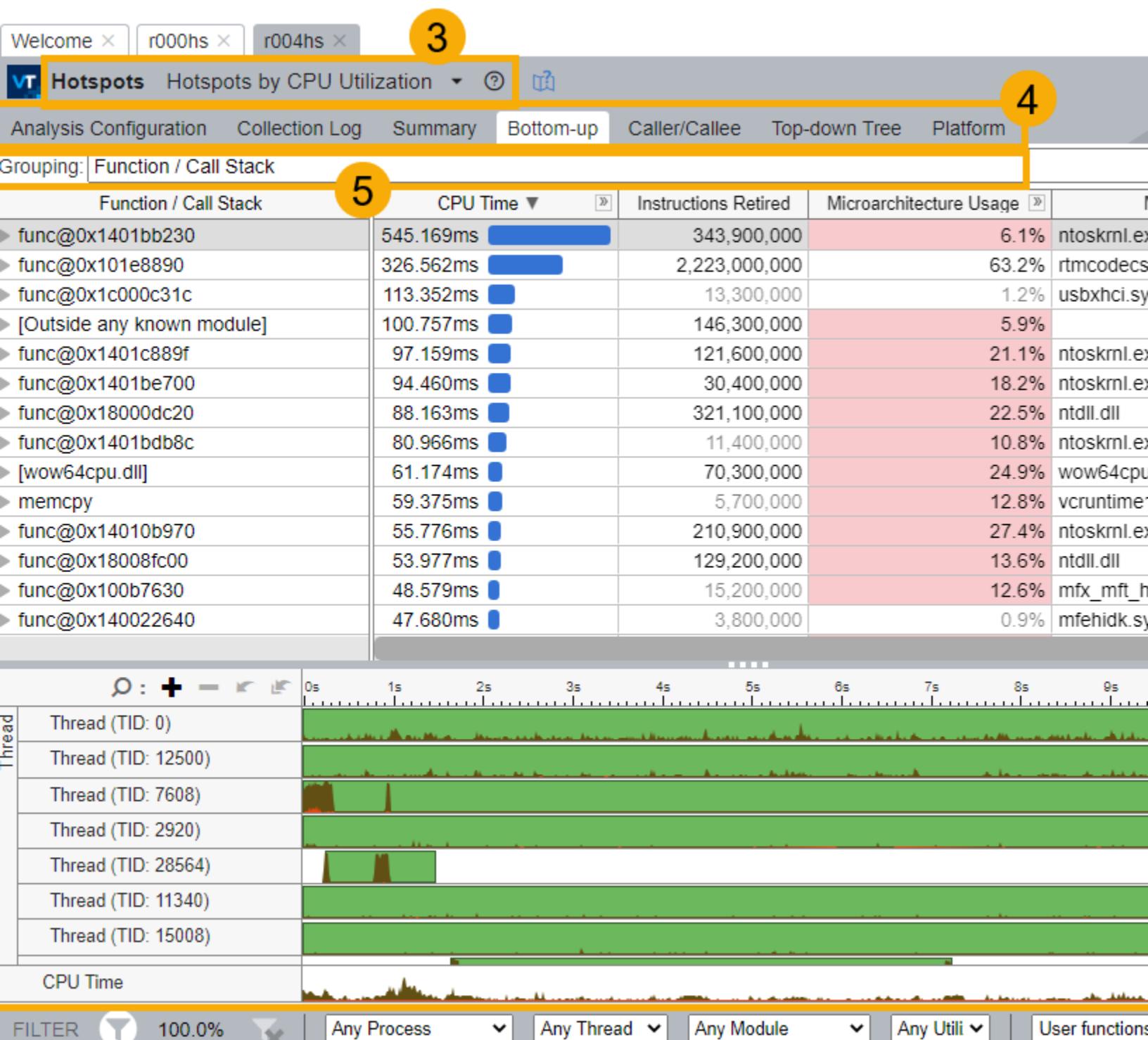
[Intel® VTune™ Profiler Command Line Interface](#)

[Eclipse* and Intel System Studio IDE Integration](#)

Microsoft Visual Studio* Integration

Intel® VTune™ Profiler Graphical User Interface

When you create a project in Intel® VTune™ Profiler, these features help you analyze data:



1 Project Navigator. Use the navigator to [manage your project](#) and collected analysis results.

2 Menu and Toolbar. Use the [VTune Profiler menu](#) and [toolbar](#) to configure and control performance analysis, define and view project properties. Click the



button to open/close the Project Navigator. Use the



Configure Analysis toolbar button to access an analysis configuration.

- 3** **Analysis type and viewpoint.** View the correlation of the analysis result and a viewpoint associated with it. A *Viewpoint* is a pre-set configuration of windows/panes for an analysis result. For most of analysis types, you can click the down arrow to switch between viewpoints and focus on particular performance metrics.
- 4** **Analysis Windows.** Switch between window tabs to explore the analysis type configuration options and collected data provided by the selected viewpoint.
- 5** **Grouping.** Use the **Grouping** drop-down menu to choose a granularity level for *grouping data* in the grid. Available groupings are based on the hierarchy of the program units and let you analyze the collected data from different perspectives; for example, if you are developing specific modules in an application and interested only in their performance, you may select the **Module/Function/Call Stack** grouping and view aggregated data per module functions.
- 6** **Filtering.** VTune Profiler provides two basic options for filtering the collected data: per object and per time regions. Use the *filter toolbar* to filter out the result data according to the selected object categories: Module, Process, Thread, and so on. To filter the data by a time region, select this region on the timeline, right-click and choose **Filter In by Selection** content menu option.

This could be useful, for example, to get region specific data in the context summary for the HPC Performance Characterization or GPU Compute/Media Hotspots analyses.

See Also

[Open Intel® VTune™ Profiler](#)

[Analyze Performance](#)

[Control Data Collection](#)

[Microsoft Visual Studio* Integration](#)

Web Server Interface

Use Intel® VTune™ Profiler in a web server mode to get an easy on-boarding experience, benefit from a collaborative multi-user environment, and access a common repository of collected performance results.

The web server interface helps you quickly get started with the tool since you do not need to install VTune Profiler as a desktop application on every client system. You can use the VTune Profiler Server to configure and control analysis on arbitrary target systems and view collected results.

To run an analysis via a web interface:

1. (Personal mode) [Run the VTune Profiler Server](#) to get a URL to access the web interface.
(Reverse proxy/SAML SSO modes) Get the server URL from your admin.
2. Access the server via the URL.
3. [Deploy the VTune Profiler Agent](#)
4. Select your target system:

- [client system \(localhost\)](#)
- [remote system](#)

5. Run the analysis

To control VTune Profiler agents, use the [Administrator Dashboard](#).

Run VTune Profiler Server

Prerequisite: VTune Profiler Server is installed with VTune Profiler GUI.

In the personal/evaluation usage mode, run the VTune Profiler Server as follows:

1. Start the VTune Profiler Server:

```
<vtune-install-dir>/bin64/vtune-backend
```

If you want the VTune Profiler Server to access a specific TCP port, specify it with the --web-port option. For example:

```
vtune-backend --web-port=8080
```

VTune Profiler Server outputs a URL to access the GUI. For the first run, the URL includes a one-time token. For example:

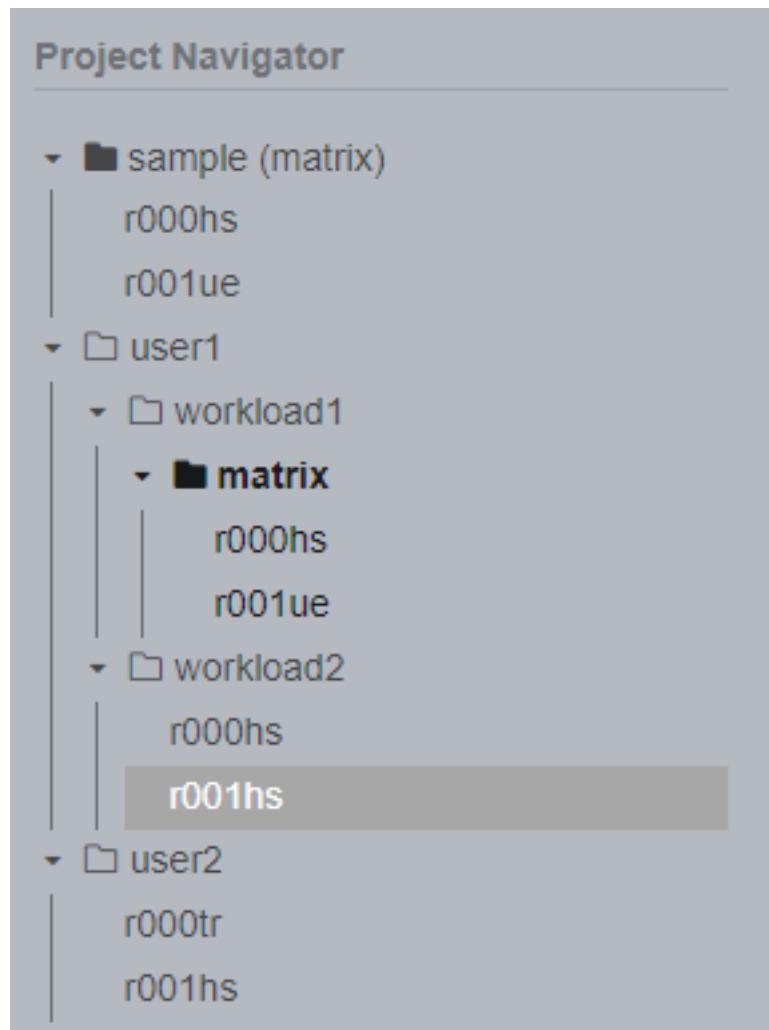
```
Serving GUI at https://127.0.0.1:64880?one-time-token=0160852eef593e0ab0a0f90991b4efa
```

Optionally, you can specify a working directory for VTune Profiler Server using the --data-directory option. For example:

```
vtune-backend --web-port=8080 --data-directory="C:\vtune-results"
```

NOTE Additional command-line options are available to make the usage of VTune Profiler Server in containers more convenient. See [Use VTune Profiler Server in Containers](#) for details.

VTune Profiler Server allows you to create a directory with a custom hierarchy, organized to best fit your needs. Once you point VTune Profiler Server to this directory using the --data-directory option, users will be able to access all projects and results, regardless of folder names and levels of nesting. This can be especially useful if you're using an HPC scheduler to regularly collect VTune Profiler performance data and put it into a shared folder on the network for later examination. For example, you can organize your results folder by users and their workloads:



NOTE

- By default, access to the VTune Profiler Server is limited to the local host only. To enable access from remote client and target systems, restart the server with the `--allow-remote-access` option.
 - By default, server host profiling is not enabled. To enable the server host profiling, restart the server with the `--enable-server-profiling` option.
-

2. Open the URL with the provided one-time token.

NOTE

If you start the VTune Profiler Server in the personal/evaluation mode with no [signed TLS certificate](#) provided, your web browser warns you that the default self-signed server certificate is not trusted and asks for your confirmation to proceed.

3. Set a passphrase in the **Set Passphrase** dialog box.

In the reverse proxy or SAML SSO usage modes, use the URL provided by your admin to access the VTune Profiler Server instance installed in a lab.

Deploy the VTune Profiler Agent

You can choose between automated and manual deployment of the VTune Profiler Agent.

Deploy the Agent automatically

NOTE

VTune Profiler Server uses SSH for automated agent deployment. Running an SSH server on the target machine is required for automated deployment.

To deploy the Agent automatically:

1. Enter the target machine username.
2. Enter the credentials for target machine:
 - For **Public key authentication**, add your public SSH key to the `authorized_keys` file on the target system for the user account that you specify in the **Username** field. Then, select the **Private key** file on your client machine. If your private key is encrypted, specify the **Private key passphrase**.
 - Alternatively, switch to **Password authentication** and provide the username and password.
3. Optionally, specify the deployment directory.
4. Click the **Deploy Agent** button.

The screenshot shows the 'Configure Analysis' interface with the 'WHERE' tab selected. The target host is set to 'localhost'. A message at the top states 'Cannot detect VTune Profiler Agent on the target system.' Below this, a note says 'SSH server is detected on the target system. You can provide credentials to automate deployment of VTune Profiler Agent. You can also download the agent and run it on the target system manually.' There are fields for 'Username' (empty), 'Private key' (choose file, currently 'No file chosen'), 'Private key passphrase' (empty), and 'Deployment directory' (set to 'c:\tmp\vtune-agent'). A checkbox for 'Share the agent with all VTune Profiler users' is unchecked. At the bottom are two buttons: 'Deploy Agent' (blue background) and 'Download Agent Manually'.

Deploy the Agent manually

To deploy the Agent manually:

1. Click the **Download Agent Manually** button In the **WHERE** pane of the **Configure Analysis** window or access the `http://<VTune Profiler Server URL>/api/collection-agent/download` URL to download the Agent.

NOTE

You can use tools such as `wget` to download the Agent directly to the target system.

2. Extract the Agent archive with your tool of choice and copy its contents to the target system.
3. Run the `vtune-agent` executable on the target system and specify the agent owner using the `-owner <vtune-user-id>` option.

NOTE

You can find your VTune Profiler user ID in the **About** dialog.

4. Compare the **Agent key fingerprint** in the **WHERE** pane of the **Configure Analysis** window with the fingerprint printed out by the agent upon startup. If they match, click the **Admit Agent** button.

Shared Agents

You can run a shared VTune Profiler Agent. In this case, the Agent will be available to all users of an instance of VTune Profiler Server. This means that any user of this VTune Profiler Server instance will be able to run data collection using this agent. It is recommended to only run shared agents using dedicated faceless accounts.

To deploy a shared agent, check the **Share the agent with all VTune Profiler users** checkbox in the **WHERE** pane of the **Configure Analysis** dialog, or use the `--shared` command line option when deploying an agent manually.

Select a Client System

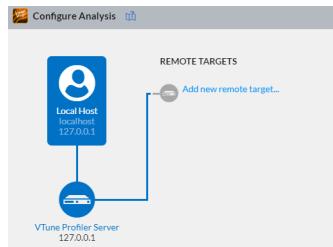
To profile a *client system*, which is the same machine that you use to access the VTune Profiler Server via a web browser, do the following:

1. Click **New Project** and specify a name for the new project.
VTune Profiler opens the project configuration with your **localhost** pre-selected as a target system.
2. Configure your [analysis target](#) and [analysis type](#).

Select a Remote System

To profile a remote target system, do the following:

1. In the **WHERE** pane of the **Configure Analysis** window, click the  down arrow to see available target systems.

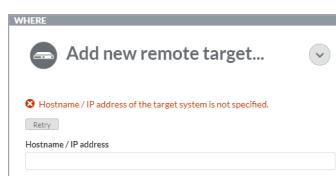


2. Select **Add new remote target....**

NOTE

VTune Profiler maintains a list of used remote systems, if any, and displays it under **Remote Targets**.

3. Enter the hostname or IP address.



Run the Analysis

Once the Agent is running, the **Configure Analysis** pane displays information that VTune Profiler is detecting the device configuration.

The Agent downloads the collectors and the target package, which is approximately 100MB in size. Once the target package is downloaded, the Agent analyzes the target system configuration and displays the applicable analysis types.

To run an analysis:

1. Install the [Intel sampling drivers](#) manually by running these commands:

On Windows* OS:

```
<vtune-agent-dir>\bin64\amplxe-sepreg.exe
```

On Linux* OS:

```
<vtune-agent-dir>/sepdk/src/build-driver
```

```
<vtune-agent-dir>/sepdk/src/insmod-sep
```

The `<vtune-agent-dir>` is the `<vtune_profiler_<version>>` installation folder created on the client system by VTune Profiler.

2. Configure your [analysis target](#) and [analysis type](#).
3. Click the



Start button to run the analysis.

Analyze Process Running Under Arbitrary Account (Linux* OS)

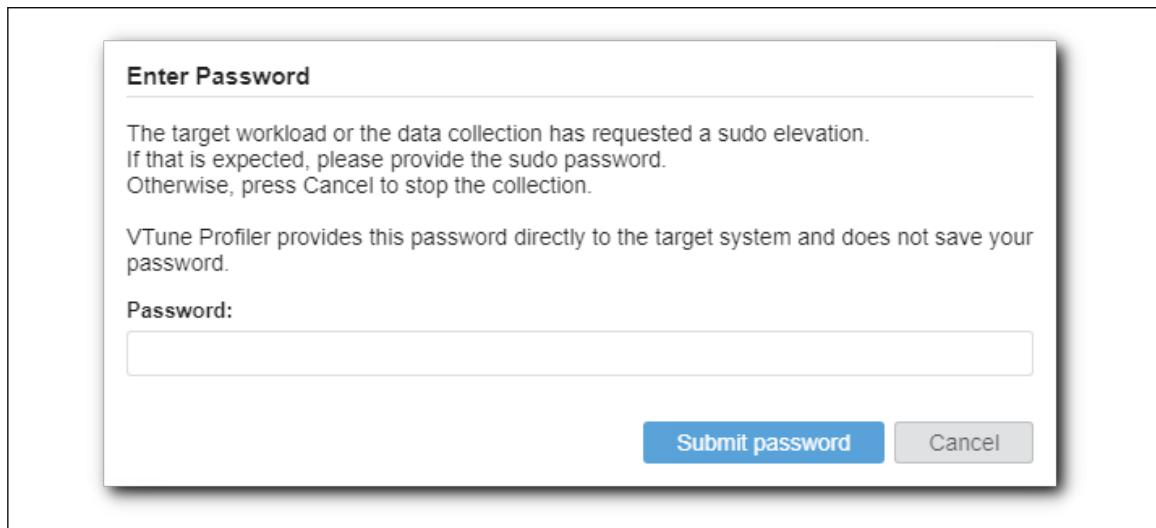
VTune Profiler Server provides a way to analyze a process that is running under an arbitrary user account. A common example is analyzing a process in **Attach to Process** mode that was previously started under an arbitrary user account. The account running the process is not necessarily the same as the account the VTune Profiler Agent was deployed for.

To enable this functionality, provide the following wrapper script in the [Advanced Options](#) section of the **WHAT** pane:

```
#!/bin/sh  
#Run VTune collector as the target process owner  
sudo -C 65000 -A -u <target process owner> "$@"
```

The `sudo` command call runs the VTune Profiler collector under the account specified under `<target process owner>`. Replace this placeholder with the account name under which the target process is running.

If the target workload or the collector request a sudo elevation during the analysis, VTune Profiler Server requests this password interactively in the Web Interface:



NOTE

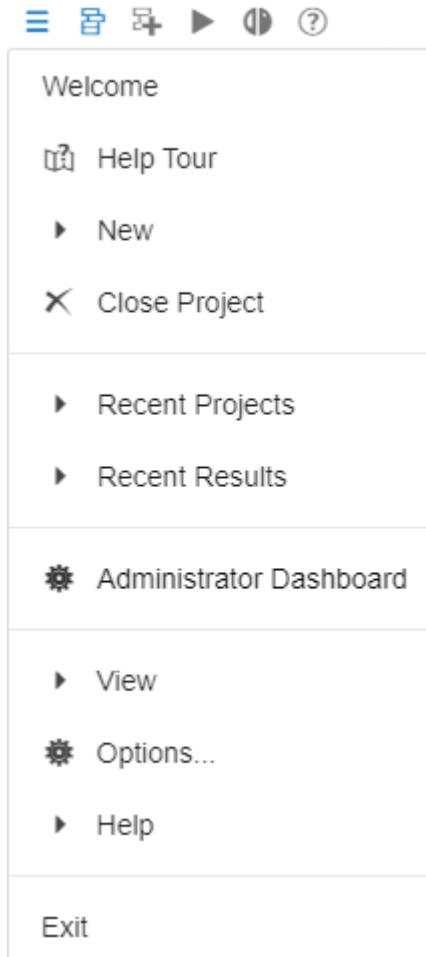
- The interactive sudo elevation requires that the VTune Profiler Agent is deployed under an account that has sudo privileges. To achieve that, ensure that the **Username** that you provide during deployment belongs to an account with sudo privileges.
 - VTune Profiler provides the password directly to the target system and does not store the password.
-

Control VTune Profiler Agents

The **Administrator Dashboard** feature of VTune Profiler Server enables you to monitor and manage one or multiple agents from a single point.

To open the Administrator Dashboard:

1. Open the VTune Profiler Server interface in your browser.
2. In the main toolbar, open the drop-down menu and select **Administrator Dashboard**.



The dashboard opens in a new tab and shows all agents that are related to this instance of VTune Profiler Server. This includes both connected and disconnected agents.

Dashboard

Status: All ▾

IP	Hostname	Username	Status
127.0.0.1		Admin	<input type="checkbox"/>
10.125.21.62		Admin	<input type="checkbox"/>

The dashboard enables you to:

- View information related to this agent:
 - Target system IP address and hostname
 - The username of the agent's user.
 - Current connection status.
- Admit or stop one or multiple agents. To admit or stop multiple agents, select the agents by ticking the checkboxes and click **Admit selected** or **Stop selected**.

See Also

[Install VTune Profiler Server](#) Set up Intel® VTune™ Profiler as a web server, using a lightweight deployment intended for personal use or a full-scale corporate deployment supporting multi-user environment.

[Cookbook: Using VTune Profiler Server in HPC Clusters](#)

Microsoft Visual Studio* Integration

You can simplify the process of debugging code and tuning your application when both your application and tuning tools are available in the same interface.

Intel® VTune™ Profiler integrates into Microsoft Visual Studio environment and enables you to create and tune your application within a single environment.*

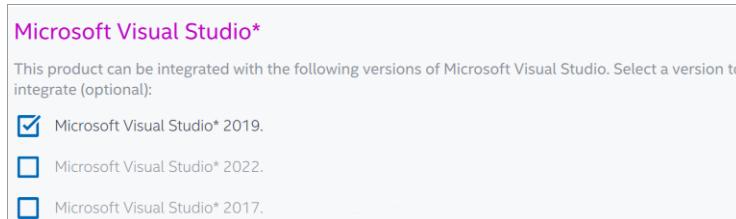
Explore details on:

- [Integrate VTune Profiler into Visual Studio during installation](#)
- [Integrate VTune Profiler into Visual Studio after installation](#)
- [Configuring VTune Profiler via Visual Studio Options pane](#)

NOTE Support for Visual Studio* 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Integrate VTune Profiler into Visual Studio During Installation

VTune Profiler integrates into Visual Studio by default. You specify the version of Visual Studio used for integration in the **IDE Integration** portion of the installation wizard. If you have several versions of Visual Studio and want to instruct the installation wizard to use a specific version for integration, click the **Customize** link and specify the required version. For example:



NOTE

You can only integrate one version of VTune Profiler into Visual Studio IDE.

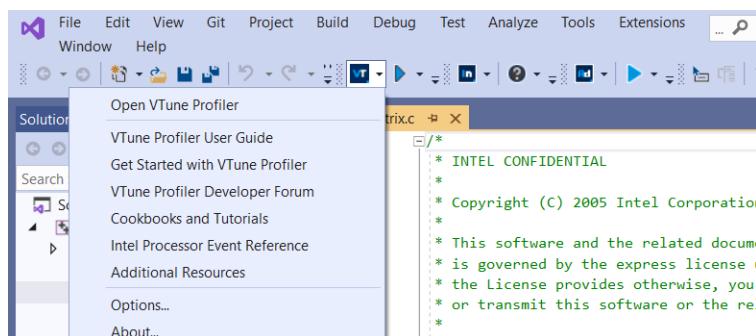
Integrate VTune Profiler into Visual Studio After Installation

If you have already installed VTune Profiler and need to integrate it into Visual Studio IDE, do the following:

1. Open the installation wizard from **Control Panel > Programs > Uninstall a program > Intel® VTune™ Profiler > Change**.
2. In the **Installed Products** window, select the **Modify > Add/Remove Components** option from the drop-down menu.
3. Click through to step 3: **Integrate IDE**. Select the required version of Visual Studio IDE.
4. Click the **Next** arrow button to complete the update.

Open VTune Profiler in Visual Studio IDE

Once you have integrated VTune Profiler in Visual Studio, open the IDE. The toolbar displays icons to start VTune Profiler and profile with it.



You can also access VTune Profiler from the **Tools** menu in the IDE.

Load a project in the Solution Explorer window. Once you have compiled it, you can profile with VTune Profiler. When you click the **Open VTune Profiler** icon from the toolbar, the application opens to the [Welcome Page](#).

The graphical interface of VTune Profiler integrated into Visual Studio is similar to the [standalone VTune Profiler interface](#).

Configure VTune Profiler for Visual Studio

To configure VTune Profiler options in the Visual Studio IDE, click the pull-down menu next to the **Open VTune Profiler** icon (



) and select **Options...**:

- Use the [General pane](#) to configure general collection options such as application output destination, management of the collected raw data, and so on.
- Use the [Result Location pane](#) to specify the result name template that defines the name of the result file and its directory.
- Use the [Source/Assembly pane](#) to manage the source file cache and specify syntax for the disassembled code.
- Use the [Privacy pane](#) to opt in/out of collecting your information for the Intel® Software Improvement Program.

If you need to change environment settings, however, read the documentation provided for the Visual Studio product.

From the standalone interface, you can access VTune Profiler options via the **File > Options...** menu.

NOTE VTune Profiler does not support the use of `CMakePresets.json` in Visual Studio.

Supported Visual Studio Projects

VTune Profiler supports the following Visual Studio project types:

- public const string FortranProjectType = "{60B2DF28-7A97-4DB5-AD4A-C0A6CFA6A9EC}";
- public const string CSProjectType = "{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}";
- public const string VBProjectType = "{F184B08F-C81C-45F6-A57F-5ABD9991F28F}";
- public const string ExeProjectType = "{911E67C6-3D85-4FCE-B560-20A9C3E3FF48}";
- public const string CPPProjectType = "{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}";
- public const string ICPPProjectType = "{EAF909A5-FA59-4C3D-9431-0FCC20D5BCF9}";
- public const string PythonProjectType = "{888888A0-9F3D-457C-B088-3A5042F75D52}";
- public const string FSProjectType = "{F2A71F9B-5D33-465A-A702-920D77279786}";

[VTune Profiler Release Notes](#)

[VTune Profiler System Requirements](#)

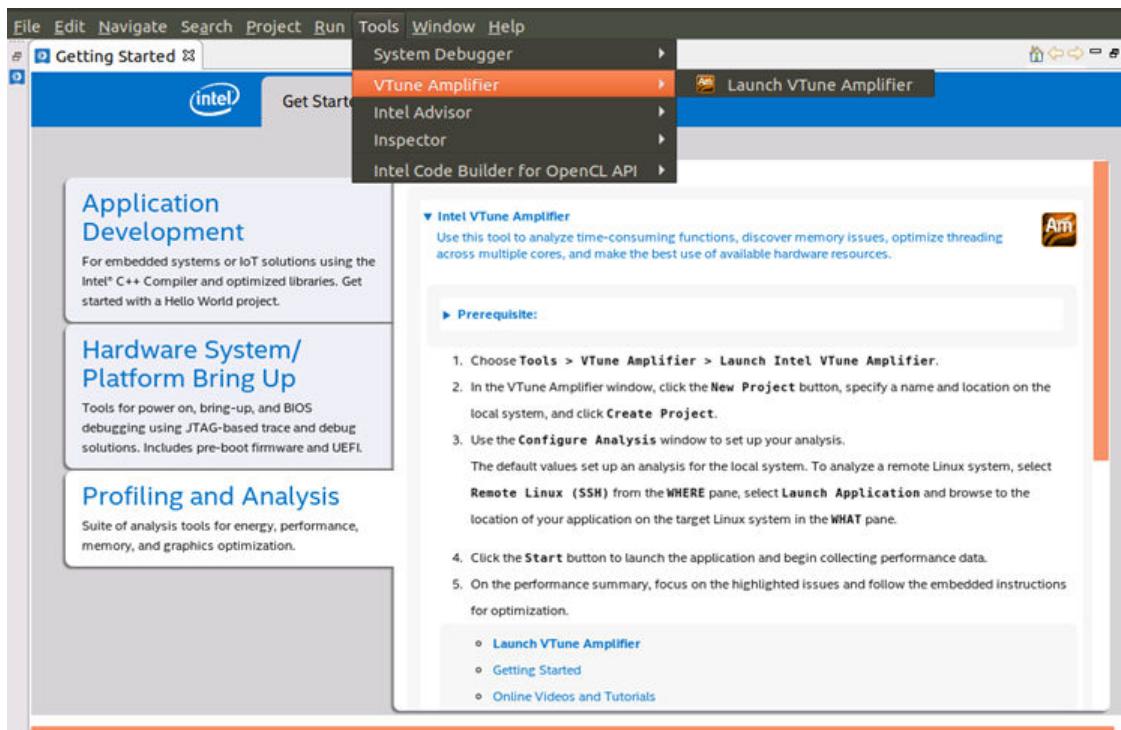
Eclipse* and Intel System Studio IDE Integration

After Intel® System Studio installation, Intel® VTune™ Profiler is integrated into the Eclipse* IDE. As a result, you get access to the VTune Profiler standalone interface.

Tip

When you launch VTune Profiler directly from Intel System Studio, you do not need to set environment variables on your system because they are set during the launch process.

To open the VTune Profiler from Intel System Studio, select the **Tools > VTune Profiler > Launch VTune Profiler** menu option.



See Also

Analyze Performance

[Intel® VTune™ Profiler Graphical User Interface](#)

Containerization Support

Use containers to set up environments for profiling:

- You can prepare a container with an environment pre-configured with all the tools you need, then develop within that environment.
- You can move that environment to another machine without additional setup.
- You can extend containers with different sets of compilers, profilers, libraries, or other components, as needed.

Depending on the setup, Intel® VTune™ Profiler supports the following target types and analyses:

Setup	Target Type	Analysis Type
VTune Profiler and app running in the same container	<ul style="list-style-type: none"> • Launch Application • Attach to Process • Profile System (not supported for Java* targets) 	<ul style="list-style-type: none"> • User-Mode Sampling Hotspots • Hardware Event-Based Sampling Hotspots • Microarchitecture Exploration
VTune Profiler in the container and an app outside the container	<ul style="list-style-type: none"> • Attach to Process • Profile System (not supported for Java targets) 	
VTune Profiler outside the container and an app in the	<ul style="list-style-type: none"> • Attach to Process • Profile System 	

Setup	Target Type	Analysis Type
container (supported containers: LXC*, Docker, Mesos*, Singularity*)		

NOTE

- The Hotspots (hardware event-based sampling mode) and Microarchitecture Exploration analyses are configured to use driver-less data collection based on the Linux Perf* tool.
- In the **Profile System** mode, VTune Profiler profiles *all* applications running in the same container or in different containers simultaneously. So, the standard [limitation for the system-wide profiling of the managed code](#) is not applicable to Java applications running in the containers.
- The Attach to Process target type for Java apps is supported only with the Java Development Kit (JDK).
- When VTune Profiler and an application are NOT running in the same container, both local and remote target system configurations are available.

See Also

[Profile Container Targets from the Host](#)

[Run VTune Profiler in a Container](#)

[Cookbook: Profiling in a Docker* Container](#)

[Cookbook: Profiling in a Singularity* Container](#)

Run VTune Profiler in a Container

Install a Docker image with Intel® VTune™ Profiler and profile native or Java* applications running inside the same container or outside the container.*

Prerequisites

- Configure a Docker image:

1. Create and configure a Docker image.

For the pre-installed Intel® oneAPI Base Toolkit including VTune Profiler, you may pull an existing Docker image from the Docker Hub repository:

```
host> image=amr-registry.caas.intel.com/oneapi/oneapi:base-dev-ubuntu20.04
host> docker pull "$image"
```

2. To enable profiling from the container and have all host processes visible from the container, run your Docker image with --pid=host as follows:

```
host> docker run --pid=host --cap-add=SYS_ADMIN --cap-add=SYS_PTRACE -it "$image"
```

where the SYS_ADMIN value adds a capability to run hardware event-based sampling analysis; the SYS_PTRACE value enables user-mode sampling analysis.

- To profile a target application running in the same container where VTune Profiler is installed, do the following:

1. Copy your application to the running Docker container. For example:

```
host> docker cp /home/samples/matrix.tar 98fec14f0c08:/var/local
```

where 98fec14f0c08 is your container ID.

- 2.** Compile your target in the container, if required.

Install and Run VTune Profiler in a Container

NOTE

These steps are NOT required if you use a Docker image with pre-installed Intel oneAPI Base Toolkit.

1. Install the command-line interface of VTune Profiler inside your Docker container.

Make sure to select the **[2] Custom installation > [3] Change components to install** and de-select components that are not required in the container environment: **[3] Graphical user interface** and **[4] Platform Profiler**.

2. After installation, set up environment variables for the VTune Profiler. For example, for VTune Profiler in Intel oneAPI Base Toolkit:

```
container> source /opt/intel/oneapi/vtune/version/env/vars.sh
```

Run Analysis for Your Container Target

Set up your analysis for a target running in the container, using the following supported target and analysis types:

Target Type	Analysis Type
<ul style="list-style-type: none"> • Launch Application • Attach to Process • Profile System (not supported for Java* targets) 	<ul style="list-style-type: none"> • User-Mode Sampling Hotspots • Hardware Event-Based Sampling Hotspots • Microarchitecture Exploration

To run an analysis, enter:

```
vtune -collect <analysis_type> [options] -- [container_target]
```

For example:

```
container> vtune -collect hotspots -knob sampling-mode=hw -- /home/samples/matrix
```

Run Analysis for Your Host Target

Set up your analysis for a target running on the host, using the following supported target and analysis types:

Target Type	Analysis Type
<ul style="list-style-type: none"> • Attach to Process • Profile System (not supported for Java targets) 	<ul style="list-style-type: none"> • User-Mode Sampling Hotspots • Hardware Event-Based Sampling Hotspots • Microarchitecture Exploration

To run an analysis, enter:

```
vtune -collect <analysis_type> [options] -- [host_target]
```

For example:

```
container> vtune -collect hotspots -target-process java
```

Known Issues:

1. **Issue:** Function-level analysis is not available by default. VTune Profiler maps the samples to the binaries from user target app but it cannot resolve the functions because the binaries from the host are not available from the container.
Solution: Run the Docker container with the mounted host folder containing the binaries and specify a [search directory](#) as an argument to the `vtune` command.
2. **Issue:** VTune Profiler is run in the container by the root user while the app on the host is run by a non-root user. As a result, User-Mode Sampling Hotspots analysis fails to run with an error "*Both target and VTune Profiler should be run by the same user*".
Solution: Make sure the same user runs VTune Profiler in the container and the target app outside the container.

See Also

- [Cookbook: Profiling in a Docker* Container](#)
[Cookbook: Profiling in a Singularity* Container](#)
[Installation Guide for VTune Profiler on Linux*](#)
[Run Command Line Analysis](#)

Profile Container Targets from the Host

Launch Intel® VTune™ Profiler from the host and profile native or Java applications running in an LXC*, Docker, Mesos*, or Singularity* container on a Linux system.*

Prerequisites

VTune Profiler automatically detects an application running in the container. No container configuration specific for performance analysis is required. But to run user-mode sampling analysis types (Hotspots or Threading), make sure to run the container with the ptrace support enabled:

```
host> docker run --cap-add=SYS_PTRACE -td myimage
```

or launch the container in the privileged mode:

```
host> docker run --privileged -td myimage
```

Configure and Run an Analysis for a Container Target

Set up your analysis for a target running in the container, using the following supported target and analysis types:

Target Type	Analysis Type
<ul style="list-style-type: none"> Attach to Process Profile System 	<ul style="list-style-type: none"> User-Mode Sampling Hotspots Hardware Event-Based Sampling Hotspots Microarchitecture Exploration

1. [Create a VTune Profiler project](#) on the host system.
2. From the **WHERE** pane of the **Configure Analysis** window, select the **Local Host** system to start analysis from your host Linux system or **Remote Linux (SSH)** to start analysis from a remote Linux system connected to your host system via SSH. For the remote Linux targets, make sure to [configure SSH connection](#).
3. From the **WHAT** section, specify your analysis target. For container target analysis, the following target types are supported: **Attach to Process** and **Profile System**.

Configure your process or system target as usual using available configuration options.

NOTE

In the **Profile System** mode, VTune Profiler profiles *all* applications running in the same container or in different containers simultaneously. So, the standard [limitation for the system-wide profiling of the managed code](#) is not applicable to Java applications running in the containers.

You can attach the VTune Profiler running under the superuser account to a Java process or a C/C++ application with embedded JVM instance running under a low-privileged user account. For example, you may attach the VTune Profiler to Java based daemons or services.

NOTE

The dynamic attach mechanism is supported only with the Java Development Kit (JDK).

4. From the **HOW** section, select an analysis and customize the analysis options, if required.

NOTE

The [Hotspots](#) (hardware event-based sampling mode) and [Microarchitecture Exploration](#) analyses are configured to use [driverless data collection](#) based on the Linux Perf* tool to gather performance data for targets running in a container.

5. Click **Start** to launch the analysis.

Alternatively, you may configure and run any of these analyses using the VTune Profiler command line interface ([vtune](#)). For example, to run a system-wide Hotspots analysis locally, enter:

```
host> vtune -collect hotspots -knob sampling-mode=hw -analyze-system -d 60
```

To run Hotspots analysis in the Attach to Process mode on a remote system, enter:

```
host> vtune -target-system=ssh:user1@172.16.254.1 -collect hotspots -knob sampling-mode=hw -target-process=java -d 60
```

View Data

The collected result opens in the default Hotspots viewpoint, where paths to container modules show up with prefixes (for instance, `docker` or `lxc`):

Function / Call Stack

Function / Call Stack	Module Path
<code>:multiply1</code>	[Compiled Java code]
<code>:random</code>	[Compiled Java code]
<code>:atomic::AtomicLong::comp</code>	[Compiled Java code]
<code>ext</code>	[Compiled Java code]
<code>:transpone</code>	[Compiled Java code]
<code>:matrix</code>	[Compiled Java code]
	[vdso]
	<code>docker:bc2707759efe257a3a17ecd473e047049e0256664a0f1c1d27f891e9e1f7ed63:/root/openjdk_8u232-b09-01</code>
	<code>docker:bc2707759efe257a3a17ecd473e047049e0256664a0f1c1d27f891e9e1f7ed63:/lib/x86_64-linux-gnu/vmlinux</code>
	<code>docker:bc2707759efe257a3a17ecd473e047049e0256664a0f1c1d27f891e9e1f7ed63:/lib/x86_64-linux-gnu/libc.so.6</code>

See Also

[Cookbook: Profiling in a Docker* Container](#)
[Cookbook: Profiling in a Singularity* Container](#)
[Java* Code Analysis](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

[Analysis Target Options](#)

Set Up Project

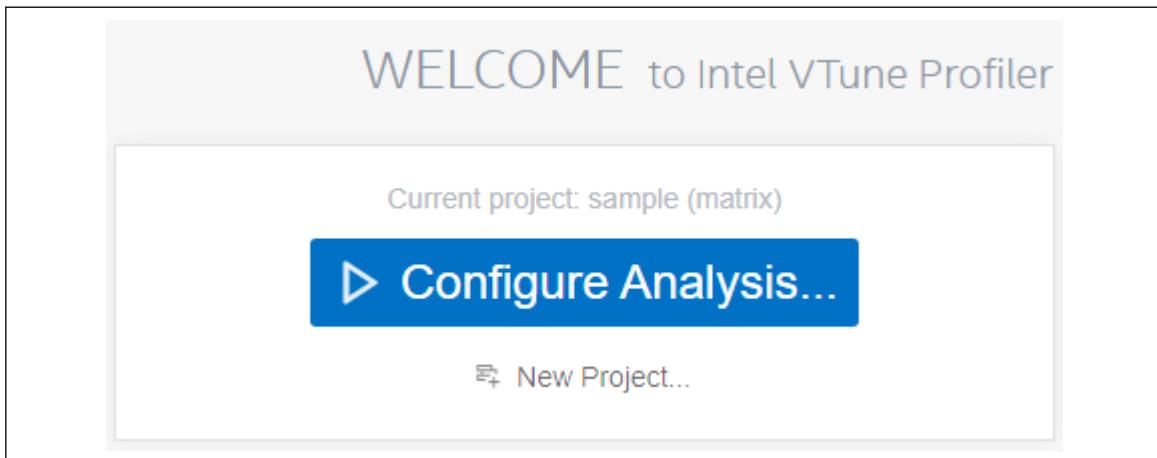
Before you run an analysis with Intel® VTune™ Profiler, you must first create a project. This is a container for an analysis target, analysis type configuration, and data collection results. You use the VTune Profiler user interface to create a project. You cannot create one from the command line.

For Microsoft Visual Studio* IDE, VTune Profiler creates a project for an active startup project, inherits Visual Studio settings and uses the application generated for the selected project as your analysis target. The default project directory is My VTune Results-[project name] in the solution directory.

For the standalone graphical interface, create a project by specifying its name and path to an analysis target. The default project directory is %USERPROFILE%\My Documents\Amplifier XE\Projects on Windows* and \$HOME/intel/vtune/projects on Linux*.

To create a VTune Profiler project for the standalone GUI:

1. Click **New Project...** in the **Welcome** screen.



2. In the **Create a Project** dialog box, configure these settings:

Use This	To Do This
Project Name field	Enter the name of a new project.
Location field and Browse button	Choose or create a directory to contain the project. Tip Store all your project directories in the same location.
Create Project button	Create a container *.vtuneproj file and open the Configure Analysis window.

3. Click the **Create Project** button.

The **Configure Analysis** window opens.

Your default project is pre-configured for the [Performance Snapshot](#) analysis. This presents an overview of issues that affect the performance of your application. Click the



Start button to proceed with the default setup.

To select a different analysis type, click on the name of the analysis in the analysis header section. This opens an Analysis Tree with all available analysis types.

NOTE

You cannot run a performance analysis or [import analysis data](#) without creating a project.

See Also

[WHERE: Analysis System](#)

[WHAT: Analysis Target](#)

[HOW: Analysis Types](#)

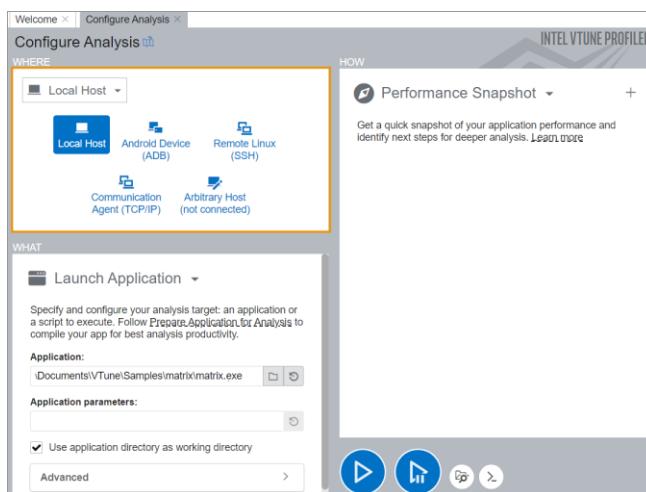
[VTune Profiler Filenames and Locations](#)

WHERE: Analysis System

Before running a performance analysis, make sure to prepare your target system, which is a system where a profiling session runs.

The target system can be the same as the *host system*, which is a system where you have installed VTune Profiler. If you run an analysis on the same system where you installed VTune Profiler (i.e. target system=host system), the target system is called a *local* system. Target systems other than local ones are called *remote systems*.

When you create a project, the **Configure Analysis** window opens pre-configured to run **Performance Snapshot** on the local host. Click on the analysis name in the **WHERE** pane to open the Analysis Tree, where you can choose a different analysis type.



Use these options to decide where you want to run the analysis.

Option	Description
Local Host	Run an analysis on the local host system. NOTE This type of the target system is not available for macOS*.
Remote Linux (SSH)	Run an analysis on a remote regular or embedded Linux* system. VTune Profiler uses the SSH protocol to connect to your remote system.
Android Device (ADB)	Run an analysis on an Android device. VTune Profiler uses the Android Debug Bridge* (adb) to connect to your Android device.
Communication Agent (TCP/IP)	Profile an embedded system running a real-time operating system using the Analysis Communication Agent.
Arbitrary Host (not connected)	Create a command line configuration for a platform NOT accessible from the current host, which is called an <i>arbitrary target</i> .

Explore system-specific requirements for analysis targets:

- [Windows* targets](#)

- [Linux* targets](#)
- [Embedded Linux targets](#) (Wind River*, Yocto*)
- [Android* targets](#)
- [Embedded system targets](#)

See Also

[Analysis System Options](#)

[Configure SSH Access for Remote Collection](#)

[Window: Configure Analysis](#)

Analysis System Options

*Specify a system targeted for performance analysis in the **Configure Analysis** window.*

Prerequisites: Make sure to prepare your [target system](#) for analysis.

To access the system configuration options:

1. Open the **Configure Analysis** window.
2. Choose a target system in the **WHERE** pane.

If you select the **Local Host** option, no system specific configuration is required. Other systems types need additional configuration.

Remote Linux* Options

When you select the **Remote Linux (SSH)** system on the WHERE pane, the VTune Profiler provides the following configuration options:

Use This	To Do This
SSH destination field	Specify a username, hostname, and port (if required) for your remote Linux machine as <code>username@hostname[:port]</code> . Make sure an SSH password-less connection is established in advance.
VTune Profiler installation directory on the remote system field	Specify a path to the VTune Profiler on the remote system. <ul style="list-style-type: none"> • If VTune Profiler is not installed on the remote system, the collectors are automatically copied over, installed in the default location (<code>/tmp</code>), and the path is supplied. • If VTune Profiler is already installed in a location other than <code>/tmp</code>, add the location here.
Temporary directory on the remote system field	Specify a path to the <code>/tmp</code> directory on the remote system where performance results are temporarily stored.
Deploy button	Deploy the collector package to the target system if the package is not found on the target system.

Android* Options

When you select the **Android Device (ADB)** system on the WHERE pane, the VTune Profiler displays the **ADB destination** menu and prompts you to specify an Android device for analysis. When the ADB connection is set up, the VTune Profiler automatically detects available devices and displays them in the menu.

Arbitrary Host Options

When you select the **Arbitrary Host (not connected)** system on the **WHERE** pane, the VTune Profiler prompts you to specify the following data for the system targeted for the analysis but currently not accessible:

Use This	To Do This
Hardware platform field	Select a hardware platform for analysis from the drop-down menu, for example: Intel® processor code named Anniedale.
Operating system field	Specify either Windows* or GNU*/Linux* operating system.

What's Next

In the **WHAT** pane, select an [analysis target](#) for the specified analysis system.

NOTE

You can launch an analysis only for targets accessible from the current host. For an *arbitrary* target, you can only [generate a command line configuration](#), save it to the buffer and later launch it on the intended host.

See Also

[Set Up Android* System](#)

[Prepare an Android* Application for Analysis](#)

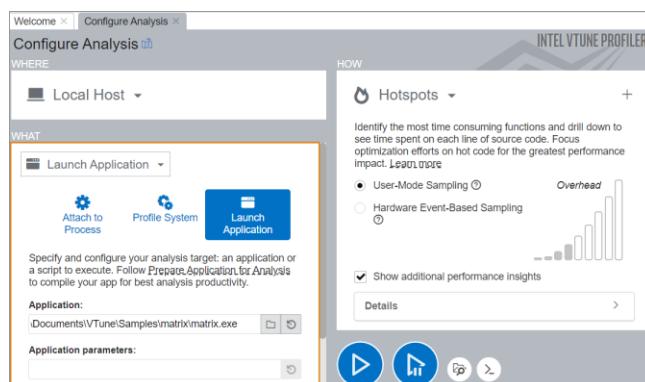
[Set Up Linux* System for Remote Analysis](#)

[Arbitrary Targets](#)

WHAT: Analysis Target

A target is an executable file you analyze using the Intel® VTune™ Profiler, which could be an executable file, a process, or a whole system.

By default, when you create a new project, the VTune Profiler opens an analysis configuration with the **Launch Application** analysis target pre-selected:



To change a target type for your project, click the



Browse button on the **WHAT** pane. Select from these target types:

Launch Application	Enable the Launch Application pane and choose and configure an application to analyze, which can be either a binary file or a script. See options for launching an application .
<hr/>	
Attach to Process	NOTE This target type is not supported for the Hotspots analysis of Android applications. Use the Attach to Process or Launch Android Package types instead.
Profile System	Enable the Profile System pane and configure the system-wide analysis that monitors all the software executing on your system.
Launch Android Package	Enable the Launch Android Package pane to specify the name of the Android* package to analyze and configure target options. See options for launching an Android package .

Options available for the target configuration depend on the target system you select in the **WHERE** pane.

To focus on analyzing particular processes, you may collect data on all processes (without selecting the Attach to Process target type) and then filter the collected results as follows:

1. From the **Grouping** drop-down menu in the **Bottom-up** window, select the grouping by Process, for example: **Process/Function/Thread/Call Stack**.
 2. In the grid, right-click the process you are interested in and select the **Filter In by Selection** option from the context menu.
- VTune Profiler updates the grid to provide data for the selected process only.
3. From the **Grouping** drop-down menu, select any other grouping level you need, for example: **Function/Call Stack**.

VTune Profiler groups the data for the selected process according to the granularity you specified.

NOTE

If attaching to a running process causes a hang or crash, consider launching your application with the VTune Profiler in a [paused state](#), and resume the collection when the application gets to an area of interest.

See Also

[WHERE: Analysis System](#)

[Analysis Target Options](#)

[Analysis System Options](#)

[HOW: Analysis Types](#)

Analysis Target Options

Manage the analysis of your target using target specific configuration options provided in the **Configure Analysis** window.

To access target configuration options:

1. Open the **Configure Analysis** window.
2. Choose a **target system** on the **WHERE** pane.
3. Choose a target type on the **WHAT** pane and configure the options below.

NOTE

To create a command line configuration for a target not accessible from the current host, choose the **Arbitrary Host** target system on the **WHERE** pane. Make sure to choose an operating system your target will be running with: **Windows** or **GNU/Linux** and a hardware platform.

Target options vary with the selected target system and target type (**Launch Application**, **Attach to Process**, or **Profile System**).

Basic Options

Use This	To Do This
Inherit settings from Visual Studio* project check box (supported for Visual Studio IDE only)	Enable/disable using the project currently opened in Visual Studio IDE and its current configuration settings as a target configuration. Checking this check box makes all other target configuration settings unavailable for editing.
Inherit system environment variables check box	Inherit and merge system and user-defined environment variables. Otherwise, only the user-defined variables are set.

Launch Application options:

Application field	Specify a full path to the application to analyze, which can be a binary file or script.
Application parameters field	Specify input parameters for your application.
Use application directory as working directory check box	Automatically match your working and application directory (enabled by default). An <i>application directory</i> is the directory where your application resides. For example, for a Linux application /home/foo/bar the application directory is /home/foo. Application and working directories may be different if, for example, an application file is located in one directory but should be launched from a different directory (<i>working directory</i>).
Working directory field	Specify a directory to use for launching your analysis target. By default, this directory coincides with the application directory.

Attach to Process options:

Process name field	Identify the executable to analyze by its name.
PID field	Identify the executable to analyze by its process ID (PID).

Attach to Process options:

Click the **Select** button to see a list of currently available processes to attach to. As soon as you select a process of interest, the VTune Profiler automatically populates the **Process name** fields with the data for the selected process.

Arbitrary Host options:

- | | |
|-----------------------------------|--|
| Use MPI launcher check box | Enable the check box to generate a command line configuration for MPI analysis. Configure the following MPI analysis options: <ul style="list-style-type: none"> • Select MPI launcher: Select an MPI launcher that should be used for your analysis. You can either enable the Intel MPI launcher option (default) or select Other and specify a launcher of your choice. • Number of ranks: Specify the number of ranks used for your application. • Profile ranks: Use All to profile all ranks, or choose Selective and specify particular ranks to profile, for example: 2-4,6-7,8. • Result location: Specify a relative or absolute path to the directory where the analysis result should be stored. |
|-----------------------------------|--|

Advanced Options

Use the **Advanced** section to provide more details on your target configuration.

Use This	To Do This
User-defined environment variables field	Type or paste environment variables required for running your application.
Managed code profiling mode menu	Select a profiling mode for managed code. Managed mode attributes data to managed source and only collects managed portion. Native mode collects everything but does not attribute data to managed source. Mixed mode collects everything and attributes data to managed source where appropriate.
Automatically resume collection after (sec)	Specify the time that should elapse before the data collection is resumed. When this option is used, the collection starts in the paused mode automatically.
Automatically stop collection after (sec)	Set the duration of data collection in seconds starting from the target run. This is useful if you want to exclude some post-processing activities from the analysis results.
Analyze child processes check box	Collect data on processes launched by the target process. Use this option when profiling an application with the script. Selecting this option enables the Per-process Configuration where you can specify child processes to analyze. For example, if your target application calls shell or makes processes, you can choose to exclude them from analysis and focus only on the processes you develop.

Use This	To Do This
	<p>The Default process configuration represents how all processes should be analyzed. This line cannot be removed, but can be customized. Depending on your choice, you may include/exclude from the data collection specific processes (self value) and the child processes they spawn (children value).</p> <p>This option is not applicable to hardware event-based analysis types.</p>
Duration time estimate menu	<p>NOTE</p> <p>This option is deprecated. Use the CPU sampling interval option on the HOW configuration pane instead.</p>
	<p>Estimate the application duration time. This value affects the size of collected data. For long running targets, sampling interval is increased to reduce the result size. For hardware event-based sampling analysis types, the VTune Profiler uses this estimate to apply a multiplier to the configured sample after value.</p>
Allow multiple runs check box	<p>Enable multiple runs to achieve more precise results for hardware event-based collections. When disabled, the collector multiplexes events running a single collection, which lowers result precision.</p>
Analyze system-wide check box	<p>Enable analyzing all processes running on the system. When disabled, only the target process is analyzed.</p> <p>This option is applicable to hardware event-based sampling analysis types only.</p>
Limit collected data by section	<p>If the amount of raw collected data is very large and takes long to process, use any of the following options to limit the collected data size:</p> <ul style="list-style-type: none"> • Result size from collection start, MB: Set the maximum possible result size (in MB) to collect. VTune Profiler will start collecting data from the beginning of the target execution and suspend data collection when the specified limit for the result size is reached. For unlimited data size, specify 0. • Time from collection end, sec: Set the timer enabling the analysis only for the last seconds before the target run or collection is terminated. For example, if you specified 2 seconds as a time limit, the VTune Profiler starts the data collection from the very beginning but saves the collected data only for the last 2 seconds before you terminate the collection.
	<p>NOTE</p> <p>The size of data stored in the result directory may not exactly match the specified result size due to the following reasons:</p> <ul style="list-style-type: none"> • The collected data may slightly exceed the limit since the VTune Profiler only checks the data size periodically. • During finalization, the VTune Profiler loads the raw data into a database with additional information about source and binary files.

Use This	To Do This
CPU mask field	Specify CPU(s) to collect data on (for example: 2-8,10,12-14). This option is applicable to hardware event-based analysis types only.
Custom collector field	Provide a command line for launching an external collection tool , if any. You can later import the custom collection data (time intervals and counters) in a CSV format to a VTune Profiler result.
Select finalization mode section	Finalization may take significant system resources. For a powerful target system, select Full mode to apply immediately after collection. Otherwise, shorten finalization with selecting the fast mode (default) or defer it to run on another system (compute checksums only).
Wrapper script field	<p>Provide a script that is launched on the target system before starting the collection. On the host system, you can prepare a custom script that prepares the target environment and calls the VTune Profiler collector in this environment.</p> <p>An example of the wrapper script:</p> <pre data-bbox="535 819 1122 1115">#!/bin/bash # Prefix script echo "Target process PID: \$VTUNE_TARGET_PID" # Run VTune collector "\${@}" # Postfix script ls -la \$VTUNE_RESULT_DIR</pre>
	You can use the script to perform any actions available through the CLI of your target operating system, and use "\${@}" or "\$*" to pass all arguments into the script and start VTune Profiler collection in this environment.
	The following environment variables are available from the script:
	<pre data-bbox="535 1290 964 1495">VTUNE_TARGET_PID VTUNE_TARGET_PROC_NAME VTUNE_RESULT_DIR VTUNE_TEMP_DIR VTUNE_TARGET_PACKAGE_DIR VTUNE_DATA_DIR VTUNE_USER_DATA_DIR</pre>
NOTE	<ul style="list-style-type: none"> VTune Profiler preserves the content of the script. The script is preserved within the project and is run for every analysis within that project. To apply any changes to the script, attach it again using the same Wrapper script field. For Linux targets, make sure that the script file is saved with LF line endings.
Result location options	Select where you want to store your result file. By default, the result is stored in the project directory.

Use This	To Do This
Trace MPI check box (Linux* targets only)	Configure collectors to trace MPI code and determine MPI rank IDs in case of a non-Intel MPI library implementation.
Analyze KVM guest OS check box (Linux targets only)	<p>Enable KVM guest system profiling. For proper kernel symbol resolution, make sure to specify:</p> <ul style="list-style-type: none"> • a local path to the <code>/proc/kallsyms</code> file copied from the guest OS • a local path to the <code>/proc/modules</code> file copied from the guest OS
Arbitrary Host options:	
Select a system for result finalization options	The result can be finalized on the same target system where the analysis is run (default). In this case make sure your target system is powerful enough for finalization. If you choose to finalize the result on another system, VTune Profiler will only compute module checksums to avoid an ambiguity in resolving binaries on a different system.

Support Limitations

- VTune Profiler provides limited support for profiling Windows* services. For details, see [Profiling Windows Services](#) article on the web.
- System-wide profiling is not supported for the user-mode sampling and tracing collection.
- For driverless event-based sampling data collection, VTune Profiler supports local and remote Launch Application, Attach to Process and Profile System target types but their support fully depends on the Linux Perf profiling credentials specified in the `/proc/sys/kernel/perf_event_paranoid` file and managed by the administrator of your system using root credentials. For more information, see the *perf_event related configuration files* topic at http://man7.org/linux/man-pages/man2/perf_event_open.2.html. By default, only user processes profiling at the both user and kernel spaces is permitted, so you need granting wider profiling credentials via the `perf_event_paranoid` file to employ the Profile System target type.

What's Next

In the **HOW** pane, select an [analysis type](#) applicable to the specified target type and click **Start** to run the analysis.

NOTE

You can launch an analysis only for targets accessible from the current host. For an *arbitrary* target, you can only generate a command line configuration, save it to the buffer and later launch it on the intended host.

See Also

[Arbitrary Targets](#)

[Managed Code Targets](#)

[Limit Data Collection](#)

[Allow Multiple Runs or Multiplex Events](#)

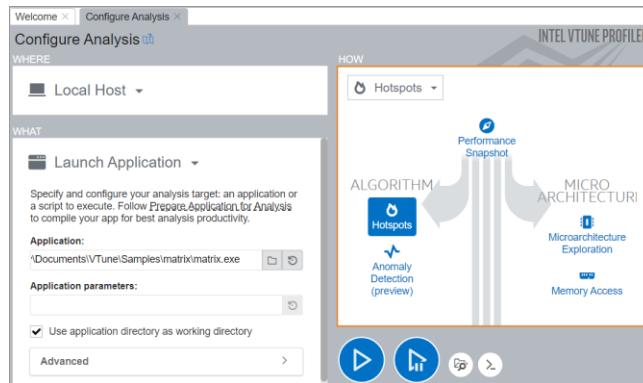
[Import External Data](#)

Generate Command Line Configuration from GUI

HOW: Analysis Types

Intel® VTune™ Profiler provides a set of pre-configured analysis types you may start with to address your particular performance optimization goals.

When you create a project, VTune Profiler opens the **Configure Analysis** window that prompts you to specify **WHAT** you want to analyze (an application, process, or a whole system), a system **WHERE** you plan to run the analysis, and select **HOW** you need to run the analysis.



Click the header in the **HOW** pane to open an analysis tree. Select from an analysis type from one of these groups:

Performance Snapshot analysis:

- Use [Performance Snapshot](#) to get an overview of issues that affect the performance of an application on your system. The analysis is a good starting point that recommends areas for deeper focus. You also get guidance on other analysis types to consider running next.

Algorithm analysis:

- Use the [Hotspots](#) analysis type to investigate call paths and find where your code is spending the most time. Identify opportunities to tune your algorithms. See *Finding Hotspots tutorial: Linux | Windows*.
- Use [Anomaly Detection](#) (preview) to identify performance anomalies in frequently recurring intervals of code like loop iterations. Perform fine-grained analysis at the microsecond level.
- [Memory Consumption](#) is best for analyzing memory consumption by your app, its distinct memory objects, and their allocation stacks. This analysis is supported for Linux targets only.

Microarchitecture analysis:

- [Microarchitecture Exploration](#) (formerly known as General Exploration) is best for identifying the CPU pipeline stage (front-end, back-end, and so on) and hardware units responsible for your hardware bottlenecks.
- [Memory Access](#) is best for memory-bound apps to determine which level of the memory hierarchy is impacting your performance by reviewing CPU cache and main memory usage, including possible NUMA issues.

Parallelism analysis:

- [Threading](#) is best for visualizing thread parallelism on available cores, locating causes of low concurrency, and identifying serial bottlenecks in your code.
- Use [HPC Performance Characterization](#) to understand how your compute-intensive application is using the CPU, memory, and floating point unit (FPU) resources. See *Analyzing an OpenMP* and MPI Application tutorial: Linux*.

I/O analysis:

- [Input and Output](#) analysis monitors utilization of the IO subsystems, CPU and processor buses.

Accelerators analysis:

- [GPU Offload](#) (preview) is targeted for applications using a Graphics Processing Unit (GPU) for rendering, video processing, and computations. It helps you identify whether your application is CPU or GPU bound.
- [GPU Compute/Media Hotspots](#) (preview) is targeted for GPU-bound applications and helps analyze GPU kernel execution per code line and identify performance issues caused by memory latency or inefficient kernel algorithms.
- [CPU/FPGA Interaction](#) analysis explores FPGA utilization for each FPGA accelerator and identifies the most time-consuming FPGA computing tasks.

Platform analysis:

- [System Overview](#) is a driverless event-based sampling analysis that monitors a general behavior of your target system and identify platform-level factors that limit performance.

VTune Profiler-Platform Profiler:

Use [VTune Profiler-Platform Profiler](#) to get a holistic view of system behavior. You can then perform system characterization on a deployed system that runs a full load over an extended period of time.

With VTune Profiler-Platform Profiler, you can get insights into these aspects:

- Platform configuration
- Utilization
- Performance
- Imbalances related to compute, memory, storage, IO, and interconnects

NOTE

A **PREVIEW FEATURE** may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

Advanced users can create a [custom analysis](#) using the data collectors provided by VTune Profiler, or combining the collector of VTune Profiler with another [custom collector](#).

Search Directories

Search directories are used to locate supporting files and display analysis information in relation to your source code.

In some cases, the Intel® VTune™ Profiler cannot locate the supporting user files necessary for displaying analysis information and you may need to configure additional search locations or override standard ones. This is required for .exe projects on Windows* created out of Microsoft Visual Studio*, where no information about project directory structure is available, for C++ projects with a third party library for which you wish to define binaries/sources, or for the imported projects with the data collected remotely. When you run a remote data collection, the VTune Profiler copies binary files from the target system to the host by default. You need to either copy symbol and source files to the host or mount a directory with these files.

VTune Profiler searches the directories in the [particular order](#) when finalizing the collected data. For the VTune Profiler integrated into the Visual Studio IDE, the search directories are defined by the Microsoft Visual Studio C++ project properties.

For successful module resolution, the VTune Profiler needs to locate the following files:

- binaries (executables and dynamic libraries)
- symbols
- source files

It automatically locates the files for C/C++ projects that are not moved after building the application and collecting the performance data.

Configure Search Directories

To configure search directories:

1. Click the



Configure Analysis toolbar button.

The **Configure Analysis** window opens.

2. Click the



Search Sources/Binaries button at the bottom to open the corresponding dialog box and specify paths for symbol, binary and source files for the file resolution *on the host*.

3. To add a new search directory in the **Search Directories** table, click the <Add a new search location> row and type in the path and name of the directory in the activated text box, or click the browse button on the right to select a directory from the list. For example, if your project was initially located in /work/projects/my_project on Linux* and then was moved to /home/user/my_project_copy, you need to specify the /home/user/my_project_copy as a search directory for binary/symbol and source files.

NOTE

The search is non-recursive. Make sure to specify all required directories.

If the search directories were not configured properly and modules were not resolved, you may see the following:

- In the **Summary** window, you see a pop-up message starting with "*Data is not complete due to missing symbol information for user modules...*". This pop-up window provides shortcut options to specify search directories and re-resolve the analysis result.
- In the **Bottom-up** or **Top-down Tree** pane, the module shows only one [Unknown] line instead of meaningful lines with function names.
- When you double-click a row to view the related source code, you get a [Cannot find the source file window](#) asking you to locate the source file.

If the VTune Profiler cannot locate symbol files for system modules, it may provide incomplete stack information in the **Bottom-up/Top-down Tree** panes and **Call Stack** pane. In this case, you may see [Unknown frame(s)] hotspots when attributing system layers to user code using the **Call Stack Mode** option on the [filter toolbar](#). To avoid this for Windows targets, make sure to [configure the Microsoft symbol server](#) or set the _NT_SYMBOL_PATH environment variable. For Linux targets, [enable Linux kernel analysis](#).

See Also

[Dialog Box: Binary/Symbol Search](#)

[Dialog Box: Source Search](#)

[Problem: Unknown Frames](#)

[Finalization](#)

[Search Directories for Remote Linux* Targets](#)

[Search Directories for Android* Targets](#)

[Specify Search Directories from Command Line](#)
from command line

Search Order

When locating binary/symbol/source files, the Intel® VTune™ Profiler searches the following directories, in the following order:

1. Directory <result dir>/all (recursively).
2. Additional search directories that you defined for this project in the VTune Profiler **Binary/Symbol Search** dialog box.
3. For local collection, an absolute path.

For remote collection, the VTune Profiler searches its cache directory for modules copied from the remote system or tries to get the module from the remote system using the absolute path.

For results copied from a different machine, make sure to copy all the necessary source, symbol, and binary files required for result finalization.

- For binaries, the path is captured in the result data files.
- For symbol files, the path is referenced in the binary file.
- For source files, the path is referenced in the symbol file.

On Linux*, to locate the `vmlinu`x file, the VTune Profiler searches the following directories:

- `/usr/lib/debug/lib/modules/`uname -r`/vmlinu`
- `/boot/vmlinuz-`uname -r``

4. Search around the binary file.

Search the directory of the corresponding binary file.

On Windows*, search the directory of the corresponding binary file and alter the name of the symbol file holding the initial extension (for example, `app.dll` + `app_x86.pdb` -> `app.pdb`).

On Linux, search the `.debug` subdirectory of the corresponding binary file directory.

5. On Windows, Microsoft Visual Studio* search directories. All directories are considered as non-recursive. Directories may be specific to the selected build configuration and platform in time of collection.
6. System directories.

On Windows:

- Binary files: `%SYSTEMROOT%\system32\drivers` (non-recursively)
- Symbol files:
 - All directories specified in the `_NT_SYMBOL_PATH` environment variable (non-recursively). Symbol server paths are possible here as well as in step 2.
 - `srv*%SYSTEMROOT%\symbols` (treated as a symbol server path)
 - `%SYSTEMROOT%\symbols\dll` (non-recursively)

On Linux:

- Binary files: If the file to search is a bare name only (no full path, no extension), it is appended by the `.ko` extension before searching in the following directories:

```
/lib/modules (non-recursively)
/lib/modules/`uname -r`/kernel (recursively)
```

- Symbol files:

- `/usr/lib/debug` (non-recursively)
- `/usr/lib/debug` with appended path to the corresponding binary file (for example, `/usr/lib/debug/usr/bin/ls.debug`)
- Source files:
 - `/usr/src` (non-recursively)
 - `/usr/src/linux-headers-`uname -r`` (non-recursively)

If the VTune Profiler cannot find a file that is necessary for a certain operation, such as viewing source, it brings up a window enabling you to enter the location of the missing file.

NOTE

VTune Profiler automatically applies recursive search to the <result dir>/all directory and some system directories (Linux only). Additional directories you specify in the project configuration are searched non-recursively.

1. For **non-recursive directories**, the VTune Profiler searches paths by merging the parts of the file path with the specified directory iteratively. For example, for the /aaa/bbb/ccc/filename.ext file on Linux:

```
/specified/search/directory/aaa/bbb/ccc/filename.ext
/specified/search/directory/bbb/ccc/filename.ext
/specified/search/directory/ccc/filename.ext
/specified/search/directory/filename.ext
```

2. For **recursive directories**, the VTune Profiler searches the same paths as for the non-recursive directory and, in addition, paths in all sub-directories up to the deepest available level. For example:

```
/specified/search/directory/subdir1/filename.ext
/specified/search/directory/subdir1/sub...subdir1/filename.ext
...
/specified/search/directory/subdir1/sub...subdirN/filename.ext
...
/specified/search/directory/subdirN/filename.ext
```

3. For **symbol server paths** on Windows, symsrv.dll is used from product distributive. Custom symsrv.dll:s are not supported.

See Also

[Search Directories](#)

[Window: Cannot Find <file type> File](#)

[Dialog Box: Binary/Symbol Search](#)

[Dialog Box: Source Search](#)

Set Up Analysis Target

When you create a project for the Intel® VTune™ Profiler performance analysis, you have to specify what you want to profile - your analysis target, which could be an executable file, a process, or a whole system.

Supported Targets

Before starting an analysis, make sure your target and system are compiled/configured properly for performance profiling.

VTune Profiler supports analysis targets that you can run in these environments:

Development Environment Integration	<ul style="list-style-type: none"> Microsoft* Visual Studio* Eclipse*
Target Platform	<ul style="list-style-type: none"> Linux* OS Windows* OS Android* OS FreeBSD* QNX* Intel® Xeon Phi® processors (code name: Knights Landing)
Programming Language	<ul style="list-style-type: none"> C/C++ Fortran C# (Windows Store applications) Java* JavaScript Python* Go* .NET* .NET Core
Programming Model	<ul style="list-style-type: none"> Windows* API OpenMP* API Intel Cilk™ Plus OpenCL™ API Message Passing Interface (MPI) Intel Threading Building Blocks Intel Media SDK API
Virtual Environment	<ul style="list-style-type: none"> VMWare* Parallels* KVM* Hyper-V* Xen*
Containers	LXC*, Docker*, Mesos*

Specify Your Target

To specify your target for analysis:

- Click the



New Project button on the toolbar to create a new [project](#).

If you need to re-configure the target for an existing project, click the



Configure Analysis toolbar button.

The **Configure Analysis** window opens. By default, the project is pre-configured to run the [Performance Snapshot](#) analysis.

2. If you do not run an analysis on the local host, expand the **WHERE** pane and select an appropriate target system.

The target system can be the same as the *host system*, which is a system where the VTune Profiler GUI is installed. If you run an analysis on the same system where the VTune Profiler is installed (i.e. target system=host system), such a target system is called *local*. Target systems other than local are called *remote systems*. But both local and remote systems are *accessible* targets, which means you can access them either directly (local) or via a connection (for example, SSH connection to a remote target).

Local Host	Run an analysis on the local host system.
	<p>NOTE This type of the target system is not available for macOS*.</p>
Remote Linux (SSH)	Run an analysis on a remote regular or embedded Linux* system. VTune Profiler uses the SSH protocol to connect to your remote system. Make sure to fill in the SSH Destination field with the username, hostname, and port (if required) for your remote Linux target system as <code>username@hostname[:port]</code> .
Android Device (ADB)	Run an analysis on an Android device. VTune Profiler uses the Android Debug Bridge* (adb) to connect to your Android device. Make sure to specify an Android device targeted for analysis in the ADB Destination field. When the ADB connection is set up, the VTune Profiler automatically detects available devices and displays them in the menu.
Arbitrary Host (not connected)	Create a command line configuration for a platform NOT accessible from the current host, which is called an <i>arbitrary target</i> .

3. From the **WHAT** pane, specify an application to launch or click the



Browse button to select a different target type:

Launch Application (pre-selected)	Enable the Launch Application pane and choose and configure an application to analyze, which can be either a binary file or a script.
	<p>NOTE This target type is not supported for the Hotspots analysis of Android applications. Use the Attach to Process or Launch Android Package types instead.</p>
Attach to Process	Enable the Attach to Process pane and choose and configure a process to analyze.
Profile System	Enable the Profile System pane and configure the system-wide analysis that monitors all the software executing on your system.
Launch Android Package	Enable the Launch Android Package pane to specify the name of the Android* package to analyze and configure target options.

NOTE

- If you use VTune Profiler as a [web server](#), the list of available targets and target systems differs.
 - For [driverless event-based sampling data collection](#), VTune Profiler supports local and remote Launch Application, Attach to Process and Profile System target types but their support fully depends on the Linux Perf profiling credentials specified in the `/proc/sys/kernel/perf_event_paranoid` file and managed by the administrator of your system using root credentials. For more information see the *perf_event related configuration files* topic at http://man7.org/linux/man-pages/man2/perf_event_open.2.html. By default, only user processes profiling at the both user and kernel spaces is permitted, so you need granting wider profiling credentials via the `perf_event_paranoid` file to employ the Profile System target type.
-

What's Next

As soon as you specified the analysis system and target, you may either click the **Start** button to run [Performance Snapshot](#) or click the analysis name in the analysis header to choose a different analysis type.

See Also

[Analysis System Options](#)

[Analysis Target Options](#)

[WHAT: Analysis Target](#)

[HOW: Analysis Types](#)

[target-system](#)

[vtune option](#)

[Arbitrary Targets](#)

(not connected)

[Collect Data on Remote Linux* Systems from Command Line](#)

[Generate Command Line Configuration from GUI](#)

Prepare Application for Analysis

Follow this guidance to understand how to compile an application for analysis with Intel® VTune™ Profiler and make your analysis more productive.

Recommendations for All Compiled Languages

These guidelines apply to all supported operating system hosts and compiled languages. It is highly recommended that you follow this guidance to make your use of VTune Profiler as effective as possible.

• Do This:

Build your application in Release mode, with maximum appropriate compiler optimization level.

Because:

- This eliminates performance issues that can be resolved by compiler optimizations, enabling you to focus on bottlenecks that require your attention.

• Do This:

Generate debug information for your application, and, if possible, download debug information for any third-party libraries it uses.

Because:

- This enables source-level analysis: view problematic source lines right in VTune Profiler.
- This enables resolution of function names and proper call stack information.
- By default, most compilers/IDEs do not generate debug information in Release mode.

Prepare a C++ Application on Windows

To fulfill the [recommendations](#) on Windows, you will need these compiler flags:

```
/O2 /Zi /DEBUG
```

- The `/O2` flag enables compiler optimizations that favor speed.

NOTE The `/O2` flag is a recommendation to ensure you are profiling the Release version of your application with optimizations that favor speed enabled. If the production use of your application calls for a different optimization level, use your required level. The key idea is to profile your application when it is compiled as close to production use as possible.

- The `/Zi` and `/DEBUG` flags enable generation of debug info in the Program Database (PDB) format.

Follow these steps to configure the optimization level and debug information generation in Microsoft Visual Studio*:

1. Enable Release build configuration:

- On the Visual Studio toolbar, from the **Solution Configuration** drop-down list, select **Release**. This also enables the `/O2` optimization level. To check, right-click on your project and open **Properties > C/C++ > Optimization**.

2. Enable Debug information generation:

- Right-click your project and select the **Properties** item in the context menu. The **Property Pages** dialog opens.
- Make sure the Release configuration is selected in the **Configuration** drop-down list.
- From the left pane, select **C++ > General**.
- In the **Debug Information Format** field, choose **Program Database (/Zi)**.
- From the left pane, select **Linker > Debugging**.
- In the **Generate Debug Info** field, select **Generate Debug Information (/DEBUG)**.
- Click **OK** to save your changes and close the dialog box.

These steps cover the most important compiler switches that apply to all C++ applications.

Additional compiler switches are recommended for applications that use OpenMP* or Intel® oneAPI Threading Building Blocks for threading. See the [Compiler Switches for Performance Analysis on Windows* Targets](#) topic for more information.

Once you have the debug information, make sure to set the [Search Directories](#) to point VTune Profiler to the PDB and source files.

Prepare a C++ Application on Linux

To fulfill the [recommendations](#) on Linux, you will need these compiler flags:

```
-O2 -g
```

- The `-O2` flag enables compiler optimizations that favor speed.

NOTE The `-O2` flag is a recommendation to ensure you are profiling the Release version of your application with optimizations that favor speed enabled. If the production use of your application calls for a different optimization level, use your required level. The key idea is to profile your application when it is compiled as close to production use as possible.

- The `-g` flag enables generation of debug information.

On Linux, VTune Profiler requires debug information in the DWARF format to enable source and call stack analysis.

The `-g` option usually produces debugging information in the DWARF format. If you are having trouble generating debug information in the DWARF format, see [Debug Information for Linux Binaries](#).

These steps cover the most important compiler switches that apply to all C++ applications.

Additional compiler switches are recommended for applications that use OpenMP* or Intel® oneAPI Threading Building Blocks for threading. See the [Compiler Switches for Performance Analysis on Linux* Targets](#) topic for more information.

Once you have the debug information, make sure to set the [Search Directories](#) to point VTune Profiler to the binary and source files.

Prepare a SYCL Application

Same [basic recommendations](#) apply to SYCL applications.

Additionally, add these flags to enable functionality specific to accelerators:

This Flag	Does This
<code>-gline-tables-only</code>	Enable generating debug information for GPU analysis of a SYCL application.
<code>-fdebug-info-for-profiling</code>	
<code>-Xsprofile</code>	Enable source-level mapping of performance data for CPU/FPGA Interaction analysis.

(Optional) Instrument Your Code

VTune Profiler also offers the [Instrumentation and Tracing Technology API \(ITT API\)](#) for C++ and Fortran, which enables you to:

- generate and collect trace data for your application
- mark logical sections—such as a multi-step data loading process—of your code and see them in VTune Profiler
- finely balance overhead and amount of trace data
- when necessary, eliminate all ITT API calls at compile time with a single macro, thus getting zero overhead

See the [Instrumentation and Tracing Technology API section](#) for details on configuration and usage.

Windows* Targets

Use the Intel® VTune™ Profiler for the performance analysis of Windows targets.*

Prepare a Windows Target for Analysis

Before you begin analyzing your target for performance, you need to configure and build it as follows:

- Enable downloading [debug information for the system libraries](#) by configuring the Microsoft* Symbol Server.

- Enable [debug information generation for your application binary files](#).
- Build your target in the **Release** mode with the recommended [compiler optimization settings](#).
- Create a baseline against which you can compare the performance improvements as a result of tuning.

For example, you instrument your code to determine how long it takes to compress a certain file. Your original target code, augmented to provide these timing data, serves as your performance baseline. Every time you modify your target, compare the performance metrics of your optimized target with the baseline, to verify that the performance has improved.

Choose a Target from Visual Studio* IDE

For [the VTune Profiler integrated into the Microsoft Visual Studio* IDE](#), you may choose an analysis target and run a performance analysis directly from your development environment.

To choose an analysis target for an existing solution:

1. Open a solution in the **Intel VTune Profiler Results** folder. To display the folder, in the Visual Studio IDE, select **View > Other Windows > Intel VTune Profiler Results**.
2. If the solution contains more than one project, select an appropriate project.

VTune Profiler toolbar and menu items are enabled. By default, the VTune Profiler inherits the Visual Studio settings and uses the application generated for the selected project as your analysis target. You may right-click the project and select



Configure Analysis toolbar button to verify target properties from the menu. By default, the target type is set to **Launch Application**.

To choose an existing standalone executable file:

1. From the Visual Studio menu, choose **File > Open > Project/Solution**.

The **Open Project** dialog box opens.

2. Select the **Executable Files (*.exe)** filter and choose an executable file.

Visual Studio software creates a solution with a single project that contains your executable file. VTune Profiler features are enabled.

3. Right-click the project and select **Intel VTune Profiler Version > Configure Analysis...** option.

The **Configure Analysis** window opens.

4. Click the **Binary/Symbol Search** or **Source Search** button at the bottom to specify search directories. By default, the search directories are defined by the Microsoft Visual Studio* C++ project properties. To view default project search directories for system functions in Visual Studio, right-click the project in the Solution Explorer and select **Properties**.

When finalizing the collected data, the VTune Profiler uses these directories to search for binary (executables and dynamic libraries), symbol (typically .pdb files), and source files supporting your target in the [particular order](#). VTune Profiler automatically locates the files for C/C++ projects which are not moved after building the application and collecting the performance data.

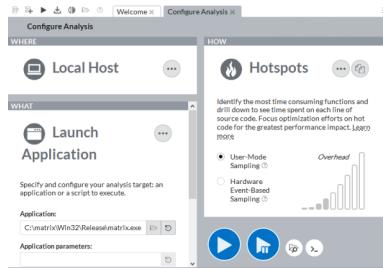
5. Save the solution.

NOTE

Different versions of Visual Studio may have different user interface elements. Refer to the Visual Studio online help for the exact user interface elements that you need to view file location.

Configure a Windows Target

When creating a VTune Profiler project, you access the **Configure Analysis** window and select any of the three available target types for further configuration: **Launch Application**, **Attach to Process**, or **Profile System**. For example, for the **Launch Application** target type, you need to specify an application (and its parameters, if required) for analysis:



When done with the configuration, click the



Browse button on the **HOW** pane on the right to select and run an analysis type.

See Also

[Analysis Target Options](#)

[Install the Sampling Drivers for Windows* Targets](#)

[Analyze Performance](#)

[Search Directories](#)

[Cookbook: Profiling JavaScript* Code in Node.js*](#)

Install the Sampling Drivers for Windows* Targets

NOTE

To install the drivers on Windows* 7 (deprecated) and Windows* Server 2008 R2 operating systems, you must enable the SHA-2 code signing support for these systems by applying [Microsoft Security update 3033929](#). If the security update is not installed, event-based sampling analysis types will not work properly on your system.

To verify the sampling driver is installed correctly on a Microsoft Windows* OS, open the command prompt as an administrator and run the `amplxe-sepreg.exe` utility located at `<install-dir>/bin64`.

To make sure your system meets all the requirements necessary for the hardware event-based sampling collection, enter:

```
amplxe-sepreg.exe -c
```

This command performs the following dependency checks required to install the sampling driver:

- platform, architecture, and OS environment
- availability of the sampling driver binaries: `sepdrv4_x.sys`, `socperf2_x.sys`, and `sepdal.sys`
- administrative privileges
- 32/64-bit installation

To check whether the sampling driver is loaded, enter:

```
amplxe-sepreg.exe -s
```

If the sampling driver is not installed but the system is supported by the VTune Profiler, execute the following command with the administrative privileges to install the driver:

```
amplxe-sepreg.exe -i
```

See Also

[Sampling Drivers](#)

[Install Intel® VTune™ Profiler](#)

Debug Information for Windows* Application Binaries

Intel® VTune™ Profiler requires debug information for the binary files it analyzes to obtain accurate performance data and enable source analysis.

Generate Debug Information in the PDB Format

On Windows* operating systems, debug information is provided in PDB files. Make sure both your system and application libraries/executable have PDB files.

By default, the Microsoft Visual Studio* IDE does not generate PDB information in the **Release** mode. For better results with the VTune Profiler, enable symbol generation manually.

To generate debug information for your binary files:

1. Right-click your C++ project and select the **Properties** item in the context menu.

The **<your_project> Property Pages** dialog box opens.

2. From the **Configuration** drop-down list, choose the **Release** configuration.

It may be already selected if your current configuration in the Visual Studio environment is Release.

3. From the left pane, select **Configuration Properties > C/C++ > General**.
4. In the **Debug Information Format** field, choose **Program Database (/Zi)**.
5. From the left pane, select **Configuration Properties > Linker > Debugging**.
6. In the **Generate Debug Info** field, choose **Generate Debug Information (/DEBUG)**.
7. Click **OK** to close the dialog box.
8. [Compile your target application with optimizations](#).

NOTE

If you configured Visual Studio to generate debug information for your files, you cannot "fix" previous results because the executable and the debug information do not match the executable you used to collect the old results.

To generate a native .PDB file for a native image of .NET* managed assembly:

Use the Native Image Generator tool (`Ngen.exe`) from the .NET Framework. Make sure the search directories, specified in the **Binary/Symbol Search** dialog box, include path to the generated `.pdb` file.

Generate Debug Information for SYCL Applications

To enable performance profiling and generate debug information for SYCL applications running on a GPU, make sure to compile your code with `-gline-tables-only` and `-fdebug-info-for-profiling` options.

See Also

[Debug Information for Windows* System Libraries](#)

Problem: Unknown Frames

Search Directories

Compiler Switches for Performance Analysis on Windows* Targets

Intel® VTune™ Profiler can analyze most native binaries on Windows* target systems. However, the settings below are recommended to make the performance analysis more productive and easier:

Use This Switch	To Do This
/Zi (highly recommended)	Enable generating the symbol information required to associate addresses with source lines and to properly walk the call stack in user-mode sampling and tracing analysis types (Hotspots and Threading).
Release build (highly recommended)	Enable maximum compiler optimization to focus VTune Profiler on performance problems that cannot be optimized with the compiler.
/MD or /MDd	Enable identifying the C runtime calls as system functions and differentiating them from the user code when a proper Call stack mode is applied to the VTune Profiler collection result.
/D "TBB_USE_THREADS=1"	Enable full support for Intel® oneAPI Threaded Building Blocks (oneTBB) in VTune Profiler. Without TBB_USE_THREADS set, the VTune Profiler will not properly identify concurrency issues related to using Intel TBB constructs.
/Qopenmp (highly recommended) (Intel C++ Compiler)	Enable the VTune Profiler to identify parallel regions due to OpenMP* pragmas.
/Qopenmp-link:dynamic (Intel C++ Compiler)	Enable the Intel Compiler to choose the dynamic version of the OpenMP runtime libraries which has been instrumented for the VTune Profiler. Usually, this option is enabled for the Intel Compiler by default.
/Qparallel-source-info=2 (Intel C++ Compiler)	Enable/disable source location emission when OpenMP or auto-parallelism code is generated. 2 is the level of source location emission that tells the compiler to emit path, file, routine name, and line information.
-gline-tables-only	Enable generating debug information for GPU analysis of a SYCL application.
-fdebug-info-for-profiling	
Intel oneAPI DPC++ Compiler	
-Xsprofile	Enable source-level mapping of performance data for FPGA application analysis .

Use This Switch	To Do This
Intel oneAPI DPC++ Compiler	

Explore the list of libraries recommended or not recommended for the user-mode sampling and tracing analysis types:

Library	Recommended	Not Recommended
OpenMP Runtime (supplied by the Intel Compiler)	libomp5md.dll or libguide40.dll	libomp5mt.lib, libguide.lib, vcomp80.dll/vcomp90.dll, or vcomp80d.dll/vcomp90d.dll
C Runtime	msvcr90.dll, msvcr80.dll, msvcr90d.dll, or msvcr80d.dll	libcmt.lib

Avoid These Switches

The following compiler settings are NOT recommended:

Do Not Use This Switch	Because Of This
debug:parallel	Enables the Intel® Parallel Debugger Extension for the Intel Compiler, which is not used for the VTune Profiler.
/Qopenmp-link:static	Chooses the static version of the OpenMP runtime libraries for the Intel Compiler. This version of the OpenMP runtime library does not contain the instrumentation data required for the VTune Profiler analysis.
/Qopenmp_stubs	Prevents OpenMP code from being parallel.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

See Also

[Debug Information for Windows* Application Binaries](#)

[Debug Information for Windows* System Libraries](#)

[Compiler Switches for Performance Analysis on Linux* Targets](#)

Debug Information for Windows* System Libraries

By default, the Microsoft Visual Studio* IDE does not generate PDB information in the **Release** mode. For better results with the Intel® VTune™ Profiler, enable symbol generation manually. For system libraries, use the Microsoft* Symbol Server to download the required PDB files from the Microsoft* web site by selecting any of the options below:

- **Option 1:** Configure the Microsoft* Symbol Server from Visual Studio.
- **Option 2:** Configure the Microsoft Symbol Server from the VTune Profiler Standalone GUI.
- **Option 3:** Set the environment variable.

NOTE

VTune Profiler does not automatically search the Microsoft symbol server for debug information for system files since this functionality:

- Requires an internet connection. Some users are collecting and viewing results on isolated lab systems and do not have internet access.
 - Adds an overhead to finalization of the collection results. For each module without debug information on the local system, a request goes out to the symbol server. If symbols are available, additional time is required to download the symbol file.
 - Uses additional disk space. If symbols for system modules are not used, this disk space is wasted.
 - May be unwanted. Many users do not need to examine details of time spent in system calls and modules. Automatically downloading symbols for system files would be wasteful in this case.
-

Configure the Microsoft* Symbol Server from Visual Studio* IDE

NOTE

The instructions below refer to the Microsoft Visual Studio* 2015 integrated development environment (IDE). They may slightly differ for other versions of Visual Studio IDE.

1. Make sure you have Internet connection available on your machine.
2. Go to **Tools > Options....**
The **Options** dialog box opens.
3. From the left pane, select **Debugging > Symbols**.
4. In the **Symbol file (.pdb) locations** field, select the **Microsoft Symbol Servers** option, typically provided by default, or click the



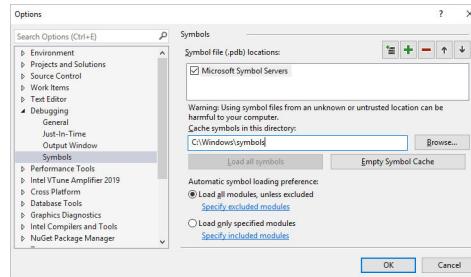
button and add the following address to the list: <http://msdl.microsoft.com/download/symbols>.

5. Make sure the added address is checked.
6. In the **Cache symbols in this directory** field, specify the directory where the downloaded symbol files will be stored.

NOTE

If you plan to download symbols from the Microsoft symbol server only once and then use local storage, use the following syntax for the cache directory: `srv*<local_dir>`. For example:
`srv*C:\Windows\symbols`.

See this example:



7. Click **OK** to close the dialog box.

For newly collected results, the VTune Profiler downloads debug information for system libraries automatically while finalizing the results. For previous results, however, you need to re-finalize the results so that the VTune Profiler can download the debug information for system libraries. To start re-finalizing the result, right-click the result node in the **Solution Explorer** and choose **Re-resolve and Open**.

NOTE

If you use the symbol server, the finalization process may take a long time to complete the first time the VTune Profiler downloads the debug information for system libraries to the local directory specified in the **Options** (for example, C:\Windows\symbols). Subsequent finalizations should be faster.

Configure the Microsoft Symbol Server from the VTune Profiler Standalone GUI

1. Click the



Configure Analysis button on the toolbar.

The [Configure Analysis](#) window opens.

2. Click the

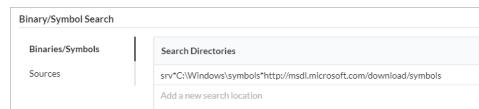


Search Binaries button at the bottom.

3. Add the following string to the list of search directories:

```
srv*C:\local_symbols_cache_location*http://msdl.microsoft.com/download/symbols
```

where *local_symbols_cache_location* is the location of local symbols. The debug symbols for system libraries will be downloaded to this location.



NOTE

If you specify different directories for different projects, the files will be downloaded multiple times, adding unwanted overhead. If you have a Visual Studio project that defines a cache directory for the symbol server, use the same directory in the standalone VTune Profiler so that you do not waste time and space downloading symbols that already exist in a cache directory.

Set the Environment Variable

Set the environment variable (system or user) `_NT_SYMBOL_PATH` to
`srv*C:\local_symbols_cache_location*http://msdl.microsoft.com/download/symbols`.

See Also

[Debug Information for Windows* Application Binaries](#)

[Enable Linux* Kernel Analysis](#)

[Compiler Switches for Performance Analysis on Windows* Targets](#)

[Prepare an Android* Application for Analysis](#)

Add Administrative Privileges

To enable such options as detecting context switches or highly accurate CPU time collection, you need local administrator privileges for running the product.

To run a standalone version of the Intel® VTune™ Profiler as an administrator, right-click the product entry in the **Start** menu and select **Run as administrator** from the context menu.

To run the Intel® VTune™ Profiler integrated into Visual Studio* IDE as the administrator, do the following:

1. From the **Start** menu, select **All Programs > Intel Studio version** and right-click **Intel Studio version with VS version** option.
The context menu opens.
2. From the context menu, select the **Run as administrator** option.
Microsoft Visual Studio* IDE opens with the administrative privileges assigned to your name.

See Also

[Highly Accurate CPU Time Data Collection](#)

Linux* Targets

Use the Intel® VTune™ Profiler for performance analysis on local and remote Linux target systems.*

To analyze your Linux target, do the following:

1. Prepare your target application for analysis:
 - Enable downloading [debug information for system kernels](#) by installing debug info packages available for your system version.
 - Enable downloading [debug information for the application binaries](#) by using the `-g` option when compiling your code. Consider using the [recommended compiler settings](#) to make the performance analysis more effective.
 - Build your target in the **Release** mode.
 - Create a baseline against which you can compare the performance improvements as a result of tuning.

For example, you instrument your code to determine how long it takes to compress a certain file. Your original target code, augmented to provide these timing data, serves as your performance baseline. Every time you modify your target, compare the performance metrics of your optimized target with the baseline, to verify that the performance has improved.

2. Prepare your target system for analysis:
 - [Build and install the sampling drivers](#), if required.

NOTE

- If the drivers were not built and set up during installation (for example, lack of privileges, missing kernel development RPM, and so on), VTune Profiler provides an error message and enables [driverless sampling data collection](#) based on the Linux Perf* tool functionality, which has a limited scope of analysis options.
- On Ubuntu* systems, VTune Profiler may fail to collect Hotspots and Threading analysis data if the scope of the `ptrace()` system call application is limited.

To workaround this issue for one session, set the value of the `kernel.yama.ptrace_scope` sysctl option to 0 with this command:

```
sysctl -w kernel.yama.ptrace_scope=0
```

To make this change permanent, see the corresponding [Troubleshooting topic](#).

- For remote analysis, [configure SSH connection](#) and [set up your remote Linux system](#) depending on the analysis [usage mode](#).
- 3.** Create a VTune Profiler [project](#) and run the [performance analysis](#) of your choice.

Ubuntu* Systems

See Also

[Compiler Switches for Performance Analysis on Linux* Targets](#)

[Set Up Remote Linux* Target](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

Build and Install the Sampling Drivers for Linux* Targets

Prerequisites for remote Linux target systems: You must have root access to the target system.

Prerequisites for all Linux systems: Sampling driver sources. You can find the sampling driver sources for the local system in the `<install_dir>/sepdk` folder of your VTune Profiler installation. For remote targets, locate the target packages for the desired system in the `<install_dir>/target` folder of your installation, copy the package to the target system, extract it, and build the driver.

Install Drivers on Linux* Host Systems

During product installation on a host Linux OS, you can control the installation options for drivers with settings in **Advanced Options**. VTune Profiler provides the following options:

Use This Option	To Do This
Sampling driver install type [build driver (default) / driver kit files only]	Choose the driver installation option. By default, VTune Profiler uses the Sampling Driver Kit to build the driver for your kernel. You may change the option to driver kit files only if you want to build the driver manually after installation.
Driver access group [vtune (default)]	Set the driver access group ownership to determine which set of users can perform the collection on the system. By default, the group is <code>vtune</code> . Access to this group is not restricted. To restrict access, see the Driver permissions option below. You may set your own group during installation in the Advanced options or change it manually after installation by executing: <code>./boot-script --group <your_group></code> from the <code><install-dir>/sepdk/src</code> directory.

Use This Option	To Do This
Driver permissions [660 (default)]	Change permissions for the driver. By default, only a <code>vtune</code> group user can access the driver. Using this access the user can profile the system, an application, or attach to a process.
Load driver [yes (default)]	Load the driver into the kernel.
Install boot script [yes (default)]	Use a boot script that loads the driver into the kernel each time the system is rebooted. The boot script can be disabled later by executing: <code>./boot-script --uninstall</code> from the <code><install-dir>/sepdk/src</code> directory.
Enable per-user collection mode [no (default) / yes]	<p>Install the hardware event-based collector driver with the per-user filtering on. When the filtering is on, the collector gathers data only for the processes spawned by the user who started the collection. When it is off (default), samples from all processes on the system are collected. Consider using the filtering to isolate the collection from other users on a cluster for security reasons. The administrator/root can change the filtering mode by rebuilding/restarting the driver at any time. A regular user cannot change the mode after the product is installed.</p> <p>NOTE For MPI application analysis on a Linux* cluster, you may enable the Per-user Hardware Event-based Sampling mode when installing the Intel Parallel Studio XE Cluster Edition. This option ensures that during the collection the VTune Profiler collects data only for the current user. Once enabled by the administrator during the installation, this mode cannot be turned off by a regular user, which is intentional to preclude individual users from observing the performance data over the whole node including activities of other users. After installation, you can use the respective <code>vars.sh</code> files to set up the appropriate environment (PATH, MANPATH) in the current terminal session.</p>
Driver build options ...	Specify the location of the kernel header files on this system, the path and name of the C compiler to use for building the driver, the path and name of the make command to use for building the driver.

Check Sampling Driver Installation

To verify that the sampling driver is installed correctly on the host Linux system:

1. Check whether the sampling drivers are installed:

```
$ cd <install-dir>/sepdk/src
$ ./insmod-sep -q
```

This provides information on whether the drivers are currently loaded and, if so, what the group ownership and file permissions are on the driver devices.

2. Check group permissions.

If drivers are loaded, but you are not a member of the group listed in the `query` output, request your system administrator to add you to the group. By default, the driver access group is `vtune`. To check which groups you belong to, type `groups` at the command line. This is only required if the permissions are other than 666.

NOTE

If there is no collection in progress, there is no execution time overhead of having the driver loaded and very little overhead for memory usage. You can let the system module be automatically loaded at boot time (for example, via the `install-boot-script` script, used by default). Unless the data is being collected by the VTune Profiler, there will be no latency impact on the system performance.

Verify Kernel Configuration

To verify kernel configuration:

1. Make sure that the kernel header sources are present on your host system. The kernel version should be 2.6.28 or later. To find the kernel version, explore `kernel-src-dir/include/linux/utsrelease.h`, or, depending on the kernel version: `kernel-src-dir/include/generated/utsrelease.h`. For more details, see the `README.txt` file in the `sepdk/src` directory.
2. Make sure the following options are enabled in the kernel configuration for hardware event-based sampling (EBS) collection:
 - `CONFIG_MODULES=y`
 - `CONFIG_MODULE_UNLOAD=y`
 - `CONFIG_PROFILING=y` (kernel versions 5.16 or older)
 - `CONFIG_SMP=y`
 - `CONFIG_KALLSYMS=y`
 - `CONFIG_TRACEPOINTS=y` (required for kernel versions 5.17 and newer; recommended for all other versions)
 - `CONFIG_KPROBES=y` (kernel versions 5.17 and newer)
3. In addition to the options above, make sure the following options are enabled in the kernel configuration for EBS collection *with stacks*:
 - `CONFIG_KRETPROBES=y`
 - `CONFIG_FRAME_POINTER=y` (optional but recommended for [Call Stack Mode](#))
4. For remote target systems, determine if signed kernel modules are required (`CONFIG_MODULE_SIG_FORCE=y`). If they are, you must have the signed key that matches your target system.

If you are building the sampling drivers from a fresh kernel source and want to use it for an existing target system, get the original key files and sign the sampling driver with the original key. Alternatively, build the new kernel and flash it to the target device so the target device uses your kernel build.

Build the Sampling Driver

Prerequisites:

- You need kernel header sources and other additional software to build and load the kernel drivers on Linux. Refer to the [Verify kernel configuration](#) section.
- To cross-build drivers for a remote target Linux system, extract the package from the `<install-dir>/target` folder to `<extract_dir>`.

NOTE

If the current version of the sampling driver that is shipped with the VTune Profiler installation does not suit your needs, for example, due to a recent change in the Linux* kernel, you can find the latest version of the sampling driver on the [Sampling Driver Downloads](#) page.

To build the driver if it is missing:

1. Change the directory to locate the build script:

- To build drivers for a local system: \$ cd <install-dir>/sepdk/src
- To cross-build drivers for a remote target system: \$ cd <extract-dir>/sepdk/src

2. Use the `build-driver` script to build the drivers for your kernel. For example:

- \$./build-driver

The script prompts the build option default for your local system.

- \$./build-driver -ni

The script builds the driver for your local system with default options without prompting for your input.

- \$./build-driver -ni -pu

The script builds the driver with the per-user event-based sampling collection enabled, without prompting for your input.

- \$./build-driver -ni \

```
--c-compiler=i586-i586-xxx-linux-gcc \
--kernel-version="<kernel-version>" \
--kernel-src-dir=<kernel-source-dir> \
--make-args="PLATFORM=x32 ARITY=smp"
--install-dir=<path>
```

The script builds the drivers with a specified cross-compiler for a specific kernel version. This is usually used for the cross-build for a remote target system on the current host. This example uses the following options:

- `-ni` disables the interactive during the build.
- `--c-compiler` specifies the cross build compiler. The compiler should be available from the PATH environment. If the option is not specified, the host GCC compiler is used for the build.
- `--kernel-version` specifies the kernel version of the target system. It should match the `uname -r` output of your target system and the `UTS_RELEASE` in `kernel-src-dir/include/generated/utsrelease.h` or `kernel-src-dir/include/linux/utsrelease.h`, depending on your kernel version.
- `--kernel-src-dir` specifies the kernel source directory.
- `--make-args` specifies the build arguments. For a 32-bit target system, use `PLATFORM=x32`. For a 64-bit target system, use `PLATFORM=x32_64`
- `--install-dir` specifies the path to a writable directory where the drivers and scripts are copied after the build succeeds.

Use `./build-driver -h` to get the detailed help message on the script usage.

To build the sampling driver as RPM using build services such as Open Build Service (OBS):

Use the `sepdk.spec` file located at the `<install-dir>/sepdk/src` directory.

Install the Sampling Drivers

Prerequisites for remote target systems: Copy the `sepdk/src` folder or the folder specified by the `--install-dir` option when building the driver to the target system using ssh, ftp, adb, sdb, or other supported means.

To install the drivers:

1. If building the drivers succeeds, install them manually with the `insmod-sep` script:

```
$ cd <install_dir>/sepdk/src  
$ ./insmod-sep -r -g <group>
```

where `<group>` is the group of users that have access to the driver.

To install the driver that is built with the per-user event-based sampling collection on, use the `-pu` (-per-user) option as follows:

```
$ ./insmod-sep -g <group> -pu
```

If you are running on a resource-restricted environment, add the `-re` option as follows:

```
$ ./insmod-sep -re
```

2. Enable the Linux system to automatically load the drivers at boot time:

```
$ cd <install_dir>/sepdk/src
```

To install the driver that is built with the per-user event-based sampling collection on, use the `-pu` (-per-user) option as follows:

```
$ ./boot-script --install -g <group> -pu
```

The `-g <group>` option is only required if you want to override the group specified when the driver was built.

To remove the driver on a Linux system, run:

```
./rmmod-sep -s
```

See Also

[Cookbook: Profiling Hardware without Sampling Drivers](#)

[Sampling Drivers](#)

[Install Intel® VTune™ Profiler](#)

[Embedded Linux* Targets](#)

[Configure Yocto Project* and Intel® VTune™ Profiler with the Linux* Target Package](#)

[Error Message: No Pre-built Driver Exists for This System](#)

Debug Information for Linux* Application Binaries

Intel® VTune™ Profiler requires debug information for the binary files it analyzes to obtain accurate performance data and enable source analysis.

Debug Information for Performance Analysis

If your system and application modules have debug information, the VTune Profiler is able to provide full-scale statistics on call stacks, source data, function names, and so on. For example, you may use the [Call Stack Mode](#) on the filter toolbar to select the **User/system functions** option and view data on both user and system functions.

If the VTune Profiler does not find debug information for the binaries, it statically identifies function boundaries and assigns hotspot addresses to generated pseudo names `func@address` for such functions, for example:

Function / Call Stack	CPU Time ▼
▼ func@0x6b29db95	2.405s
▶ ↵ pthread_mutex_lock ← draw_task	2.370s
▶ ↵ video::next_frame ← draw_task::o	0.036s
▶ GdipDrawImagePointRectI	0.990s

If a module is not found or the name of a function cannot be resolved, the VTune Profiler displays module identifiers within square brackets, for example: `[module]`.

If the debug information is absent, the VTune Profiler may not unwind the call stack and display it correctly in the **Call Stack** pane. Additionally in some cases, it can take significantly more time to [finalize](#) the results for modules that do not have debug information.

Generate Debug Info in the DWARF Format

Compile your code using the `-g` option that usually produces debugging information in the DWARF format.

If DWARF is not a default debugging information format for the compiler, or if you are using MinGW/Cygwin GCC*, use the `-gdwarf-version` option, for example: `-gdwarf-2` or `-gdwarf-3`.

Generate Debug Info File for the ELF Format

You can create separate debug info files and link them with an executable/library via debug link or build ID. Please refer to the GNU* Binutils documentation for more details.

VTune Profiler recognizes both types of linking:

- If an executable file in the ELF format contains a build ID and has a separate debug info file with the name generated by the build ID, the VTune Profiler is able to find and validate the separate symbol file if proper [search directories](#) are set. While searching the symbol file, the VTune Profiler checks the `.build-id` subdirectory of each search directory for a file named `hh/hhhhhhhhh.debug` where `hh` is the first 2 hexadecimal characters of build ID and `hhhhhhhhhh` is the remaining part.
- If an executable file contains a debug link (specified in the `.gnu_debuglink` section) with a name of separate debug info file, VTune Profiler tries to find it.

Generate Debug Information for SYCL* Applications

To enable performance profiling and generate debug information for SYCL applications running on a GPU, make sure to compile your code with `-gline-tables-only` and `-fdebug-info-for-profiling` options.

Generate Debug Information for OpenMP* Offload Applications

When you build OpenMP* Offload applications with the [Intel® oneAPI DPC++/C++ Compiler](#) or Intel Fortran compiler, compile your code with the `--info-for-profiling` switch.

NOTE When using the Intel Fortran compiler to compile OpenMP Offload code, make sure to use the `-debug offload` option.

See Also

[Compiler Switches for Performance Analysis on Linux* Targets](#)

[Enable Linux* Kernel Analysis](#)

[Problem: Unknown Frames](#)

[Search Directories](#)

Compiler Switches for Performance Analysis on Linux* Targets

Intel® VTune™ Profiler can analyze most native binaries on Linux target systems. However, the settings below are recommended to make the performance analysis more productive and easier:

Use This Switch	To Do This
<code>-g</code> (highly recommended)	Enable generating the symbol information required to associate addresses with source lines and to properly walk the call stack in user-mode sampling and tracing collection types (Hotspots and Threading).
Release build or <code>-O2</code> (highly recommended)	Enable maximum compiler optimization to focus the VTune Profiler on real performance problems that cannot be optimized with the compiler.
<code>-shared-intel</code> (Intel® C++ Compiler)	Enable identifying the <code>libm</code> and C runtime calls as system functions and differentiating them from the user code when a proper filter mode is applied to the VTune Profiler collection result.
<code>-shared-libgcc</code> (GCC* Compiler)	
<code>-debug inline-debug-info</code>	Enable the VTune Profiler to identify inline functions and, according to the selected inline mode , associate the symbols for an inline function with the inline function itself or its caller. This is the default mode for GCC* 4.1 and higher.
(Intel C++ Compiler)	<p>NOTE The <code>debug inline-debug-info</code> option is enabled by default for the Intel® oneAPI DPC++/C++ Compiler if you compile with optimizations (<code>-O2</code> or higher) and debug information (<code>-g</code> option).</p>
<code>-DTBB_USE_THREADING_TOOLS</code>	<p>Enable Intel® oneAPI Threading Building Blocks Analysis (oneTBB) for the VTune Profiler. This macro is automatically set if you compile with <code>-D_DEBUG</code> or <code>-DTBB_USE_DEBUG</code>.</p> <p>Without <code>TBB_USE_THREADING_TOOLS</code> set, the VTune Profiler will not properly identify concurrency issues related to using oneTBB constructs.</p>
<code>-qopenmp</code> (highly recommended)	Enable the VTune Profiler to identify parallel regions due to OpenMP* pragmas.

Use This Switch	To Do This
(Intel C++ Compiler)	
-qopenmp-link dynamic (Intel C++ Compiler)	Enable the Intel Compiler to choose the dynamic version of the OpenMP runtime libraries which has been instrumented for the VTune Profiler. Usually, this option is enabled for the Intel Compiler by default.
-parallel-source-info=2 (Intel C++ Compiler)	Enable/disable source location emission when OpenMP or auto-parallelism code is generated. 2 is the level of source location emission that tells the compiler to emit path, file, routine name, and line information.
--info-for-profiling Intel oneAPI DPC++ Compiler Intel Fortran Compiler	Enable generating debug information for GPU analysis of a SYCL application. Generate debug information for OpenMP* Offload applications compiled by Intel Fortran compiler
-Xsprofile Intel oneAPI DPC++ Compiler	Enable source-level mapping of performance data for FPGA application analysis .

Avoid These Switches

The following compiler settings are NOT recommended:

Do Not Use This Switch	Because Of This
Debug build or -O0	Changes the performance of your application compared to a release build and may dramatically impact the performance profiling potentially causing you to analyze and attempt optimization on a section of code that is not a performance problem in the release build.
-static -static-libgcc	Prevents the VTune Profiler from being able to run the user-mode sampling and tracing analysis types. See below for more details.
	<p>NOTE</p> <p>When you specify the <code>-fast</code> switch with the Intel Compiler, it automatically enables <code>-static</code>.</p>
-static-intel	Prevents the user-mode sampling and tracing analysis types from distinguishing system functions properly. This is the default option for the Intel Compiler.
-qopenmp-link static	Chooses the static version of the OpenMP runtime libraries for the Intel Compiler. This version of the OpenMP runtime library does not contain the instrumentation data required for the VTune Profiler analysis.

Do Not Use This Switch	Because Of This
-qopenmp_stub s	Prevents OpenMP code from being parallel.
-mssse4a, -m3dnow	Generates binaries that use instructions not supported by Intel processors, which may cause unknown behavior when profiling with the VTune Profiler.
-debug [parallel extended emit-column expr- source-pos semantic- stepping variable- locations]	VTune Profiler works best with <code>-debug full</code> (the default mode when using <code>-g</code>). Other options including <code>parallel</code> , <code>extended</code> , <code>emit-column</code> , <code>expr-source-pos</code> , <code>semantic-stepping</code> , and <code>variable-locations</code> are not supported by the VTune Profiler. See <code>-debug inline-debug-info</code> for more information.
-coarray	Prevents the Threading analysis from identifying properly the locks that disable scaling in Coarray Fortran.

Compiling for the User-Mode Sampling and Tracing Analysis

For successful user-mode sampling and tracing analysis (Hotspots and Threading) of your executable and all shared libraries, use the following switches to properly walk through the call stack:

- Use `-g` to generate the symbol information and enable the source code analysis.
- Use `-fno-omit-frame-pointer` to enable the frame pointers analysis.

NOTE

There are other options that may add frame pointers to your binary as a side effect, for example: `-fexceptions` (default for C++) or `-O0`. To make sure the executable (and shared libraries) have this information, use the `objdump -h <binary>` command and make sure you see the `.eh_frame_hdr` section there.

User-mode sampling and tracing analysis types work better with dynamic versions of the following libraries:

Library	Dynamic Version (Recommended)	Static Version (Not Recommended)
OpenMP Runtime (supplied by the Intel Compiler)	<code>libiomp5.so</code> or <code>libguide40.so</code>	<code>libiomp5.a</code> or <code>libguide4.a</code>
Posix Thread	<code>libpthread.so</code>	<code>libpthread.a</code>
C Runtime	<code>libc.so</code>	<code>libc.a</code>

Library	Dynamic Version (Recommended)	Static Version (Not Recommended)
C++ Runtime	libstdc++.so	libstdc++.a
Intel Libm	libm.so	libm.a

User-mode sampling and tracing collection has the following limitations for analyzing statically linked libraries/functions:

- The static version of the OpenMP runtime library supplied by the Intel Compiler does not provide the necessary instrumentation for the Threading analysis type.
- Call Stack mode cannot properly distinguish user code from system functions.
- User-mode sampling and tracing collection cannot execute unless various C Runtime functions are exported. There are multiple ways to do this; for example, use the `-u` command of the GCC compiler:

- `-u malloc`
- `-u free`
- `-u realloc`
- `-u getenv`
- `-u setenv`
- `-u __errno_location`

If your application creates Posix threads (either explicitly or via the static OpenMP library or some other static library), you need to explicitly define the following additional functions:

- `-u pthread_key_create`
- `-u pthread_key_delete`
- `-u pthread_setspecific`
- `-u pthread_getspecific`
- `-u pthread_spin_init`
- `-u pthread_spin_destroy`
- `-u pthread_spin_lock`
- `-u pthread_spin_trylock`
- `-u pthread_spin_unlock`
- `-u pthread_mutex_init`
- `-u pthread_mutex_destroy`
- `-u pthread_mutex_trylock`
- `-u pthread_mutex_lock`
- `-u pthread_mutex_unlock`
- `-u pthread_cond_init`
- `-u pthread_cond_destroy`
- `-u pthread_cond_signal`
- `-u pthread_cond_wait`
- `-u __pthread_cleanup_push`
- `-u __pthread_cleanup_pop`
- `-u pthread_setcancelstate`
- `-u pthread_self`
- `-u pthread_yield`

The easiest way to do this is by creating a file with the above options and passing it to `gcc` or `ld`. For example:

```
gcc -static mysource.cpp @Cdefs @Pdefs
```

where `Cdefs` is a file with options for the required C functions and `Pdefs` is a file with the options for the required POSIX functions.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

See Also

[Compiler Switches for Performance Analysis on Windows* Targets](#)

[Debug Information for Windows* Application Binaries](#)

[Enable Linux* Kernel Analysis](#)

[Analyze Statically Linked Binaries on Linux* Targets
on Linux targets](#)

Enable Linux* Kernel Analysis

For successful performance analysis of the kernel and system libraries, do the following:

1. Enable kernel modules resolution.
2. Download and install debug info packages available for your Linux system version.
3. Build the Linux kernel with debug information.

Enable Kernel Modules Resolution

To provide accurate performance statistics for the Linux kernel, the VTune Profiler requires kernel modules information provided in the `/proc/kallsyms` file. Make sure the `/proc/sys/kernel/kptr_restrict` file contains values that enable reading `/proc/kallsyms` and providing non-zero addresses for the kernel pointers:

- If the `kptr_restrict` value is 0, kernel addresses are provided without limitations (recommended).
- If the `kptr_restrict` value is 1, addresses are provided if the current user has a `CAP_SYSLOG` capability.
- If the `kptr_restrict` value is 2, the kernel addresses are hidden regardless of privileges the current user has.

See more details at: <http://lwn.net/Articles/420403/>, <http://man7.org/linux/man-pages/man7/capabilities.7.html>.

If kernel pointers information was explicitly hidden by setting the `kptr_restrict` to a non-zero value, hardware event-based analysis results may not contain functions from kernel modules. As a result, you may see the CPU time associated with the [Outside any known module] item. To workaround this problem for the current session, set the contents of the `/proc/sys/kernel/kptr_restrict` file to 0 before starting the VTune Profiler as follows:

```
sysctl -w kernel.kptr_restrict=0
```

NOTE

To enable kernel profiling without the Intel Sampling Driver via perf, set the `perf_event_paranoid` value to <= 1. See the [Linux kernel documentation](#) for details.

To resolve symbols for the Linux kernel, the VTune Profiler also uses the `System.map` file created during the kernel build and shipped with the system by default. If the file is located in a non-default directory, you may add it to the list of search directories in the **Binary/Symbol Search** dialog box when configuring your target properties.

NOTE

The settings in the `/proc/kallsyms` and `System.map` file enable the VTune Profiler to resolve kernel symbols and view kernel functions and kernel stacks but do not enable the assembly analysis.

Download and Install Available Debug Kernel Versions

After installing the Linux operating system, the kernel is contained in `vmlinuz`, or `vmlinuz`, or `bzImage` in `/boot`. Linux vendors typically release compressed kernel files stripped of symbols (`vmlinuz` or `bzImage`). `vmlinuz` is the uncompressed Linux kernel, but it does not include debug information. So, by default the VTune Profiler cannot retrieve kernel function information from these kernels and presents all hot addresses captured in the kernel as a unique function or module named `[vmlinuz]`. However, some vendors have released special debug versions of their kernels that are suitable for performance analysis.

1. Use the `uname -r` command to identify the running Linux kernel version.
2. Download and install two RPMs matching your system: `kernel-debug-debuginfo-* .rpm` and `kernel-debuginfo-common-* .rpm`. To do this, use any of the following options:
 - Browse through the RPMs on your installation CDs or DVDs. For example, for SuSE Linux Enterprise* 9, 10, and 11 distros, SuSE provides "debug" kernel RPMs (`kernel-debug-* .rpm`) available on the install CD or from the website. After installing the RPM, the debug version of the kernel file is located under `/boot/vmlinuz-*-debug` or under `/boot/vmlinuz-*-*-debug`. You need to manually uncompress this kernel file using the `gunzip` program.
 - Browse through the OS vendor FTP site and download the packages. For example: look at `ftp://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/os` to get packages for Redhat* Enterprise Server.
 - Look for other sources on the internet. For example, for Red Hat Enterprise* Linux 3, 4 and 5 distros, Red Hat provides `debuginfo` RPMs at `http://people.redhat.com/duffy/debuginfo/`. After installing the RPM, the debug version of the kernel file is located under `/usr/lib/debug/boot (EL 3)` or `/usr/lib/debug/lib/modules (EL 4, 5)`.
3. Use the following commands to install the RPMs:

```
rpm -ivh kernel-debuginfo-common-* .rpm
rpm -ivh kernel-debug-debuginfo-* .rpm
```

For some operating systems, you can use `yum` to install packages directly, for example:

```
yum --enablerepo=rhel-debuginfo install kernel-debuginfo
```

4. Verify that the packages have been installed, for example:

```
rpm -qa | grep kernel
```

5. Modify the VTune Profiler target properties and specify the path to the uncompressed kernel binary in the [Dialog Box: Binary/Symbol Search](#), for example: `/usr/lib/debug/lib/modules/2.6.18-128.el5debug/`.

Build the Linux Kernel with Debug Information

1. [Configure the kernel sources.](#)
2. Edit the kernel source top-level Makefile and add the `-g` option to the following variables:


```
CFLAGS_KERNEL := -g
CFLAGS := -g
```
3. Run `make clean; make` to create the `vmlinux` kernel file with debug information. Once a debug version of the kernel is created or obtained, specify that kernel file as the one to use during performance analysis.

As soon as the debug information is available for your kernel modules, any future analysis runs will display the kernel functions appropriately. To resolve the previously collected data against this new symbol information, update the project [Search Directories](#) and click the **Re-resolve** button to apply the changes.

See Also

[Debug Information for Windows* Application Binaries](#)

[Compiler Switches for Performance Analysis on Linux* Targets on Linux* targets](#)

[Analyze Statically Linked Binaries on Linux* Targets](#)

[Call Stack Mode](#)

Resolution of Symbol Names for Linux-Loadable Kernel Modules

To resolve symbol information for Linux kernel modules, Intel® VTune™ Profiler uses content in the `/sys/module/<module-name>/sections/` directory during the finalization step.

Default permissions for the `/sys/module/<module-name>/sections/` directory may allow access only for the root user. In this case, VTune Profiler reports a warning message. Run VTune Profiler with root privileges or change permissions for all files in this directory.

Limitations

When you collect data on a remote Linux system, VTune Profiler does not read `/sys/module/<module-name>/sections/*` for results. In this case, to resolve symbols properly:

1. Copy the `<module-name>/sections` folder manually from the target system to `.../parent directory/<module-name>/sections` on the host system.
2. Add `<parent directory>` to VTune search directories for binary and symbol files.

See Also

[Compiler Switches for Performance Analysis on Linux* Targets](#)

[Enable Linux* Kernel Analysis](#)

[Problem: Unknown Frames](#)

[Search Directories](#)

Analyze Statically Linked Binaries on Linux* Targets

To profile a statically linked binary file, temporary stop stripping the binary file during compilation and make sure the binary file exports the following symbols from system libraries:

- `_init()` in the main executable: if you profile a tree of processes, consider using the `strategy` option.
- `libc.so`:
 - A target exports `setenv`, `getenv()`, and `__errno_location()` symbols unconditionally.
 - If a target employs `recv()` API, it exports `recv()` and `poll()`.
 - If a target employs `sleep()` or `usleep()` APIs, it exports `sleep()` or `usleep()` respectively, and `nanosleep()` symbol.
- `libpthread.so`:
 - If a target employs `pthread_create()` API, it exports the following symbols:
 - `pthread_create()`
 - `pthread_key_create()`
 - `pthread_setspecific()`
 - `pthread_getspecific()`
 - `pthread_self()`
 - `pthread_getattr_np()`
 - `pthread_attr_destroy()`
 - `pthread_attr_setstack()`
 - `pthread_attr_getstack()`
 - `pthread_attr_getstacksize()`
 - `pthread_attr_setstacksize()`
 - If a target employs `pthread_cancel()` API, it exports the following symbols:
 - `pthread_cancel()`
 - `_pthread_cleanup_push()`
 - `_pthread_cleanup_pop()`
 - If a target employs `_pthread_cleanup_push()` or `_pthread_cleanup_pop()` API, it exports the following symbols:
 - `_pthread_cleanup_push()`
 - `_pthread_cleanup_pop()`
 - If a target employs `pthread_mutex_lock()` API, it exports `pthread_mutex_lock()` and `pthread_mutex_trylock()` symbol.
 - If a target employs `pthread_spin_lock()` API, it exports `pthread_spin_lock()` and `pthread_spin_trylock()` symbol.
- `libdl.so`:

If a target employs any of `dlopen()`, `dlsym()`, or `dlclose()` APIs, it exports all three of them simultaneously.

If the binary file does not export some of the symbols above, use the `-u` linker switch (for example, specify `-Wl,-u__errno_location` if you use compiler for linking) to include symbols into the binary file at the linking stage of compilation.

See Also

[Compiler Switches for Performance Analysis on Linux* Targets](#)

[Control Data Collection](#)

Set Up Remote Linux* Target

Use the Intel® VTune™ Profiler installed on the Windows, Linux*, or macOS* host to analyze code performance on remote Linux systems.*

VTune Profiler supports the following usage modes for remote analysis of Linux applications on regular and embedded systems:

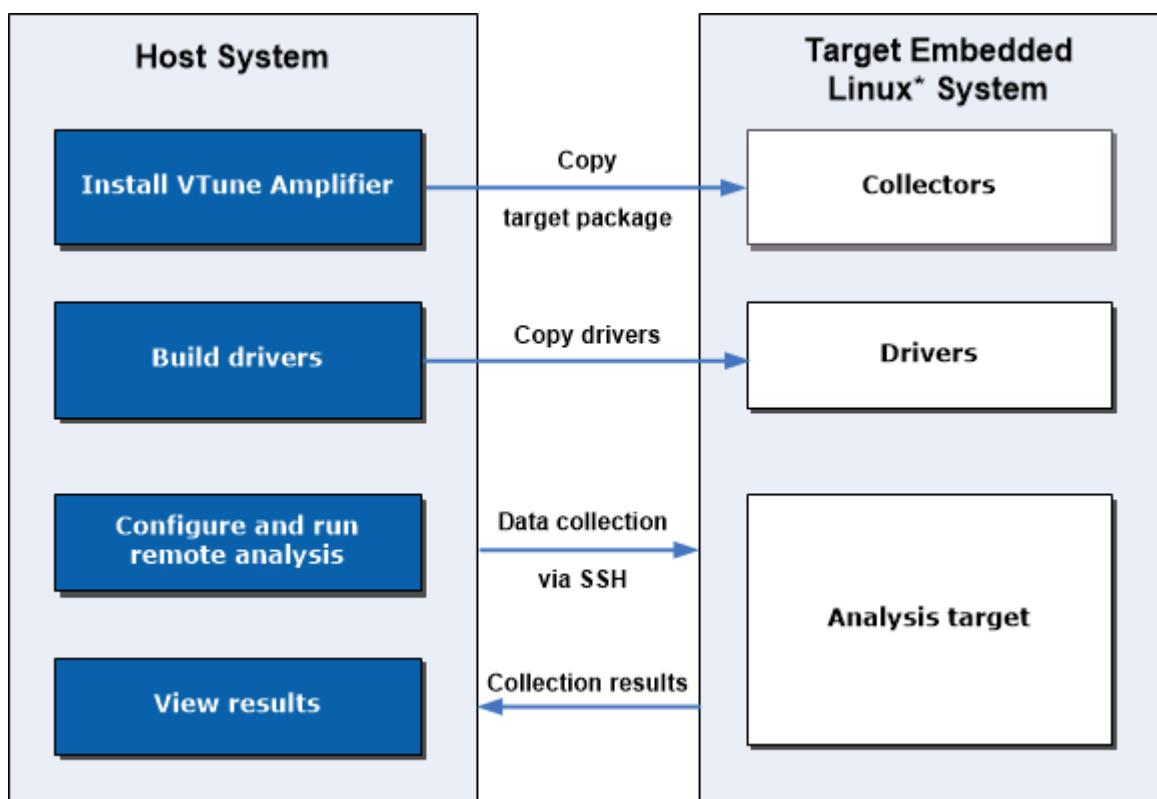
- [Remote CLI \(vtune\)](#) or [GUI \(vtune-gui\)](#) (recommended for regular and embedded systems)
- [Native CLI](#) with installing and running `vtune` directly on a remote Linux system
- [Native SEP](#) with `sep` (recommended for tiny embedded systems)

Remote CLI and GUI Usage Mode

Requirements for the target system: ~25 MB disk space

This mode is recommended for most cross-development scenarios supported by the VTune Profiler, especially if your target system is resource-constrained (insufficient disk space, memory, or CPU power) or if you use a highly customized Linux target system.

To collect data on a remote Linux system:



- | | |
|--|---|
| 1. Install VTune Profiler | Install the full-scale VTune Profiler product on the host system. |
| 2. Prepare your target system for analysis | <ol style="list-style-type: none"> 1. Set up a password-less SSH access to the target using RSA keys. 2. Install the VTune Profiler target package with data collectors on the target Linux system. |

NOTE

If you choose to install the target package to a non-default location, make sure to specify the correct path either with the **VTune Profiler installation directory on the remote system** option in the **WHERE** pane (GUI) or with the `-target-install-dir` option (CLI).

3. **Build the drivers** on the host (if required), copy them to the target system and install the drivers.

NOTE

To build the sampling driver as RPM using build services as Open Build Service (OBS), use the `sepdk.spec` file located at `<install_dir>/sepdk/src` the directory.

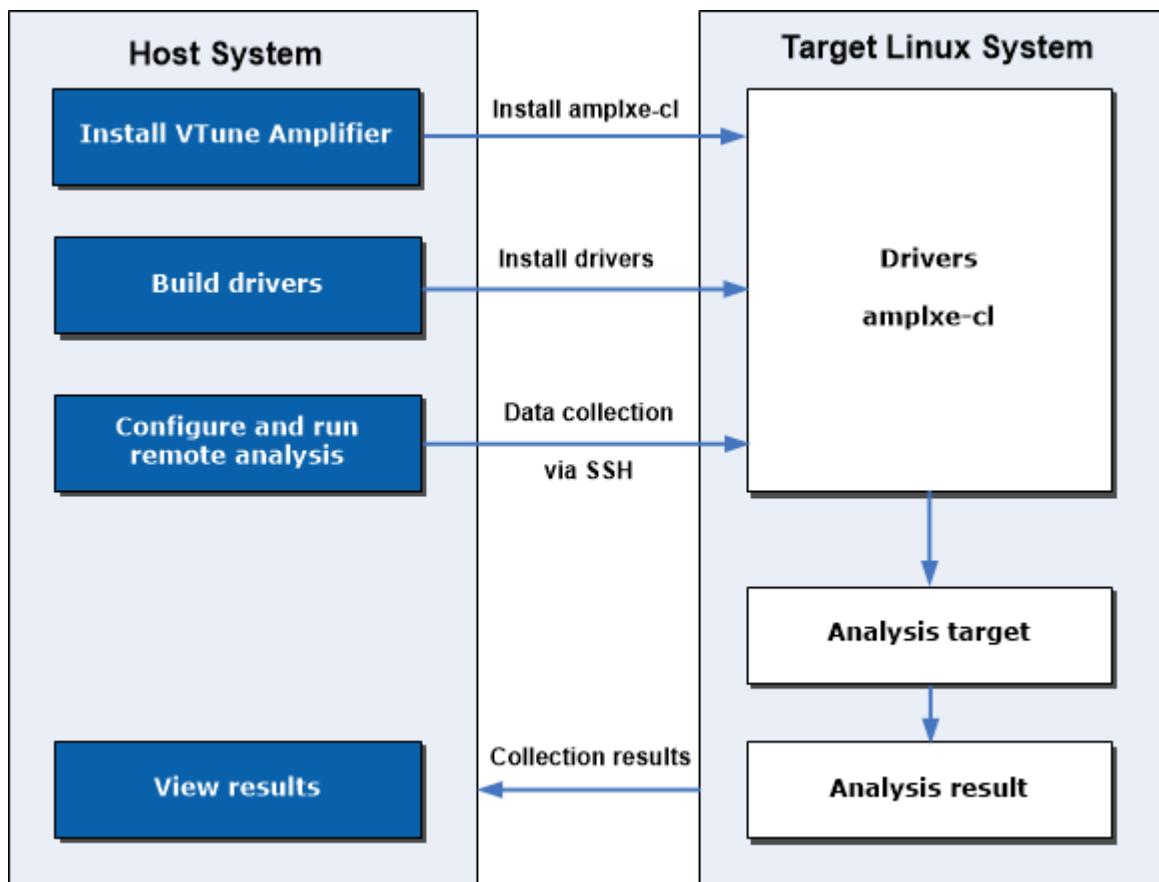
- | | |
|--------------------------------------|---|
| 3. Configure and run remote analysis | <ol style="list-style-type: none">1. On your host system, open the VTune Profiler GUI and select Configure Analysis.2. In the Where pane, specify an SSH connection to a remote Linux system.3. In the What pane, specify your target application on the remote system. Make sure to specify search directories for symbol/source files required for finalization on the host.4. In the How pane, choose and configure an analysis type.5. Start the analysis. <p>VTune Profiler launches your application on the target, collects data, copies the analysis result and binary files to the host, and finalizes the data.</p> |
| 4. View results | View the collected data on the host. |

Native Usage Mode

Requirements for the target system: ~200 MB disk space.

This mode is recommended for regular Linux target systems from supported operating systems listed in the product Release Notes. In this mode, you install the full-scale VTune Profiler product on the host system and install the [command line interface of the VTune Profiler](#), `vtune`, on the target system, which enables you to run native data collection directly on the target.

The following figure shows an overview of the remote analysis that is run with `vtune` directly on the target system:



In the native usage mode, workflow steps to configure and run analysis on a remote system are similar to the [remote collectors mode](#).

Native Sampling Collector (SEP) Usage Mode

Sampling collector (SEP) is a command-line tool for hardware event-based sampling analysis targeted for resource-restricted systems. The SEP package is delivered as part of the target package of the VTune Profiler. The SEP package contains both `sep` utilities and the `sepdk` source code (for `pax.ko` and `sep4_x.ko`) to build the sampling drivers.

To use SEP, extract the SEP package from the `vtune_profiler_target_sep_x86.tgz` or `vtune_profiler_target_sep_x86_64.tgz` file, build the driver and upload both driver and `sep` utilities to the target, and then collect the event-based sampling performance data in command line. See the *Sampling Enabling Product User's Guide* for more details.

NOTE

VTune Profiler also provides the `sepdk` sources for building sampling drivers. This source code could be same as the source code provided in the SEP package, if the VTune Profiler uses the same driver as SEP. VTune Profiler `sepdk` sources also include the event-based stack sampling data collector that is not part of the SEP package.

See Also

[Deploy your SSH Key for Intel® VTune™ Profiler](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

Set Up Linux* System for Remote Analysis

You can collect data remotely on a target Linux* system by specifying the system as the analysis target in Intel® VTune™ Profiler by selecting **Remote Linux (SSH)** in the **Where** pane when configuring an analysis. VTune Profiler provides an option to automatically install the appropriate collectors on the target system. Specify a location for the install using the **VTune Profiler installation directory on the remote system** field.

NOTE

The automatic installation on the remote Linux system does not build the sampling drivers although you can install the pre-built sampling drivers if you connect via password-less SSH as the root user. [Driverless sampling data collection](#) is based on the Linux Perf* tool functionality, which is available without Root access and has a limited scope of analysis options. To collect advanced hardware event-based sampling data, manually install the sampling driver or set up the password-less SSH connection with the Root user account.

1. Install the VTune Profiler collectors on the target system.
 - [Install the VTune Profiler collectors automatically](#).
 - If the collectors are not automatically installed or you get an error message after an automatic install attempt, use the following steps to manually prepare for data collection on a remote Linux system:
[Install the VTune Profiler collectors manually](#).
2. Build and install sampling drivers. (Optional).
3. Set up an SSH access to the target system.

Install the VTune Profiler Collectors Automatically

When you enter the connection parameters in the **Remote Linux* (SSH)** window of the **WHERE** pane, VTune Profiler checks for the presence of VTune Profiler collector package on the target system specified.

If an appropriate package was not located on the target system, VTune Profiler offers to deploy the package automatically.

WHERE **Remote Linux (SSH)** ▾ **VTune cannot detect remote machine configuration.****Retry****SSH destination**

root@10.125.98.125

**VTune Profiler installation directory on the remote system**

/tmp

Temporary directory on the remote system

/tmp

Press to deploy VTune Profiler target package to the remote system.

Deploy

Press the **Deploy** button to start the automatic collectors package deployment process.

If the collectors are not automatically installed or you get an error message after an automatic install attempt, you can install the collectors manually.

Install the VTune Profiler Collectors Manually

Use the following steps to set up analysis on a target regular or embedded Linux target system.

1. Copy the required target package archive to the target device using ftp, sftp, or scp. The following target packages are available on the host system where the VTune Profiler is installed:

- <install-dir>/target/linux/vtune_profiler_target_sep_x86.tgz - provides hardware event-based sampling collector only (SEP) for x86 systems
- <install-dir>/target/linux/vtune_profiler_target_sep_x86_64.tgz - provides hardware event-based sampling collector only (SEP) for 64-bit systems
- <install-dir>/target/linux/vtune_profiler_target_x86.tgz - provides all VTune Profiler collectors for x86 systems
- <install-dir>/target/linux/vtune_profiler_target_x86_64.tgz - provides all VTune Profiler collectors for 64-bit systems

NOTE

Use both *_x86 and *_x86_64 packages if you plan to run and analyze 32-bit processes on 64-bit systems.

2. On the target device, unpack the product package to the `/tmp` directory or another writable location on the system:

```
target> tar -zxvf <target_package>.tgz
```

VTune Profiler target package is located in the newly created directory `/tmp/vtune_profiler_<version>.<package_num>`.

When collecting data remotely, the VTune Profiler looks for the collectors on the target device in its default location: `/tmp/vtune_profiler_<version>.<package_num>`. It also temporary stores performance results on the target system in the `/tmp` directory. If you installed the target package to a different location or need to specify another temporary directory, make sure to configure your target properties in the **Configure Analysis** window as follows:

- Use the **VTune Profiler installation directory on the remote system** option to specify the path to the VTune Profiler on the remote system. If default location is used, the path is provided automatically.
- Use the **Temporary directory on the remote system** option to specify a non-default temporary directory.

Alternatively, use the `-target-install-dir` and `-target-tmp-dir` options from the `vtune` command line.

Build and Install the Drivers Manually

NOTE

Building the sampling drivers is only required if the drivers were not built as part of the collector installation. The installation output should inform you if building the sampling driver is required.

To enable [hardware event-based sampling analysis](#) on your target device:

1. Build the [sampling driver](#) on the target system.

NOTE

- Make sure kernel headers correspond to the kernel version running on the device. For details, see the `README.txt` file in the `sepdk/src` directory.
 - Make sure compiler version corresponds to the architecture (x86 or x86_64) of the kernel running on the target system.
 - For Hotspots in hardware event-based sampling mode, Microarchitecture Exploration, and Custom event-based sampling analysis types, you may not need root credentials and installing the sampling driver for systems with kernel 2.6.32 or higher, which exports CPU PMU programming details over `/sys/bus/event_source/devices/cpu/format` file system. Your operating system limits on the maximum amount of files opened by a process as well as maximum memory mapped to a process address space still apply and may affect profiling capabilities. These capabilities are based on Linux Perf* functionality and all its limitations fully apply to the VTune Profiler as well. For more information, see the *Tutorial: Troubleshooting and Tips* topic at https://perf.wiki.kernel.org/index.php/Main_Page.
-

2. On the target device, [install the drivers](#).

If the `insmod-sep` script does not work on the target system due to absence of standard Linux commands, you may install drivers manually using the Linux OS `insmod` command directly.

NOTE

To build the sampling driver as RPM using build services as Open Build Service (OBS), use the `sepdk.spec` file located at the `<install-dir>/sepdk/src` the directory.

See Also

[Set Up Remote Linux* Target](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

[Set Up Analysis Target](#)

Configure SSH Access for Remote Collection

To collect data on a remote Linux system, a password-less SSH connection is required.*

NOTE

A root connection is required to load the sampling drivers and to collect certain performance metrics. You (or your administrator) can configure the system using root permissions and then set up password-less SSH access for a non-root user if desired. For example, build and load the sampling drivers on the target system using root access and then connect to the system and run analysis as a non-root user. If you set up access without using the sampling drivers, then [driverless event-based sampling](#) can still be used.

Use one of the methods below to enable password-less SSH access:

- [Enable a password-less connection from Windows* to Linux*](#)
- [Manually configure a connection from macOS*/Linux to Linux](#)

NOTE

Versions of Intel® VTune™ Profiler older than 2019 Update 5 have a different configuration for password-less SSH. For legacy instructions, see this [article](#).

Enable a Password-less SSH Access from Windows to Linux

For Windows-to-Linux remote analysis, the VTune Profiler automatically configures a password-less access based on the public key identification.

1. Create a VTune Profiler project.
2. In the **Configure Analysis** window, select the **Remote Linux (SSH)** target system from the **WHERE** pane.
3. Specify your remote system in the **SSH destination** field as `user@target`; for example:
`root@172.16.254.1`.

VTune Profiler verifies your SSH connection and, if fails, it generates public/private keys required for enabling the password-less access and reports the results via an interactive terminal window.

4. When the public/private keys are generated, press any key to enter your credentials and let VTune Profiler *automatically* copy and apply the public/private keys.

Alternatively, you may press Ctrl-C to stop the automation. In this case, you need to *manually* add the already generated public/private keys from the paths specified in the terminal window to `~/.ssh/authorized_keys` on the remote system.

NOTE

VTune Profiler does not keep your credentials but uses them only once to enable the password-less access.

When the keys are applied, the terminal window closes and you can proceed with the project configuration and analysis. For all subsequent sessions, you will not be asked to provide credentials for remote accesses to the specified system.

Configure a Password-less SSH Access from Linux/macOS to Linux

For remote collection on a Linux target system, set up the password-less mode on the local Linux or macOS host as follows:

1. Generate the key with an empty passphrase:

```
host> ssh-keygen -t rsa
```

2. Copy the key to target system:

```
host> ssh-copy-id user@target
```

Alternatively, if you do not have `ssh-copy-id` on your host system, use the following command:

```
host> cat .ssh/id_rsa.pub | ssh user@target 'cat >> .ssh/authorized_keys'
```

3. Verify that a password is not required anymore, for example:

```
host> ssh user@target ls
```

Possible Issues

If the keys are copied but the VTune Profiler cannot connect to the remote system via SSH, make sure the permissions for `~/.ssh` and home directories, as well as SSH daemon configuration, are set properly.

Permissions

Make sure your `~/.ssh` and `~/.ssh/authorized_keys` directory permissions are not too open. Use the following commands:

```
chmod go-w ~/  
chmod 700 ~/.ssh  
chmod 600 ~/.ssh/authorized_keys
```

SSH Configuration

Check that the `/etc/ssh/sshd_config` file is properly configured for the public key authentication.

NOTE

For this step, you may need administrative privileges.

If present, make sure the following options are set to yes:

```
RSAAuthentication yes  
PubkeyAuthentication yes  
AuthorizedKeysFile .ssh/authorized_keys
```

For root remote connections, use:

```
PermitRootLogin yes
```

If the configuration has changed, save the file and restart the SSH service with:

```
sudo service ssh restart
sudo service sshd restart (on CentOS)
```

See Also

[Set Up Remote Linux* Target](#)

Search Directories for Remote Linux* Targets

For accurate module resolution and source analysis of your remote Linux application, make sure the Intel® VTune™ Profiler has access to your binary/symbol and source files on the host system.

If debug information is provided in separate files for your binaries, you need to specify search paths for these files on the host *when configuring a performance analysis*. If these files are not present on the host system, make sure to either copy them from the target system or mount the directory with these files. Then, add these locations to the search paths of the analysis configuration.

To add search paths, use any of the following options:

- From command line, use the `--search-dir`/`--source-search-dir` options. For example, from a Windows* host:

```
host>./vtune -target-system=ssh:user1@172.16.254.1 --collect hotspots -knob sampling-mode=hw -r
system_wide_r@@@ --search-dir C:\my_projects\symbols
```

- From GUI, use the [Binary/Symbol Search](#) and [Source Search](#) dialog boxes.

NOTE

The search is non-recursive. Make sure to specify all required directories.

When you run a remote analysis, the VTune Profiler launches your application on the remote target, collects data, copies all binary files to the host, and finalizes the analysis result. During [finalization](#), the VTune Profiler searches the directories for binary/symbol and source data in the following order:

1. Directory `<result dir>/all` (recursively).
2. Additional search directories that you defined for this project in the **Binary/Symbol Search/Source Search** dialog boxes or `--search-dir`/`--source-search-dir` command line options.
3. Absolute path on the remote target or VTune Profiler cache directory (binary files only).

See Also

[Set Up Remote Linux* Target](#)

[Search Directories](#)

[Specifying Search Directories](#)
from command line

[Debug Information for Linux* Application Binaries](#)

[Enable Linux* Kernel Analysis](#)

Temporary Directory for Performance Results on Linux* Targets

Configure a temporary directory for the remote or local data collection on Linux target systems.

When performing a [hardware event-based sampling collection](#) with the Intel® VTune™ Profiler or configuring the result directory for analysis on a mounted share, temporary data files are written to the system global temporary directory. Typically the global temporary directory is `/tmp`.

Depending on the length of the VTune Profiler analysis and data collected, significant temporary disk space may be required. The temporary data may exceed the current allocated or available global temporary storage space. If the system global temporary space is exceeded, the VTune Profiler analysis may fail with a warning similar to the following: **Warning: Cannot load data file `/home/user/r001hs/data.0/tbs0123456789.tb6' (tbrw call.....) failed: Invalid sample file (24)**. Note that the VTune Profiler temporary files may no longer be in the temporary storage location, giving you the false impression that there is plenty of space available. In this case, you may wish to check the temporary storage usage while the analysis is running. If the usage of system temporary storage reaches 100%, this may be the root cause of the error.

If the cause of the error is insufficient temporary disk space, you may set up an alternative temporary directory for collected data. VTune Profiler may still keep writing some scratch files of insignificant size (for example, the socket file `sep_ipc_socket_0`) to the system global temporary directory. However, it will utilize the defined alternative temporary location for the larger files such as those beginning with `lwp` (for example, `lwp28478_wallclock.tb7`, `lwp28478_user.mrk`, `lwp28478_7.txt`). When the VTune Profiler completes [finalization](#), all temporary scratch files are automatically removed.

Configuring an Alternative Temporary Directory for Local Targets

For local targets, you may set the standard Linux `TMPDIR` environment variable to an alternate directory path with the sufficient temporary storage space. To configure the `TMPDIR` environment variable, do the following:

1. From within the shell where you will be running the VTune Profiler command line or GUI, assign a value and export `TMPDIR`, for example:

```
> export TMPDIR=/directory_path/ tmp
```

2. Verify the assignment:

```
> echo $TMPDIR
```

3. Verify directory permissions are sufficient for the directory assigned to `TMPDIR`:

```
> ls -ld /directory_path/ tmp
```

4. From the shell window, run the VTune Profiler hardware event-based sampling collection using either the command line or GUI.

Configuring an Alternative Temporary Directory for Remote Targets

To change the temporary directory for remote targets from GUI, do the following:

1. Click the



Configure Analysis button.

2. Select the **remote Linux (SSH)** target system.
3. In the **Temporary directory on the remote system** field specify your alternative temporary directory.

To specify an alternative temporary directory from the command line, use the `target-tmp-dir` option, for example:

```
host>./vtune --target-system=ssh:vtune@10.125.21.170 -target-tmp-dir=/home/tmp -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -- /home/samples/matrix
```

See Also

[Set Up Analysis Target](#)

Embedded Linux* Targets

Use the Intel® VTune™ Profiler for performance analysis on Embedded Linux systems, Wind River*, Yocto Project*, FreeBSD* and others.*

Embedded device performance data can be collected remotely on the embedded device and running the analysis from an instance of VTune Profiler installed on the host system. This is useful when the target system is not capable of local data analysis (low performance, limited disk space, or lack of user interface control).

NOTE

Root access to the operating system kernel is required to install the collectors and drivers required for performance analysis using VTune Profiler.

To enable performance analysis on an embedded device, use any of the following:

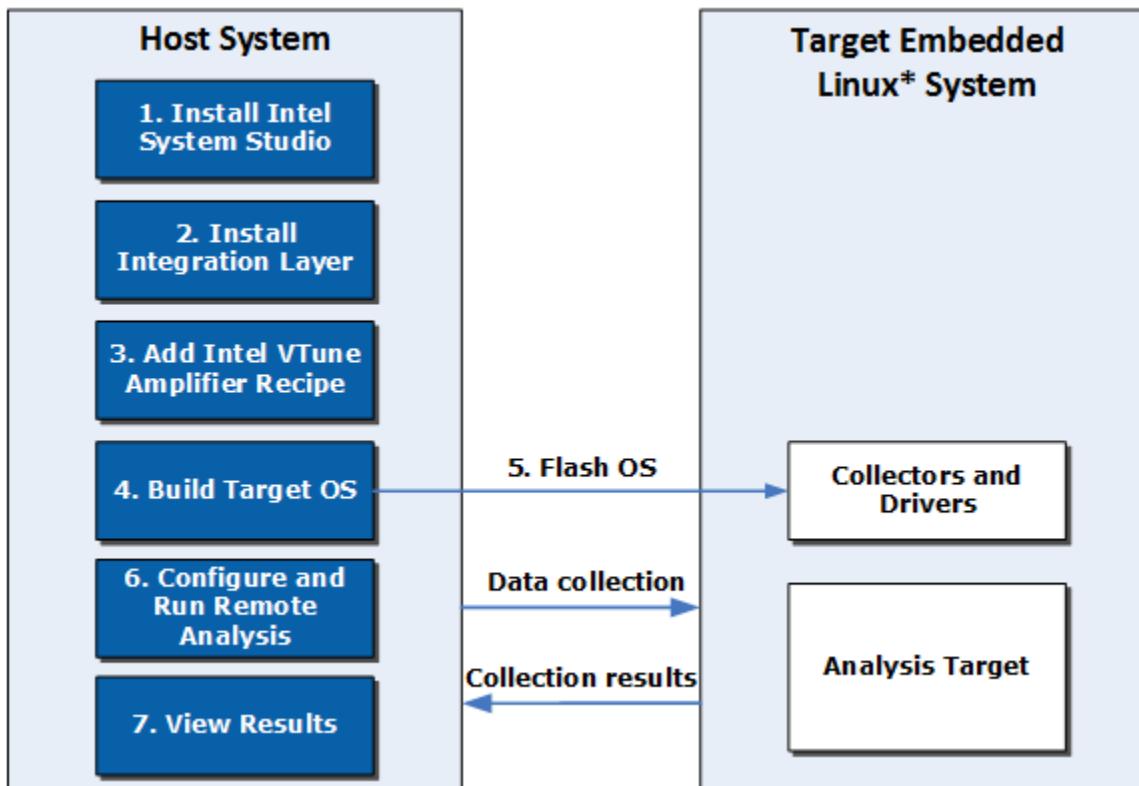
- [Intel System Studio integration layer](#) (Wind River* Linux and Yocto Project* only)
- [Intel VTune Profiler Yocto Project Integration Layer](#)
- bundled VTune Profiler installation packages

Use the Intel System Studio Integration Layer

NOTE

The Intel System Studio integration layer works for embedded systems with Wind River Linux or Yocto Project installed.

The Intel System Studio integration layer allows the Intel System Studio products to be fully integrated with a target operating system by building the drivers and corresponding target packages into the operating system image automatically. Use this option in the case where a platform build engineer has control over the kernel sources and signature files, but the application engineer does not. The platform build engineer can integrate the product drivers with the target package and include them in the embedded device image that is delivered to the application engineer.



1. Install Intel System Studio using the installer GUI.
2. Install the Intel System Studio integration layer.
 - a. Copy the integration layer from the Intel System Studio installation folder to the target operating system development folder.
 - b. Run the post-installation script: <iss-install-dir>/YoctoProject/meta-intel-iss/yp-setup/postinst_<OS>_iss.sh <ISS_BASE_dir>
For example, for Wind River Linux: /YoctoProject/meta-intel-iss/yp-setup/postinst_wr_iss.sh
3. Build the recipe that includes the appropriate VTune Profiler package.
 - a. Add the path to the /YoctoProject/meta-intel-iss to the bblayers.conf file:


```
BBLAYERS= "\n    ...\n    <OS_INSTALL_DIR>/YoctoProject/meta-intel-iss\n    ...\n"
```
 - b. Add the VTune Profiler recipes to conf/local.conf. Possible recipes include:
 - intel-vtune-drivers: integrates all VTune Profiler drivers for PMU-based analysis with stacks and context switches. Requires additional kernel options to be enabled.
 - intel-vtune-sep-driver: integrates drivers for PMU-based analysis with minimal requirements for kernel options.

For more information about these collection methods, see [Remote Linux Target Setup](#).
4. Build the target operating system, which will complete the integration of the VTune Profiler collectors and drivers.
5. Flash the operating system to the target embedded device.

After flashing the operating system to the target embedded device, ensure that the appropriate VTune Profiler drivers are present. For more information, see [Building the Sampling Drivers for Linux Targets](#).

6. Run the analysis on the target embedded device from the host system using an SSH connection or using the SEP commands.
 - a. Set up a password-less SSH access to the target using RSA keys.
 - b. Specify your target application and remote system.

NOTE

After configuring the remote connection, VTune Profiler will install the appropriate collectors on the target system.

- c. Choose an analysis type.
- d. Run the analysis from the host.

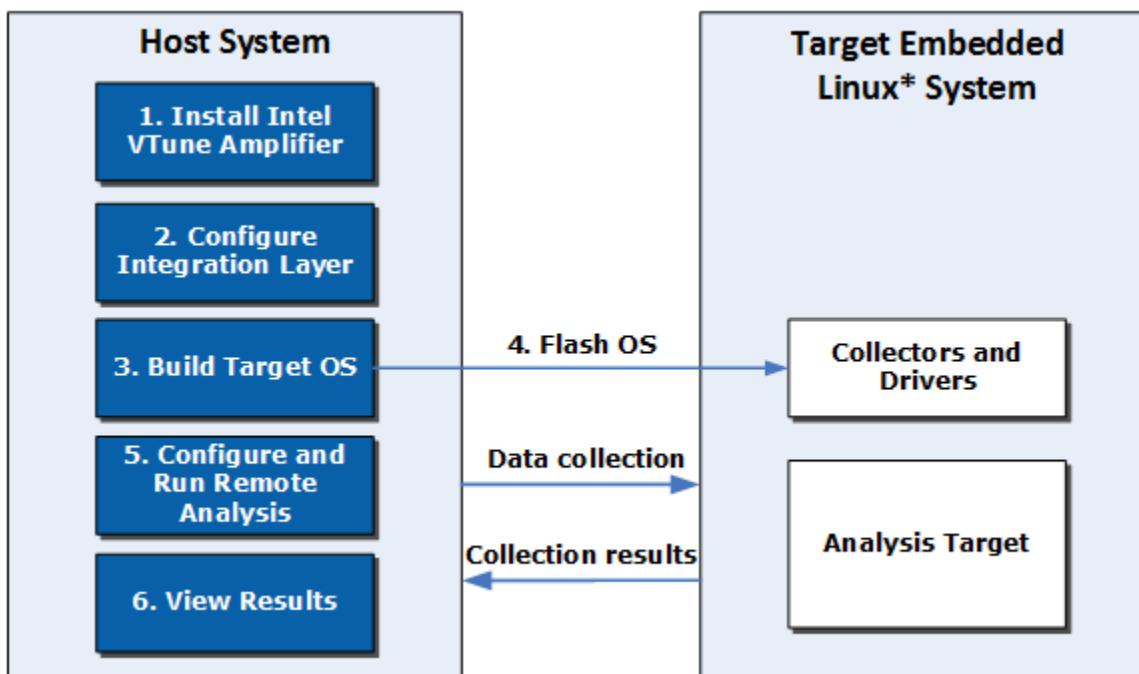
Use the information available in the *Sampling Enabling Product User's Guide* to run the SEP commands.

7. View results in the VTune Profiler GUI on the host.

Examples: [Configuring Yocto Project* with the Intel System Studio Integration Layer](#)

Use the Intel VTune Profiler Yocto Project Integration Layer

Intel VTune Profiler Yocto Project integration layer builds the drivers into the operating system image automatically. Use this option in the case where a platform build engineer has control over the kernel sources and signature files, but the application engineer does not. The platform build engineer can integrate the product drivers with the target package and include them in the embedded device image that is delivered to the application engineer.



1. Install Intel VTune Profiler.
2. Configure the integration layer.
 - a. Extract the `<install-dir>/target/linux/vtune_profiler_target_x86.tgz` or `<install-dir>/target/linux/vtune_profiler_target_x86_64.tgz` package.
 - b. Modify the `sepdk/vtune-layer/conf/user.conf` file to specify user settings.

- a. Specify one of the following paths:
 - Path to unzipped target package: VTUNE_TARGET_PACKAGE_DIR = "<PATH>"
 - Path to VTune Profiler installation directory: VTUNE_PROFILER_2020_DIR = "<PATH>"
- b. (Optional) To integrate the SEP driver during system boot, specify ADD_TO_INITD = "y".
- c. Copy the integration layer to the Yocto Project development environment.
- d. Add the path to the layer to the bblayers.conf file:

```
BBLAYERS= "\\\n...\\n<OS_INSTALL_DIR>/vtune-layer\\n...\\n"
```

- e. Add the VTune Profiler recipes to conf/local.conf. Possible recipes include:
 - intel-vtune-drivers: integrates all VTune Profiler drivers for PMU-based analysis with stacks and context switches. Requires additional kernel options to be enabled.
 - intel-vtune-sep-driver: integrates drivers for PMU-based analysis with minimal requirements for kernel options.

For more information about these collection methods, see [Remote Linux Target Setup](#).

3. Build the target operating system, which will complete the integration of the VTune Profiler collectors and drivers.
4. Flash the operating system to the target embedded device.
After flashing the operating system to the target embedded device, ensure that the appropriate VTune Profiler drivers are present.
5. Run the analysis on the target embedded device from the host system using an SSH connection or using the SEP commands.
 - a. [Set up a password-less SSH access](#) to the target using RSA keys.
 - b. Specify your target application and remote system.
 - c. Choose an analysis type.
 - d. Run the analysis from the host.

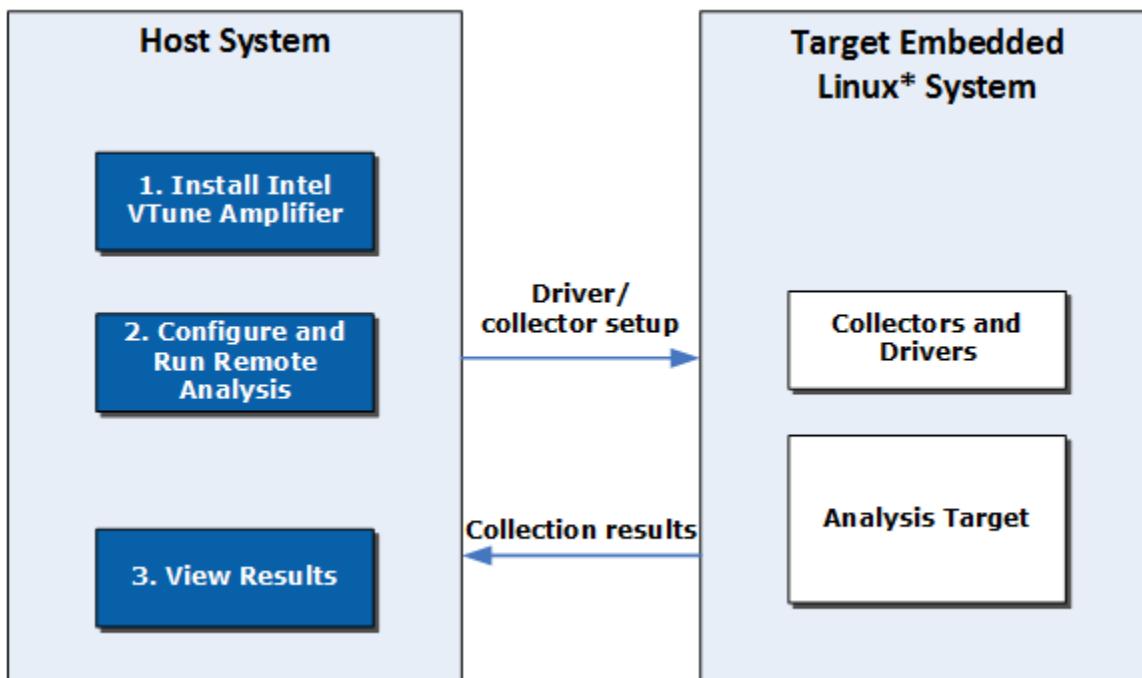
Use the information available in the *Sampling Enabling Product User's Guide* to run the SEP commands.

6. View results in the VTune Profiler GUI.

Example: [Configuring Yocto Project with the VTune Profiler Integration Layer](#)

Use the Bundled Intel VTune Profiler Installation Packages

You can build the appropriate drivers and install the VTune Profiler collectors on your kernel image manually with a command line. This option requires root access to the configured kernel source.



1. Install Intel VTune Profiler using the installer GUI.
2. Run the analysis on the target embedded device.
 - a. Set up a password-less SSH access for the root user to the target using RSA keys.
 - b. Specify your target application and remote system. The collectors and drivers within the package should be automatically installed.
 - c. Choose an analysis type.
 - d. Run the analysis from the host.
3. View results in the VTune Profiler GUI on the host.

Troubleshooting

If the drivers were not built during collector installation, the installation output should inform you that building the sampling driver is required.

The drivers are built either on the target system or on the host system, depending on compiler toolchain availability:

1. If the compiler toolchain is available on the target system:
 - a. On the target embedded device, build the driver from the <install-dir>/sepdk/src directory using the `./build-driver` command.
 - b. Load the driver into the kernel using the `./insmod-sep` command.
2. If the compiler toolchain is not available on the target system:
 - a. On the host system, cross-build the driver using the driver source from the target package sepdk/src directory with the `./build-driver` command. Provide the cross-compiler (if necessary) and the target kernel source tree for the build.
 - b. Copy the sepdk/src folder to the target system.
 - c. Load the driver into the kernel using the `./insmod-sep` command.

Example: Configuring Yocto Project with Intel VTune Profiler Target Packages

See Also

[Build and Install the Sampling Drivers for Linux* Targets](#)

Configure Yocto Project* and VTune Profiler with the Integration Layer

NOTE Profiling support for the Yocto Project* is deprecated and will be removed in a future release.

Intel® VTune™ Profiler can collect and analyze performance data on embedded Linux* devices running Yocto Project*. This topic provides an example of setting up the VTune Profiler to collect performance data on an embedded device with Yocto Project 1.8 installed using the Intel VTune Profiler integration layer provided with the product installation files. The process integrates the VTune Profiler product drivers with the target package and includes them in the embedded device image. Root access to the kernel is required.

NOTE

VTune Profiler is able to collect some performance data without installing the VTune Profiler drivers. To collect driverless event-based sampling data, installing the drivers and root access is not required. For [full capabilities](#), install the VTune Profiler drivers as described here.

Select the Target Package

VTune Profiler provides two Yocto Project recipes in the following packages:

- The `vtune_profiler_target_sep_x86_64.tgz` package includes the `intel-vtune-sep-driver` recipe, which enables performance data collection using hardware event-based sampling. Attempting to collect stacks when using this recipe will automatically switch to [driverless collection mode](#). This recipe has [minimal requirements](#) for Linux kernel configuration.
- The `vtune_profiler_target_x86_64.tgz` package includes the `intel-vtune-drivers` recipe, which enables the full performance data capabilities using hardware event-based sampling. This recipe has [additional requirements](#) for Linux kernel configuration. The `intel-vtune-drivers` recipe is a superset of the `intel-vtune-sep-driver` recipe.

Only one recipe can be used at a time. There is no difference between the `x86` and `x86_64` target packages for building recipes within Yocto Project. Both can be used on either 32 bit or 64 bit systems.

1. Download the VTune Profiler target package or locate the package in the `<install-dir>/target/linux` directory on the host system where VTune Profiler is installed.
2. Copy the selected target package to a location on the Yocto Project build system.

Prepare the Integration Layer

1. On the Yocto Project build system, extract the `vtune_profiler_target_sep_x86_64.tgz` or `vtune_profiler_target_x86_64.tgz` archive to a writeable location.

```
cd $HOME
tar xvzf vtune_profiler_target_x86_64.tgz
```

2. (Optional) Modify the `$HOME/vtune_profiler_<version>/sepdk/vtune-layer/conf/user.conf` file to specify user settings.

- a. If the VTune Profiler recipe has been split from the target package, specify one of the following paths:

- Path to unzipped target package: `VTUNE_TARGET_PACKAGE_DIR = "$HOME/vtune_profiler_<version>"`
- Path to VTune Profiler: `VTUNE_PROFILER_2020_DIR = "/opt/intel/vtune_profiler"`

- b. To integrate the SEP driver during system boot:

Specify `ADD_TO_INITD = "y"` for init-based Yocto systems;

Or specify `ADD_TO_SYSTEMD = "y"` for systemd-based Yocto systems.

- 3.** In the Yocto Project development environment, add the path to the layer to the `bblayer.conf` file. For example:

```
vi conf/bblayers.conf
BBLAYERS = "$HOME/vtune_profiler_<version>/sepdk/vtune-layer\"
```

Your file should look similar to the following:

```
BBLAYERS ?= " \
$HOME/source/poky/meta \
$HOME/source/poky/meta-poky \
$HOME/source/poky/meta-yocto-bsp \
$HOME/source/poky/meta-intel \
$HOME/vtune_profiler/sepdk/vtune-layer \
"
```

- 4.** Specify the Intel VTune Profiler recipe in `conf/local.conf`. In this example, the `intel-vtune-drivers` is used.

```
vi "conf/local.conf"
IMAGE_INSTALL_append = " intel-vtune-drivers"
```

NOTE

You cannot add both `intel-vtune-drivers` and `intel-vtune-sep-driver` at the same time.

Build and Flash the Target Operating System

- 1.** Build the target operating system. For example:

```
bitbake core-image-sato
```

NOTE

If you modified the kernel configuration options, make sure the kernel is recompiled.

- 2.** Flash the operating system to the embedded device.

Configure and Run Remote Analysis

Use the following steps on the host system to set up and launch the analysis on the embedded device:

1. Set up a password-less SSH access to the target using RSA keys.
2. Create a new project.
3. Select the **remote Linux (SSH)** analysis system and specify the collection details.
4. Configure the analysis type.
5. Start the analysis.

See Also

[Embedded Linux* Targets](#)

[Configure Yocto Project* and Intel® VTune™ Profiler with the Linux* Target Package](#)

[Configure Yocto Project*/Wind River* Linux* and Intel® VTune™ Profiler with the Intel System Studio Integration Layer](#)

Configure Yocto Project*/Wind River* Linux* and Intel® VTune™ Profiler with the Intel System Studio Integration Layer

NOTE Profiling support for the Yocto Project* is deprecated and will be removed in a future release.

You can use Intel® VTune™ Profiler to collect and analyze performance data on embedded Linux* devices running Yocto Project* or Wind River* Linux*. This example describes how you set up VTune Profiler using the Intel System Studio integration layer, to collect performance data on an embedded device with Yocto Project 1.8 or Wind River* Linux* installed. The integration layer is available with the product installation files. The process integrates the VTune Profiler product drivers with the target package and includes them in the embedded device image. For this example, you need root access to the kernel.

Install the Intel System Studio Integration Layer

Prerequisite: Install Intel System Studio on the host system.

1. Copy the integration layer from the Intel System Studio installation folder to the appropriate development folder.

For Yocto Project*:

```
cp -r <ISS_BASE_DIR>/YoctoProject/meta-intel-iss <YOCTO_HOME>/
```

For Wind River* Linux*:

```
cp -r <ISS_BASE_DIR>/YoctoProject/meta-intel-iss <WR_HOME>/
```

where

- <ISS_BASE_DIR> : Root folder of the Intel System Studio installation. By default, this is /opt/intel/system_studio_<version>.x.y/. For example, for the 2019 version, the root folder is /opt/intel/system_studio_2019.0.0/.
- <YOCTO_HOME> : Root folder of the Yocto Project* cloned directory.
- <WR_HOME> : Root folder of the Wind River* Linux* cloned directory.

2. Register the layer by running the post-installation script.

For Yocto Project*:

In the shell console, go to the <YOCTO_HOME> folder and run this command::

```
$ meta-intel-iss/yp-setup/postinst_yp_iss.sh <ISS_BASE_DIR>
```

For Wind River* Linux*:

In the shell console, go to the <WR_HOME> folder and run this command::

```
$ meta-intel-iss/yp-setup/postinst_wr_iss.sh <ISS_BASE_DIR>
```

To uninstall the Intel System Studio integration:

1. Run the appropriate script to uninstall:

For Yocto Project*:

In the shell console, go to the <YOCTO_HOME> folder and run this command::

```
$ meta-intel-iss/yp-setup/uninst_yp_iss.sh
```

For Wind River* Linux*:

In the shell console, go to the <WR_HOME> folder and run this command::

```
$ meta-intel-iss/yp-setup/uninst_wr_iss.sh
```

2. Remove the meta-intel-iss layer.

Add the Intel VTune Profiler Recipe

1. Add the path to the wr-iss-<version> to the bblayer.conf file. For example:

```
vi /path/to/poky-fido-10.0.0/build/conf/bblayers.conf
BBLAYERS = "$HOME/source/poky/wr-iss-2019\"
```

Your file should look similar to the following:

```
BBLAYERS ?= " \
$HOME/source/poky/meta \
$HOME/source/poky/meta-poky \
$HOME/source/poky/meta-yocto-bsp \
$HOME/source/poky/meta-intel \
$HOME/source/poky/wr-iss-2019 \
"
```

2. Add the Intel VTune Profiler recipe to conf/local.conf. Two recipes are available,

intel-vtune-drivers and intel-vtune-sep-driver. In this example, the intel-vtune-drivers is used so the analysis can be run from the VTune Profiler GUI on the host system.

```
vi "conf/local.conf"
IMAGE_INSTALL_append = " intel-vtune-drivers"
```

NOTE

You cannot add both intel-vtune-drivers and intel-vtune-sep-driver at the same time.

Build and Flash the Target Operating System

1. Build the target operating system. For example:

```
bitbake core-image-sato
```

2. Flash the operating system to the embedded device.

Configure and Run Remote Analysis

Use the following steps on the host system to set up and launch the analysis on the embedded device:

1. Set up a password-less SSH access to the target using RSA keys.
2. Create a new project.
3. Select the **remote Linux (SSH)** analysis system and specify the collection details.
4. Configure the analysis type.
5. Start the analysis.

See Also

[Embedded Linux* Targets](#)

[Configure Yocto Project*/Wind River* Linux* and Intel® VTune™ Profiler with the Intel System Studio Integration Layer](#)

[Configure Yocto Project* and Intel® VTune™ Profiler with the Linux* Target Package](#)

Configure Yocto Project* and Intel® VTune™ Profiler with the Linux* Target Package

NOTE Profiling support for the Yocto Project* is deprecated and will be removed in a future release.

Intel® VTune™ Profiler can collect and analyze performance data on embedded Linux* devices. This topic provides an example of setting up Intel VTune Profiler to collect performance data on an embedded device running Yocto Project*. The first section provides information for a typical use case where the required collectors are automatically installed. The second section provides steps to manually install the collectors and the VTune Profiler drivers for hardware event-based sampling data collection.

Automatically Configure and Run Remote Analysis

Use the following steps on the host system to set up and launch the analysis on the embedded device:

1. [Set up a password-less SSH access](#) to the target using RSA keys.
2. Open VTune Profiler and create a new project.
3. Select the **remote Linux (SSH)** analysis target and specify the collection details. VTune Profiler connects to the target system and installs the appropriate collectors. If the automatic installation fails or if you want to collect hardware event-based sampling with the VTune Profiler drivers, follow the instructions below to manually configure the target system.
4. Select the [analysis type](#).
5. Start the analysis.

Manually Configure the Linux Target System

Use these steps only if the automatic installation fails.

1. Copy the target package archive to the target device. The following target packages are available:
 - <install-dir>/target/vtune_profiler_target_sep_x86.tgz - provides hardware event-based sampling collector only (SEP) for x86 systems
 - <install-dir>/target/vtune_profiler_target_x86.tgz - provides all VTune Profiler collectors for x86 systems
 - <install-dir>/target/vtune_profiler_target_sep_x86_64.tgz - provides hardware event-based sampling collector only (SEP) for 64-bit systems
 - <install-dir>/target/vtune_profiler_target_x86_64.tgz - provides all VTune Profiler collectors for 64-bit systems

For example, the following command copies the `vtune_profiler_target_x86_64.tgz` package to the embedded device using SCP:

```
scp -r vtune_profiler_target_x86_64.tgz root@123.45.67.89:/opt/intel/
```

2. Extract the file on the target system. For example:

```
tar -xvsf vtune_profiler_target_x86_64.tgz
```

3. Make sure the sampling driver is available on the target system. The installation output should inform you if building the sampling driver is required. If it is not, you will need to build the sampling driver and install it on the target system.

If the compiler toolchain is available on the target embedded system, build the driver on the target device using the following steps:

- a. Open a command prompt and navigate to the <install-dir>/sepdk/src directory. For example:

```
cd /opt/intel/vtune_profiler_2020.0.0.0/sepdk/src
```

- b.** Build the driver using the `./build-driver` command. For example:

```
./build-driver -ni \
  --kernel-src-dir=/usr/src/kernel/ \
  --kernel-version=4.4.3-yocto-standard \
  --make-args="PLATFORM=x64 ARITY=smp"
```

- c.** Load the driver into the kernel using the `./insmod-sep` command.

If the compiler toolchain is not available on the target embedded system, build the driver on the host system and install it on the target device using the following steps:

- a.** Open a command prompt and navigate to the `<install-dir>/sepdk/src` directory. For example:

```
cd /opt/intel/vtune_profiler_2020.0.0.0/sepdk/src
```

- b.** Cross-build the driver using the `./build-driver` command. Provide the cross-compiler (if necessary) and the target kernel source tree for the build. For example:

```
mkdir drivers
./build-driver -ni \
  --c-compiler=i586-i586-xxx-linux-gcc \
  --kernel-version=4.4.3-yocto-standard \
  --kernel-src-dir=/usr/src/kernel/ \
  --make-args="PLATFORM=x32 ARITY=smp" \
  --install-dir=./drivers
```

- c.** Copy the `sepdk/src/drivers` folder to the target system.

- d.** Load the driver into the kernel using the `./insmod-sep` command.

See Also

Embedded Linux* Targets

[Configure Yocto Project*/Wind River* Linux* and Intel® VTune™ Profiler with the Intel System Studio Integration Layer](#)

FreeBSD* Targets

Intel® VTune™ Profiler allows you to collect performance data on a FreeBSD target system.*

Intel VTune Profiler is not installed on the FreeBSD target system. Instead, you are able to install VTune Profiler on a Linux*, Windows*, or macOS* host system and use a target package for collecting event-based sampling data on a remote FreeBSD target system in one of the following ways:

- Using VTune Profiler's automated remote collection capability (command line or user interface)
- Collecting the results locally on the FreeBSD system and copying them to the host system for viewing with VTune Profiler (command line only)

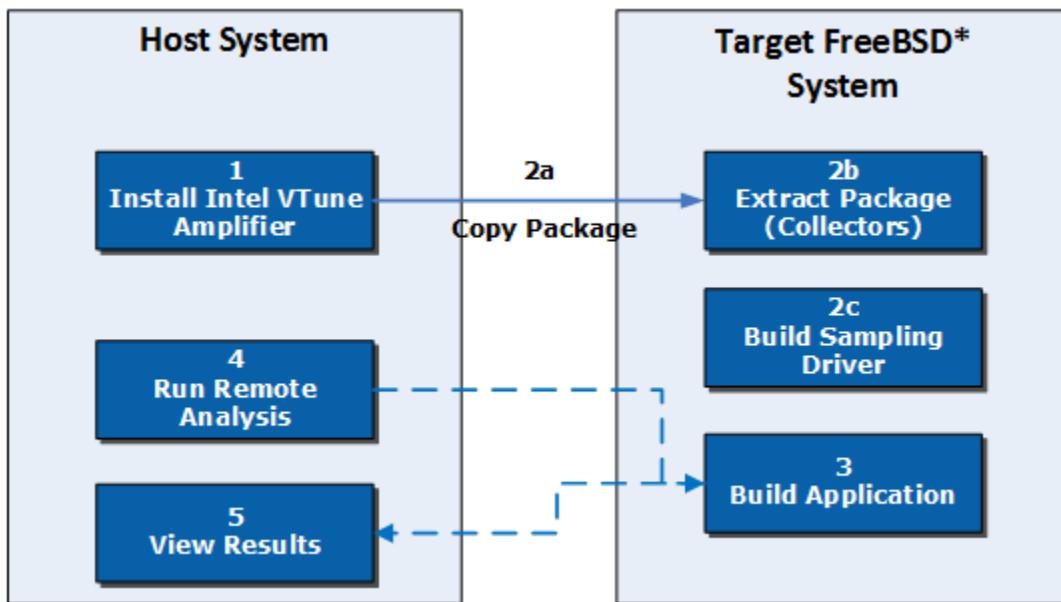
The following sections explain these options in more detail.

Supported Features

Remote Collection	Local Collection
Collection from Linux, Windows, or macOS host system using the Intel VTune Profiler GUI or command line (<code>vtune</code>)	Collection from the FreeBSD system using: <ul style="list-style-type: none"> • Intel VTune Profiler command line (<code>vtune</code>) • Sampling enabling product (SEP) collectors
Analysis Types:	Analysis types (VTune Profiler command line only): <ul style="list-style-type: none"> • hotspots

Remote Collection	Local Collection
<ul style="list-style-type: none"> • Hotspots (hardware event-based sampling mode) • Microarchitecture Exploration • Memory Access (without heap object allocation tracking) • Input and Output (with hardware event-based metrics and SPDK analysis; without MMIO accesses and DPDK analysis) • Custom Analysis 	<ul style="list-style-type: none"> • uarch-exploration • memory-access • io (with hardware event-based metrics and SPDK analysis; without MMIO accesses and DPDK analysis) • custom event-based sampling analysis
View results on host system	View results in VTune Profiler on a Linux, Windows, or macOS host system

Remote Collection from Host System



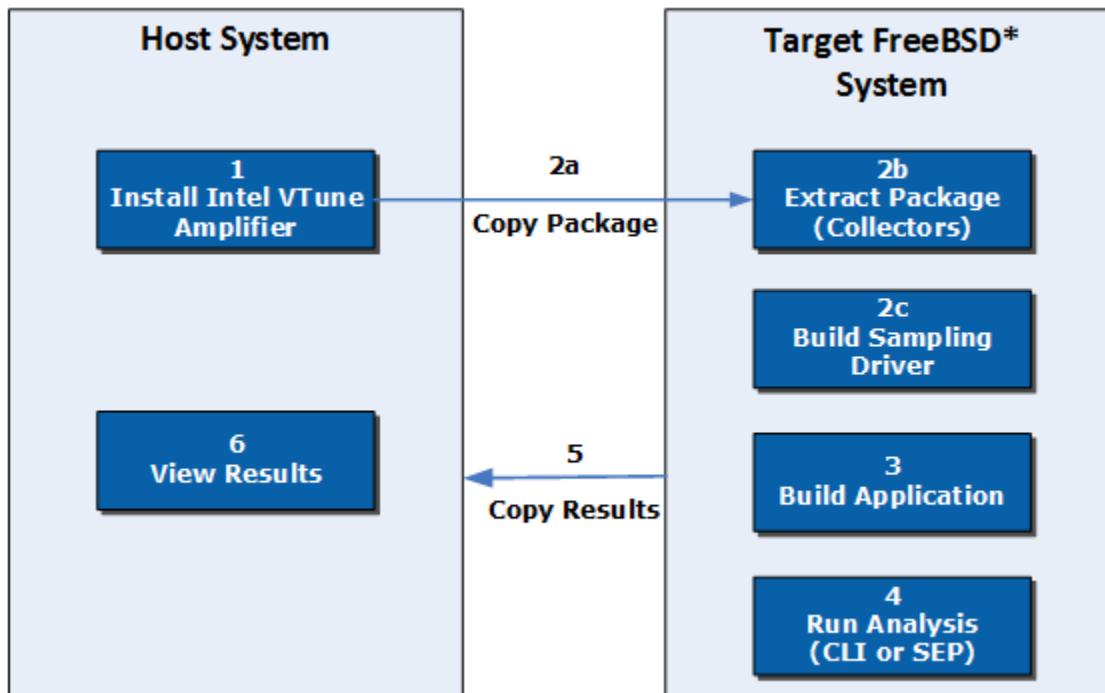
1. Install VTune Profiler on your Linux*, Windows*, or macOS* host. Refer to the [Installation Guide](#) for your host system for detailed instructions.
2. Install the appropriate sampling drivers on the FreeBSD target system. For more information, see [FreeBSD* System Setup](#).
3. [Optional] If you want to collect performance data with stacks, build your FreeBSD target application using the `-fno-omit-frame-pointer` compiler option, to allow the sampling collector to determine the call chain via frame pointer analysis.
4. Collect performance data using remote analysis from the host system from the VTune Profiler command line or GUI.
 - a. Create or open a project.
 - b. Specify your target application and remote system and make sure to specify search directories for [symbol/source files](#) required for finalization on the host.
 - c. Choose and configure an analysis type.

Supported VTune Profiler analysis types (event-based sampling analysis only) include:

- [Hotspots](#) (hardware event-based sampling mode)
- [Microarchitecture Exploration](#)

- [Memory Access](#) (without heap object allocation tracking)
 - [Input and Output](#) (with hardware event-based metrics and SPDK analysis; without MMIO accesses and DPDK analysis)
 - [Custom Analysis](#)
- d. Run the analysis from the host. Depending on your settings, the application launches and runs automatically. Once collection is finished, the result is finalized and displayed with the **Summary** window open.
5. Review the results on the host system.

Native Collection on FreeBSD System



1. Install VTune Profiler on your Linux*, Windows*, or macOS* host. Refer to the [Installation Guide](#) for your host system for detailed instructions.
2. Install the appropriate sampling drivers on the FreeBSD target system. For more information, see [FreeBSD* System Setup](#).
3. [Optional] If you want to collect performance data with stacks, build your FreeBSD target application using the `-fno-omit-frame-pointer` compiler option, which allows the sampling collector to determine the call chain via frame pointer analysis.
4. Collect performance data using one of the following methods. For more information about each of these methods, see [Remote Linux Target Setup](#).
 - Native analysis on the target system using the VTune Profiler command line (`vtune`). Supported analysis types include: [hotspots](#), [uarch-exploration](#), [memory-access](#), [io](#) or [custom](#) event-based sampling analysis.
 - Native analysis on the target system using the sampling enabling product (SEP) collectors. For more information, see the [Sampling Enabling Product User Guide](#).
5. Copy the results to the host system.
6. Review the results with VTune Profiler.
 - If you used the `vtune` command, open the `*.vtune` file.
 - If you collected SEP data, import the `*.tb7` file.

See Also

[Introduction](#)

[Set Up Remote Linux* Target](#)

[Set Up FreeBSD* System](#)

Set Up FreeBSD* System

Intel® VTune™ Profiler allows you to collect performance data remotely on a FreeBSD target system.*

Intel® VTune™ Profiler includes a target package for collecting event-based sampling data on a FreeBSD* target system either via the remote collection capability or by collecting the results locally on the FreeBSD system and copying them to a Linux*, Windows*, or macOS* host system. The collected data is then displayed on a host system that supports the graphical interface.

1. Install VTune Profiler on your Linux, Windows, or macOS host. Refer to the [Installation Guide](#) for your host system for detailed instructions.
2. Install the appropriate sampling drivers on the FreeBSD target system. Use the <vtune-install-dir>/target/freebsd/vtune_profiler_target_x86_64.tgz file for analysis using VTune Profiler or the <vtune-install-dir>/target/freebsd/vtune_profiler_target_sep_x86_64.tgz file for analysis using the sampling enabling product (SEP) collectors.
3. Collect performance data using one of the following methods. For more information about each of these methods, see [FreeBSD* Targets](#) and [Remote Linux Target Setup](#).
 - Remote analysis from the host system using the VTune Profiler command line or GUI.
 - Native analysis on the target system using the VTune Profiler command line.
 - Native analysis on the target system using the SEP collectors.
4. Review the results on the host system.

Install the Sampling Drivers on FreeBSD

Use the following steps to configure your FreeBSD target system for event-based sampling analysis. Root privileges are required on the target system to install the VTune Profiler drivers.

1. Copy the <vtune-install-dir>/target/freebsd/vtune_profiler_target_x86_64.tgz file to the target system using FTP, SFTP, or SCP.
2. Extract the archive to the /opt/intel directory on the target system.
3. Navigate to the following location: /opt/intel/sepdk/modules
4. Run the following commands to build the appropriate drivers:

```
$ make  
$ make install
```

5. Run the following command to install the drivers:

```
$ kldload sep pax
```

Allow non-root users to run an event-based sampling analysis by running the following commands after installing the drivers:

```
$ chgrp -R <user_group> /dev/pax  
$ chgrp -R <user_group> /dev/sep
```

Remove the Sampling Drivers from FreeBSD

Run the following command to unload the sampling drivers:

```
$ kldunload sep pax
```

See Also

[FreeBSD* Targets](#)

Set Up Remote Linux* Target

QNX* Targets

Intel® VTune™ Profiler supports collecting performance data on QNX target systems.*

Data collection is possible via command line interface from a host system running Windows* or Linux* to the target QNX system. The collected traces are transferred to the host system via ethernet and stored for review. After collection, the performance results can be imported and viewed in the Intel VTune Profiler user interface.

The target collector can be integrated into the target QNX image during the image build process and requires only 1 MB of space on the target file system. Because the traces are transferred to the host system, collection can be done on target systems with limited storage capacity or with read-only file systems.

1. Prerequisites
2. Set up your system
3. Run analysis
4. View and interpret results

Prerequisites

- Host System: Linux* or Windows* system with QNX BSP and VTune Profiler installed
- Target System: Supported processor with QNX7 operating with instrumental kernel, connected to the host system via ethernet. Supported processors include Intel® Pentium®, Intel® Celeron®, or Intel Atom® processors formerly code named Apollo Lake or Intel Atom® processors formerly code named Denverton.
- Turn off firewall restrictions for network connections between the host system and target system

Set up Your System

Complete the following steps on your host and target system to install collectors and enable performance analysis using Intel VTune Profiler:

1. Ensure that the host system is connected to the target QNX system via ethernet and log in to the target QNX system using a command window.
2. Make the <install-dir>/target/qnx_x86_64/bin64/sep file on the host system available on the target QNX system by copying, mounting a network share, or integrating it into the target image.
3. On the host system, launch the VTune Profiler user interface, click **New Project**, specify a project name, and click **Create Project**.
4. Click **Configure Analysis**, select **local host** in the **WHERE** pane, and click **Search Binaries**.
5. In the **Binary/Symbol Search** window, browse to the location of the kernel and application target modules on the host system, and click **OK**.

Run Analysis

Analysis is run using collectors previously installed on the target QNX system and a command invoked on the host Windows or Linux system. All result files are saved to the host system.

1. On the target QNX system, run the following command: <sep-dir>/sep
Where <sep-dir> is the location where the sep file was copied. The target collector loads and waits for the host system to connect.
2. On the host system, run one of the following analysis commands.
 - **Hotspots with call stacks:** <install-dir>/bin64/sep -start -d <duration> -target-ip <target-ip-address> -target-port 9321 -lbr call_stack -out <filename>.tb7

Example command:

```
/opt/intel/vtune_profiler/bin64/sep -start -d 60 -target-ip 12.345.67.89 -target-port 9321 -lbr
call_stack -out hotspots_callstacks.tb7
```

NOTE

Call stacks are hardware based and limited to a depth of 16 frames. Due to hardware limitations, the depth of the captured call stack can be less than 16 frames.

- **Custom CPU events:** <install-dir>/bin64/sep.exe -start -d <duration> -target-ip <target-ip-address> -target-port 9321 -ec "<event-list>" -out <filename>.tb7

Example command:

```
/opt/intel/vtune_profiler/bin64/sep.exe -start -d 60 -target-ip 12.345.67.89 -target-port 9321 -
ec "MEM_LOAD_UOPS_RETIREDR.DRAM_HIT, MEM_LOAD_UOPS_RETIREDR.HITM, MEM_LOAD_UOPS_RETIREDR.L2_HIT" -out
custom.tb7
```

See the [Sampling Enabling Product User's Guide](#) for more information.

3. After collection begins, run the application on the target QNX system or ensure that it is already running. The analysis collects system-wide data. Collection stops automatically when the specified duration is complete.
4. After collection is complete, stop the application on the target QNX system if it is not already finished.

View and Interpret Results

After collection is complete, the *.tb7 result file is available on the host system.

1. On the host system, [import](#) the *.tb7 file into the previously created project.
2. Switch to the **Hotspots** viewpoint and review the performance data collected.
 - If you collected hotspots data, begin with the **Summary** window in the **Hotspots** viewpoint. The **Top Hotspots** list shows the top 5 functions that occupied the most CPU time. Double-click a function to be taken to the **Bottom-up** window where you can see aggregated performance data and a timeline showing activity over the entire collection. For more information, see [Hotspots View](#).
 - If you collected CPU event data, begin with the **Microarchitecture Exploration** viewpoint. For more information, see [Microarchitecture Exploration View](#).

See Also

[Cookbook: Profiling Operating System Boot Time on Linux* and QNX*](#)

Managed Code Targets

Enable performance analysis of Java, .NET*, Python*, Go* or Windows* Store targets by configuring the managed code profiling options.*

To configure the managed code analysis:

1. Click the



Configure Analysis button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From the **WHERE** pane, select a required target system (for example, **local host**).
3. From the **WHAT** pane, select a target type (for example, **Launch Application**).
4. Expand the **Advanced** section and configure the **Managed code profiling mode** by choosing one of the following options:

- **Native** mode collects data on native code only, does not attribute data to managed source.
- **Managed** mode collects everything, resolves samples attributed to native code, attributes data to managed source only. The call stack in the analysis result displays data for managed code only.
- **Mixed** mode collects everything and attributes data to managed source where appropriate. Consider using this option when analyzing a native executable that makes calls to the managed code.
- **Auto** mode automatically detects the type of target executable, managed or native, and switches to the corresponding mode.

NOTE

- On Windows* OS, the managed code profiling setting is inherited automatically from the Visual Studio* project. For native targets, the **Managed code profiling mode** option is disabled.
 - System-wide profiling for managed code is not supported on Windows* OS.
 - **Managed** and **Mixed** modes are not supported on Linux* OS.
-

See Also

[.NET* Targets](#)

[Windows Store Application Targets](#)

[Go* Application Targets](#)

[Java* Code Analysis](#)

[Python* Code Analysis](#)

[Set up Analysis Target](#)

[mrte-mode](#)

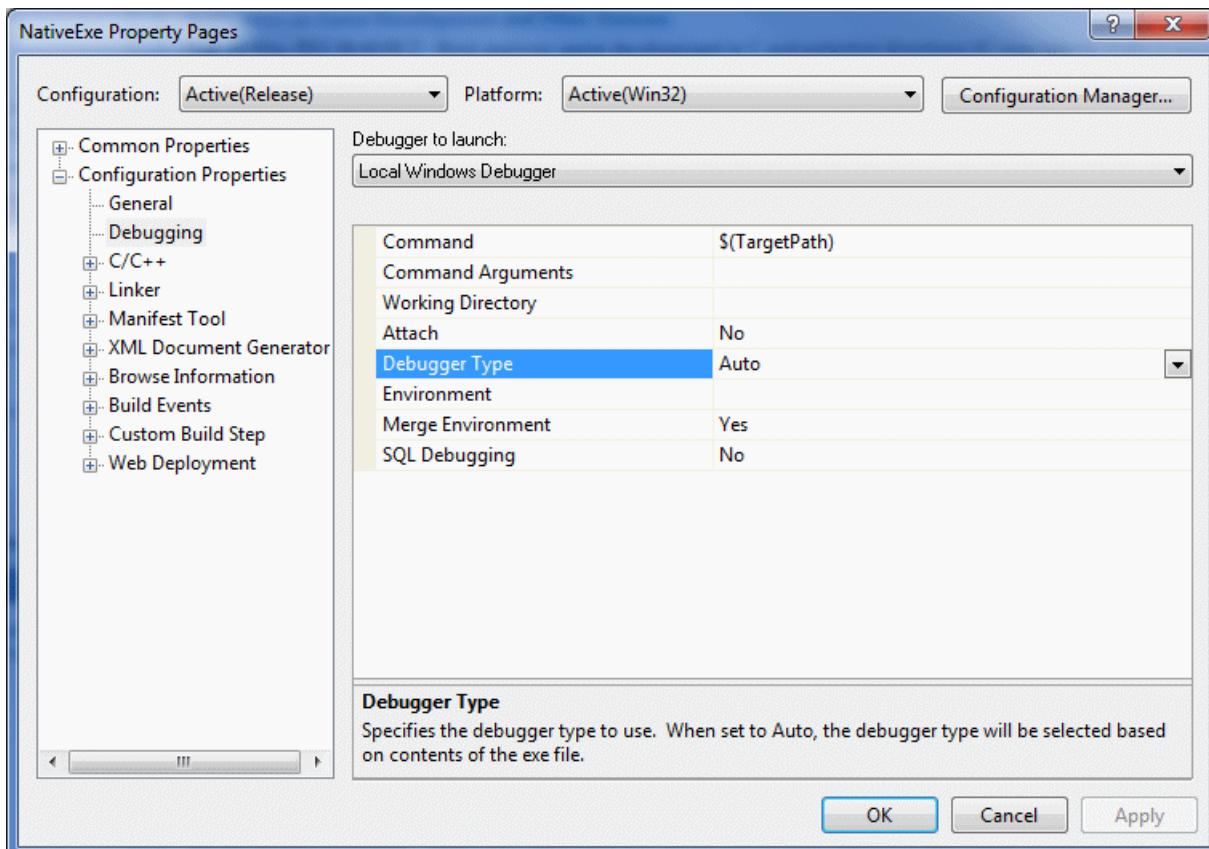
[vtune option](#)

[Java* Code Analysis from Command Line](#)

.NET* Targets

Explore performance analysis specifics for pure .NET applications or native applications with .NET calls.*

Intel® VTune™ Profiler automatically identifies the type of the code based on the debugger type specified in the Visual Studio project property pages:



VTune Profiler inherits this setting to set the profiling mode for the analysis target. The following types are possible:

- **Native** mode collects data on native code only, does not attribute data to managed source.
- **Managed** mode collects everything, resolves samples attributed to native code, attributes data to managed source only. The call stack in the analysis result displays data for managed code only.
- **Mixed** collects everything and attributes data to managed source where appropriate. Consider using this option when analyzing a native executable that makes calls to the managed code.
- **Auto** mode automatically detects the type of target executable, managed or native, and switches to the corresponding mode.

Profiling Pure .NET Applications

If you analyze a pure .NET application, the VTune Profiler resolves the **Auto** mode to **Mixed**.

Before profiling a pure .NET application, make sure to generate debug information for a native image of .NET managed assembly, which is required for successful [module resolution](#) and source analysis:

1. Use the Native Image Generator tool (Ngen.exe) from the .NET Framework to generate a native .pdb file.
2. Click the



Configure Analysis button on the toolbar.

3. In the **Configure Analysis** window, click the



Search Binaries button at the bottom.

-
4. In the **Binary/Symbol Search** dialog box, add a path to the generated native .pdb file.

Profiling Native Applications with .NET Calls

If you analyze a native application that calls managed code, the VTune Profiler resolves the **Auto** mode to **Native** and does not profile managed code. In this case, if you want to enable the VTune Profiler to profile the managed code called from the native application, set the profiling mode to **Mixed** as follows:

1. Click the



Configure Analysis button on the toolbar.

The **Configure Analysis** window opens.

2. De-select the **Inherit settings from Visual Studio* project** check box.

The **Managed code profiling mode** option is enabled.

3. In the **WHAT** pane, from the **Advanced > Managed code profiling mode** menu, select the required profiling mode.

NOTE

- System-wide profiling is not supported for managed code.
- Starting with the VTune Amplifier 2018 Update 2, you can use the [Hotspots](#) analysis in the hardware event-based sampling mode (former Advanced Hotspots) to profile .Net Core applications running on Linux* or Windows* systems in the **Launch Application** mode. For the product versions prior to 2018 Update 2, make sure to manually [configure CoreCRL environment variables](#) to enable the Advanced Hotspots analysis.

See Also

[Problem: Analysis of the .NET* Application Fails](#)

[mrte-mode](#)
vtune option

Windows Store Application Targets

Intel® VTune™ Profiler supports a [hardware event-based sampling analysis](#) for Windows Store C/C++, C# and JavaScript applications running via the **Attach to Process** and **Profile System** modes. The **Launch Application** mode is not supported.

Before analysis make sure you have [administrative privileges](#) to run the data collection.

Support Limitations for Windows Store C# Application Analysis

Starting from Microsoft Windows 8*, all Windows Store C# applications are automatically pre-compiled with the NGEN service during each 24 hours. VTune Profiler cannot resolve Native Image methods since symbol information for these methods is absent. As a result, when you profile a pre-compiled application with the VTune Profiler, you have [unknown] function entries instead of C# methods. You can either generate .pdb files for native images via the `Ngen.exe` tool or temporarily workaround this problem until the next automatic NGEN pre-compilation:

1. Locate automatically pre-compiled assemblies. Typically 32-bit assemblies are located in `C:\Users\<Administrator>\AppData\Local\Packages\<package>\AC\Microsoft\CLR_v4.0_32\NativeImages\` and 64-bit assemblies are located in `C:\Users\Administrator\AppData\Local\Packages\<package>\AC\Microsoft\CLR_v4.0_32\NativeImages\` folders.

NOTE

<package> varies with applications. To identify the package, use any of the following options:

- Open the **Task Manager** and check the properties for your application. The **General** tab contains the package value including the version that should be omitted. For example, if the **General** tab displays 47828<app_name>_1.0.0.4_neutral__sgvg9sxsmibt4, then NGEN'ed modules are located in C:\Users\Administrator\AppData\Local\Packages \47828<app_name>_sgvg9sxsmibt4\AC\Microsoft\CLR_v4.0_32\NativeImages\.
- Use the **Process Explorer** tool: explore the list of modules loaded in the application, find *.ni.exe modules and get their location.

-
2. Rename the folders that include *.ni.dll or *.ni.exe. For example, rename C:\Users\ Administrator\AppData\Local\Packages\47828<app_name>_sgvg9sxsmibt4\AC\Microsoft\CLR_v4.0_32\NativeImages\<app_name> to C:\Users\Administrator\AppData\Local\Packages\47828<app_name>_sgvg9sxsmibt4\AC\Microsoft\CLR_v4.0_32\NativeImages\<app_name>.
 3. Re-start your application.

CLR JIT-compiles the methods. You can use the VTune Profiler to profile your C# application until the next automatic NGEN pre-compilation.

NOTE

This workaround is not recommended for .NET* Framework libraries (for example, mscorlib.dll).

Support Limitations for Windows Store JavaScript Application Analysis

VTune Profiler supports mapping to the source file for JavaScript modules. But when you dive to the source from the grid or Timeline pane, the VTune Profiler does not locate the most performance-critical code line by default but opens the first line of the function in the Source pane. Use the [navigation buttons](#) to switch between hot code lines.

See Also

[Set Up Analysis Target](#)

Go* Application Targets

Use the Intel® VTune™ Profiler to analyze Go applications using the hardware event-based sampling data collection.*

Prerequisites: When configuring your analysis target, use the [Search Sources](#) button to specify paths for your application source files so that the VTune Profiler can resolve the functions and display statistics per source line.

VTune Profiler supports Go applications profiling with the following analysis types:

- [Hotspots](#) (hardware event-based sampling mode)
- [Microarchitecture Exploration](#)
- [Custom Analysis](#)

Limitations

- Only Go applications compiled with a compiler version 1.6 and later are supported.
- Only 64-bit version of Go applications is supported.
- On Windows* OS, call stack collection is not supported.

See Also

[Get Started with Intel® VTune™ Profiler](#)

[Hardware Event-based Sampling Collection](#)

[Set Up Analysis Target](#)

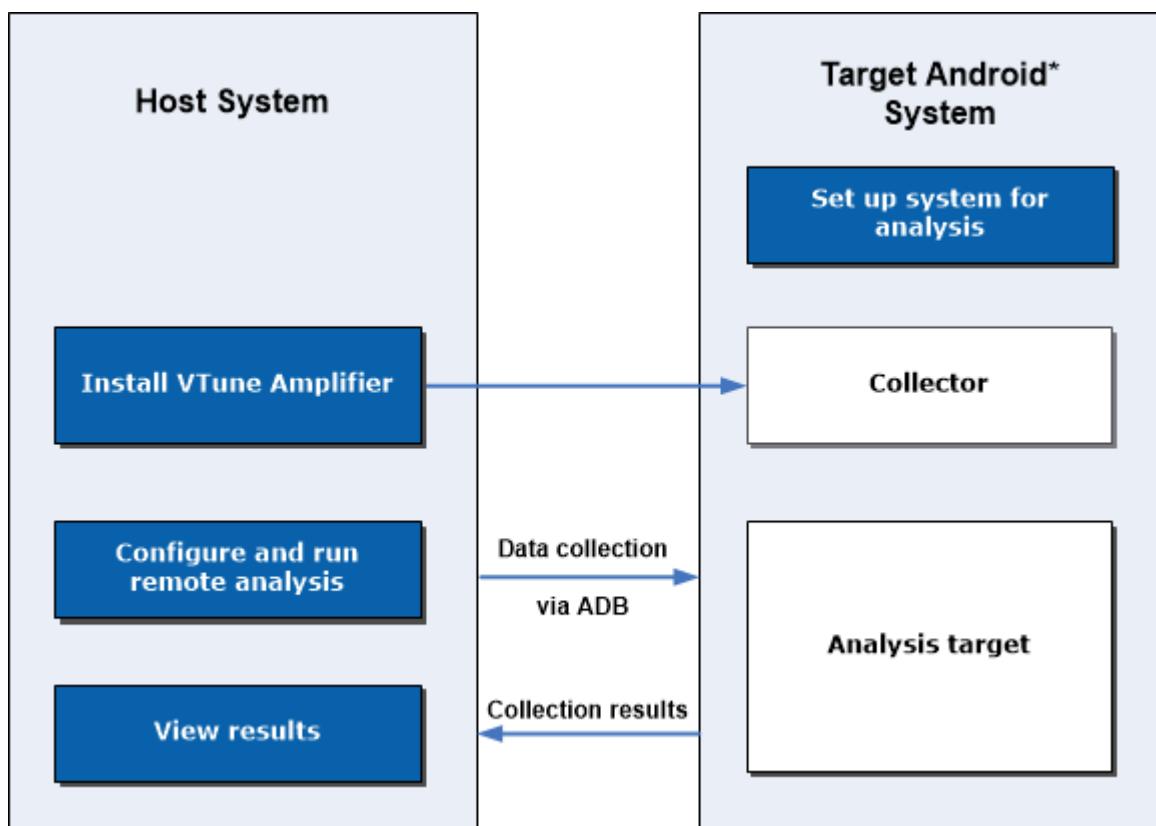
Android* Targets

Use the Intel® VTune™ Profiler installed on the Windows*, Linux,* or macOS* host to analyze code performance on a remote Android* system.

NOTE

For successful product operation, the target Android system should have ~25 MB disk space.

VTune Profiler supports the following usage mode with VTune Profiler remote collector and ADB communication:



1. Install VTune Profiler

Install the full-scale VTune Profiler product on the host system. By default, the VTune Profiler also installs the remote collector on the target Android system as soon as you run the first remote collection.

NOTE

If the remote VTune Profiler collector is installed on a non-rooted device, during installation you may get an error message on missing/incorrect drivers. You can dismiss this message if you plan to run the user-mode sampling and tracing collection (Hotspots) only.

2. Prepare your target system for analysis

- Configure your Android device for analysis.
 - Gain adb access through TCP/IP to an Android device.
 - To enable hardware-event-based sampling analysis or Java* analysis, gain root mode adb access to the Android device.
-

NOTE

Depending on your system configuration, you may not need to gain a root mode access for Hotspots (hardware event-based sampling mode), Microarchitecture Exploration and Custom EBS analysis types.

- To enable hardware-event-based sampling analysis, verify that version compatible pre-installed signed drivers are on the target Android system.

3. Configure and run remote analysis

1. Prepare your Android application for analysis.

Tip

Use ITT APIs to control performance data collection by adding basic instrumentation to your application.

2. Specify your analysis target and remote system.

NOTE

You may use the **Analyze unplugged device** option to exclude the ADB connection and power supply impact on the performance results. In this case, the collection starts as soon as you disconnect the device from the USB cable or a network. The analysis results are transferred to the host when you plug in the device back.

3. Optionally, [specify binary and source search directories](#).
4. Choose an analysis type.

NOTE

On Android platforms, the VTune Profiler supports hardware event-based sampling analysis types and Hotspots analysis in the user-mode sampling mode. Other algorithmic analysis types are not supported.

5. Configure the analysis type.
6. Run the analysis from the host.

4. View collected data

View the collected data on the host.

NOTE

To run [Energy analysis](#) on an Android system, use the Intel® SoC Watch tool.

See Also

[Set Up Android* System](#)

[Android* Target Analysis from the Command Line](#)

[Manage Data Views](#)

Build and Install Sampling Drivers for Android* Targets

On some versions of Android systems, including most of the Intel® supplied reference builds for SDVs, the [required drivers](#) are pre-installed in `/lib/modules` or `/system/lib/modules`. If the drivers are not pre-installed in any of these directories, you need to build them manually from the command line. Optionally, you can get the drivers integrated into the Android build so that they are built and installed when the operating system is built.

Android requires signed drivers. Every time the Android kernel is built, a random private/public key is generated. Drivers must be signed with the random private key to be loaded. The drivers (`socperf2_x.ko`, `pax.ko`, `sep4_x.ko`, and `vtsspp.ko`) must be signed with the same key and be compiled against the same kernel headers/sources as what is installed on the Android target system.

VTune Profiler has options for building a new driver on the Linux host system and installing it on a target Android system. This is not the default and will only work if you provide the proper kernel headers/sources and a signing key. For example, the VTune Profiler uses the `--with-drivers` option for building PMU drivers and `--kernel-src-dir` option for providing the configured kernel headers/sources tree path.

To build the sampling drivers on the host Linux system, enter:

```
<install-dir>/bin{32,64}/vtune-androidreg.sh --package-command=build --with-drivers --kernel-src-dir=/ path/to/configured/kernel/sources [--jitztuneinfo=jit|src|dex|none]
```

To install the sampling drivers from the Linux host, enter:

```
<install-dir>/bin{32,64}/vtune-androidreg.sh --package-command=install --with-drivers --kernel-src-dir=/ path/to/configured/kernel/sources [--jitztuneinfo=jit|src|dex|none]
```

To sign the drivers after the drivers are built:

Typically the VTune Profiler automatically signs drivers if kernel sources with the keys are available when it builds the drivers. Otherwise, to manually sign the drivers, use the following command:

```
$KERNEL_SRC/source/scripts/sign-file CONFIG_MODULE_SIG_HASH $KERNEL_SRC/ \
signing_key.priv $KERNEL_SRC/signing_key.x509 driver.ko
```

where the `CONFIG_MODULE_SIG_HASH` value is extracted from the `$KERNEL_SRC/.config` file.

NOTE

You need the "exact" signing key that was produced at the time and on the system where your kernel was built for your target.

See Also

[Sampling Drivers](#)

[Install Intel® VTune™ Profiler](#)

Prepare an Android* Application for Analysis

Set Up Android* System

Set Up Android* System

When using the VTune Profiler to collect data remotely on a target Android device, make sure to:

- [Configure your Android device for analysis.](#)
- [Gain adb access to an Android device.](#)
- For hardware event-based sampling, [gain a root mode adb access to the Android device.](#)
- [Use the pre-installed drivers on the target Android system.](#)

Optionally, do the following:

- [Enable Java* analysis.](#)
- To view functions within Android-supplied system libraries, device drivers, or the kernel, get access from the host development system to the exact version of these binaries with symbols not stripped.
- To view sources within Android-supplied system libraries, device drivers, or the kernel, get access from the host development system to the sources for these components.

Configure an Android Device for Analysis

To configure your Android device, do the following:

1. Allow Debug connections to enable adb access:
 - a. Select **Settings > About <device>**.
 - b. Tap **Build number** seven times to enable the **Developer Options** tab.
 - c. Select the **Settings > Developer Options** and enable the **USB debugging** option.

NOTE

Path to the **Developer Options** may vary depending on the manufacture of your device and system version.

2. Enable **Unknown Sources** to install the VTune Profiler Android package without Google* Play. To do this, select **Settings > Security** and enable the **Unknown Sources** option.

Gain ADB Access to an Android Device

VTune Profiler collector for Android requires connectivity to the Android device via adb. Typically Android devices are connected to the host via USB. If it is difficult or impossible to get adb access to a device over USB, you may get adb over Ethernet or WiFi. To connect ADB over Ethernet or WiFi, first connect to Ethernet or connect to a WiFi access point and then do the following:

1. Find the IP Address of the target. The IP address is available in Android for Ethernet via **Settings>Wireless&Networks>Ethernet>IP Address** or for Wi-Fi via **Settings>Wireless&Networks>Wi-Fi><Connected Access Point>>IP Address**.
2. Make sure adb is enabled on the target device. If not enabled, go to Terminal App (of your choice) on the device and type:

```
> su  
> setprop service.adb.tcp.port 5555  
> stop adbd  
> start adbd
```

3. Connect adb on the host to the remote device. In the Command Prompt or the Terminal on the host, type:

```
> adb connect <IPAddres>:5555
```

Gain a Root Mode ADB Access to the Android Device

For performance analysis on Android platforms, you typically need a root mode adb access to your device to:

- Install and load drivers needed for hardware event-based sampling.
- Enable the Android device to support Java* analysis.
- Run hardware event-based sampling analysis.

NOTE

There are several analysis types on Android systems that do NOT require root privileges such as [Hotspots Analysis](#) (user-mode samplingmode) and Perf*-based driverless sampling event-based collection.

Depending on the build, you gain root mode adb access differently:

- **User/Production builds** : Gaining root mode adb access to a user build of the Android OS is difficult and different for various devices. Contact your manufacturer for how to do this.
- **Engineering builds** : Root-mode adb access is the default for engineering builds. Engineering builds of the Android OS are by their nature not "optimized". Using the VTune Profiler against an engineering build is likely to result in VTune Profiler identifying code to optimize which is already optimized in user and userdebug builds.
- **Userdebug builds** : Userdebug builds of the Android OS offer a compromise between good results and easy-to-run tools. By default, userdebug builds run adb in user mode. VTune Profiler tools require root mode access to the device, which you can gain via typing `adb root` on the host. These instructions are based on userdebug builds.

Use the Pre-installed Drivers on the Target Android System

For hardware event-based sampling analysis, the VTune Profiler needs sampling drivers to be installed. On some versions of Android systems, including most of the Intel supplied reference builds for SDVs, the following drivers are pre-installed in `/lib/modules` or `/system/lib/modules` :

- Hardware event-based analysis collectors:
 - `socperf2_x.ko`
 - `pax.ko`
 - `sep3_x.ko`
 - `sep4_x.ko`
 - `vtsspp.ko`

Typically having pre-installed drivers is more convenient. You can check for pre-installed drivers by typing:

```
adb shell ls [/lib/modules|/system/lib/modules]
```

If the drivers are not available or the version does not match requirements, consider [building and installing the drivers](#).

See Also

[Prepare an Android* Application for Analysis](#)

[Cookbook: Profiling Hardware Without Sampling Drivers](#)

Enable Java* Analysis on Android* System

Explore configuration settings required to enable Java analysis with Intel® VTune™ Profiler on an Android system:

- [Enable Java analysis on rooted devices](#)

- Enabling Java analysis for code generated with ART* compiler

Enabling Java Analysis on Rooted Devices

By default, the VTune Profiler installs the remote collector on the target rooted Android devices with the `--jitvtuneinfo=src` option. To change the Java profiling option for rooted devices, you need to re-install the remote collector on the target manually using the `--jitvtuneinfo=[jit|src|dex|none]` option on `amplxe-androidreg.bat` (Windows) or `amplxe-androidreg.sh` (Linux). For example:

On Windows*:

```
<install-dir>\bin32\amplxe-androidreg.bat --package-command=install --jitvtuneinfo=src
```

On Linux*:

```
<install-dir>/bin{32,64}/amplxe-androidreg.sh --package-command=install --jitvtuneinfo=src
```

VTune Profiler updates the `/data/local.prop` file as follows:

- Basic information about the compiled trace: `root@android:/ # cat /data/local.prop dalvik.vm.extra-opts=-Xjitvtuneinfo:jit`
- Mapping from JIT code to Java source code and basic information about the compiled trace: `root@android:/ # cat /data/local.prop dalvik.vm.extra-opts=-Xjitvtuneinfo:src`
- Mapping from JIT code to DEX code and basic information about the compiled trace: `root@android:/ # cat /data/local.prop dalvik.vm.extra-opts=-Xjitvtuneinfo:dex`
- JIT data collection. By default, JIT collection is disabled if you do not supply any options: `root@android:/ # cat /data/local.prop dalvik.vm.extra-opts=-Xjitvtuneinfo:none`

Additionally, if your Dalvik JVM supports instruction scheduling, disable it by adding `-Xnoscheduling` at the end of `dalvik.vm.extra-opts`. For example:

```
root@android:/ # cat /data/local.prop dalvik.vm.extra-opts=-Xjitvtuneinfo:src -Xnoscheduling
```

NOTE

Java analysis currently requires an instrumented Dalvik JVM. Android systems running on the 4th Generation Intel® Core™ processors or Android systems using ART vs. Dalvik for Java are not instrumented to support JIT profiling. You do not need to specify `--jitvtuneinfo=N`.

Tip

If you are able to see the `--generate-debug-info` option in the logcat output (`adb logcat *:S dex2oat:I`), the compiler uses this option.

Enabling Java Analysis for Code Generated with ART* Compiler

To enable a source-level analysis, the VTune Profiler requires debug information for the analyzed binary files. By default, the ART compiler does not generate the debug information for Java code. Depending on your usage scenario, you may choose how to enable generating the debug information with the ART compiler:

NOTE

For releases prior to Android 6.0 Marshmallow*, the `--generate-debug-info` in the examples below should be replaced with `--include-debug-symbols`.

To Do This:	Do This:
Profile a 3rd party application or system application installed as an .apk file	<p>1. Set the system property <code>dalvik.vm.dex2oat-flags</code> to <code>--generate-debug-info</code>:</p> <pre>adb shell setprop dalvik.vm.dex2oat-flags --generate-debug-info</pre> <p>2. If you use <code>--compiler-filter=interpret-only</code>, set the optimization level to speed:</p> <pre>adb shell setprop dalvik.vm.dex2oat-filter speed</pre> <p>3. (Re-)install the application.</p> <pre>adb shell install -r TheApp.apk</pre>
Profile all applications installed as .apk or .jar files by re-building the Android image when pre-optimization for private applications is enabled (<code>LOCAL_DEX_PRE_OPT:=true</code> property set in <code>device.mk</code>)	<p>1. On your host system, open the <code>/build/core/dex_preopt_libart.mk</code> file, located in your Android OS directory structure.</p> <p>2. Modify the <code>--no-generate-debug-info</code> line to <code>--generate-debug-info</code> and save and close the file.</p> <p>3. Rebuild the Android image and flash it to your device.</p> <p>4. If you are using an Android image that is not PIC configured (<code>(WITH_DEXPREOPT_PIC:=false</code> property set in <code>device.mk</code>), generate <code>classes.dex</code> from <code>odex</code> using the <code>patchoot</code> command. <code>classes.dex</code> should appear in <code>/data/dalvik-cache/x86/</code> <code>system@app@appname@appname.apk@classes.dex</code></p>
Profile all applications installed as .apk or .jar files by re-building the Android image when pre-optimization for private applications is disabled (<code>LOCAL_DEX_PRE_OPT:=false</code> property set in <code>device.mk</code>)	<p>1. Set the system property <code>dalvik.vm.dex2oat-flags</code> to <code>--generate-debug-info</code>:</p> <pre>adb shell rm -rf /data/dalvik-cache/x86/ system@app@webview@webview.apk@classes.dex adb shell setprop dalvik.vm.dex2oat-flags --generate-debug-info</pre> <p>2. Stop and start the adb shell:</p> <pre>adb shell stop adb shell start</pre> <p>3. Generate the dex file:</p> <pre>adb shell ls /data/dalvik-cache/x86/ system@app@webview@webview.apk@classes.dex adb pull /data/dalvik-cache/x86/system@app@webview@webview.apk@classes.dex</pre>
Profile an application executed by the <code>dalvikvm</code> executable	Add the compiler option <code>--generate-debug-info</code> followed by <code>-Xcompiler-option</code> . Make sure the application has not been compiled yet. <pre>rm -f /data/dalvik-cache/*/*TheApp.jar* adb shell dalvikvm -Xcompiler-option --include-debug-symbols -cp TheApp.jar</pre>

To Do This:	Do This:
<p>Profile system and core classes</p> <p>NOTE This action is required if Java core classes get compiled to the /data/dalvik-cache/ subdirectory. Manufacturers may place them in different directories. If manufacturers supply the precompiled boot.oat file in /system/framework/x86, Java core classes will not be resolved because they cannot be recompiled with debug information.</p>	<p>Set the system property dalvik.vm.image-dex2oat-flags to --generate-debug-info and force recompilation:</p> <pre>adb shell stop adb shell rm -f /data/dalvik-cache/*/* adb shell setprop dalvik.vm.dex2oat-flags --generate-debug-info adb shell setprop dalvik.vm.image-dex2oat-flags --generate-debug-info adb shell start</pre> <p>If you run the application before the system classes are compiled, you should add another compiler option -Ximage-compiler-option --generate-debug-info:</p> <pre>adb shell rm -f /data/dalvik-cache/*/* adb shell dalvikvm -Xcompiler-option --generate-debug-info -Ximage-compiler-option --generate-debug-info -cp TheApp.jar</pre>

See Also

[Prepare an Android* Application for Analysis](#)

Prepare an Android* Application for Analysis

Before starting an analysis with the VTune Profiler, make sure your Android application is compiled with required settings:

Compilation Settings

Performance analysis is only useful on binaries that have been optimized and have symbols to attribute samples to source code. To achieve that:

- Compile your code with release level settings (for example, do not use the /O0 setting on GCC*).
- Do not set APP_OPTIM to debug in your Application.mk as this setting disables optimization (it uses /O0) when the compiler builds your binary.
- To run performance analysis (Hotspots) on non-rooted devices, make sure to compile your code setting the debuggable attribute to true in AndroidManifest.xml.

NOTE

If your application is debuggable (`android:debuggable="true"`), the default setting will be `debug` instead of `release`. Make sure to override this by setting `APP_OPTIM` to `release`.

By default, the Android NDK build process for Android applications using JNI creates a version of your `.so` files with symbols.

The binaries with symbols included go to `[ApplicationProjectDir]/obj/local/x86`.

The stripped binaries installed on the target Android system via the `.apk` file go to `[ApplicationProjectDir]/libs/x86`. These versions of the binaries cannot be used to find source in the VTune Profiler. However, you may collect data on the target system with these stripped binaries and then later use the binaries with symbols to do analysis (as long as it is an exact match).

When the VTune Profiler finishes collecting the data, it copies `.so` files from the device (which have had their symbols stripped). This allows the very basic functionality of associating samples to assembly code.

Tip

Use ITT APIs to control performance data collection by adding basic instrumentation to your application.

See Also

[Android* Target Analysis from the Command Line](#)

[Instrumentation and Tracing Technology APIs](#)

Analyze Unplugged Devices

Configure the Intel® VTune™ Profiler to run the collection on a detached device by using the **Analyze unplugged device** option.

Intel VTune Profiler allows you to run an analysis on a mobile system that is detached from the network or USB drive during the collection. Detaching the device from an ADB connection allows for increased accuracy in certain system performance metrics, such as power consumption. Unplugged analysis is currently supported for Android* target devices.

1. Connect to the target Android device via ADB. For more information, see [Android* System Setup](#).
2. Click the **Configure Analysis** button on the VTune Profiler toolbar.
3. Select the **Android device (ADB)** option from the **WHERE** pane.
4. In the **WHAT** pane, select the analysis target.
5. Expand the **Advanced** section of the **WHAT** pane and select the **Analyze unplugged device** option.
6. Select the analysis type from the **HOW** pane and click **Start**.
7. Unplug the device from the USB.

- Collection begins as soon as the device is disconnected. Data is collected on the device using the settings selected. An alert appears after collection completes. You can also tap the **Stop** button on your device to stop the collection.
8. Reconnect the device to the USB when collection completes. The collected results are automatically transferred to Intel VTune Profiler, processed, and displayed in the viewpoint appropriate to the analysis type selected. If you plug in the device before collection completes, the collection stops and the results are transferred to Intel VTune Profiler.

See Also

[-no-unplugged-mode](#)
[vtune](#) option

Search Directories for Android* Targets

For accurate module resolution and source analysis of your Android* application, make sure to specify search paths for binary and source files when configuring performance analysis:

- from command line, use the `--search-dir/--source-search-dir` options; for example:

```
host>./vtune --collect hotspots -knob sampling-mode=hw -r system_wide_r@@@ --search-dir ~/AndroidOS_repo/out/target/product/ctp_pr1/symbols/
```

- from GUI, use the [Dialog Box: Binary/Symbol Search](#) and [Dialog Box: Source Search](#) dialog boxes

If you have not set the project search directories at the time of collection or import, you will not be able to open the source code. Only Assembly view will be available for source analysis.

Consider the following when adding search paths:

- By default, the VTune Profiler pulls many binaries from the target device.
- The Kernel [vmlinux] is one file that does not contain symbols on the target device. Typically it is located in [AndroidOSBuildDir]/out/target/product/[your target]/linux/kernel/vmlinux.
- Many operating system binaries with symbols are located in either [AndroidOSBuildDir]/out/target/product/[your target]/symbols, or [AndroidOSBuildDir]/out/target/product/[your target]/obj.
- Application binaries with symbols are located in [AndroidAppBuildDir]/obj/local/x86.
- Application source files for the C/C++ modules are usually located in [AndroidAppBuildDir]/jni, not in [AndroidAppBuildDir]/src (where the Java *source files are). Some third-party software in Android does not provide binaries with symbols. You must contact the third party to get a version of the binaries with symbols.
- You can see if a binary has symbols by using the `file` command in Linux and make sure that it says `not stripped`.

```
file MyBinary.ext
MyBinary.ext: ELF 32-bit LSB shared object, Intel 80386, version 1
(SYSV), dynamically linked, not stripped
```

See Also

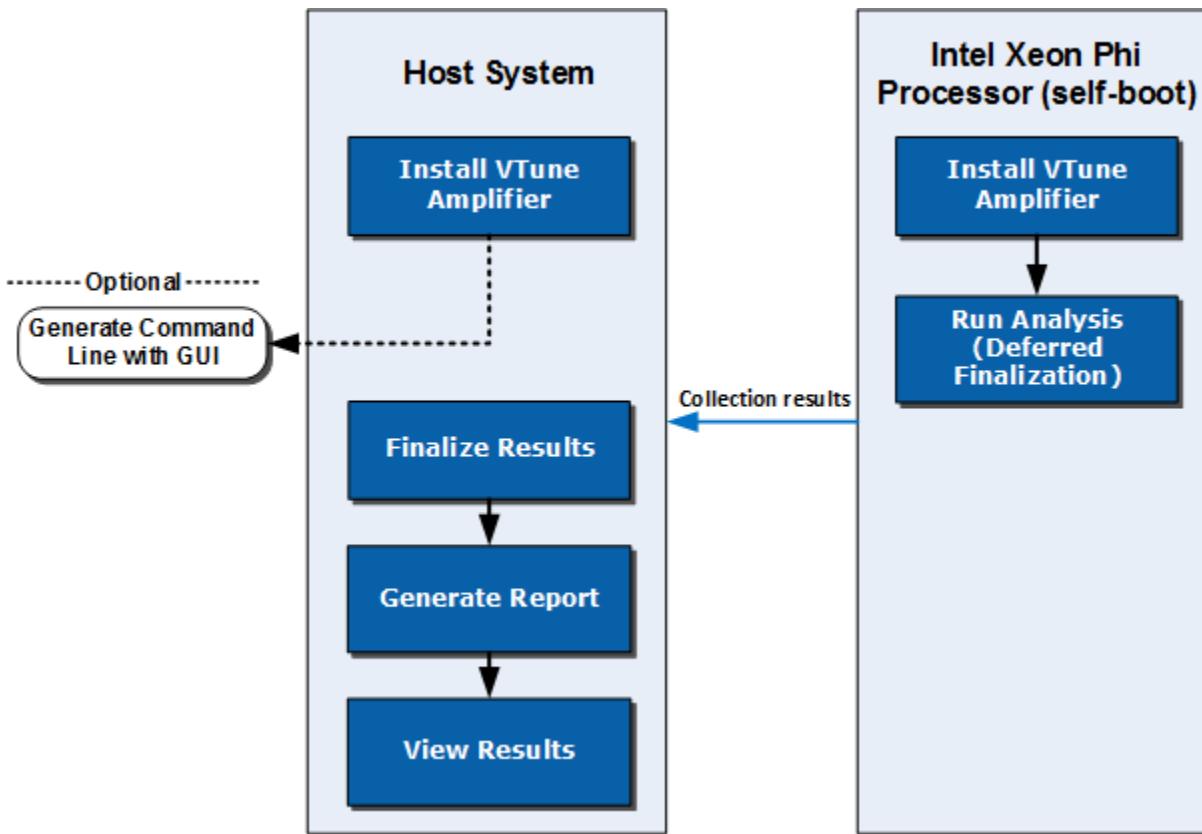
[Search Directories](#)

Intel® Xeon Phi™ Processor Targets

The following figure shows basic workflow required to analyze an application running on Intel® Xeon Phi™ processors (code named Knights Landing and Knights Mill) based on Intel Many Integrated Core Architecture (Intel® MIC Architecture) or perform a system-wide analysis using Intel® VTune™ Profiler. Analysis is supported on a Linux* target with the self-boot version of the Intel Xeon Phi processor. You may choose to run one of the predefined analysis types, HPC Performance Characterization, Memory Access, Microarchitecture Exploration, Hotspots, or create a custom analysis type.

NOTE

Instrumentation-based collections such as Hotspots in the user-mode sampling mode or Threading analysis can cause a significant overhead on the number of worker threads. Instead, use Hotspots analysis in the hardware event-based sampling mode or HPC Performance Characterization to explore application scalability.

**NOTE**

The workflow represented in the diagram is the recommended flow to speed up the analysis process. It is possible to run the full Intel VTune Profiler collection on the Intel Xeon Phi processor, but finalization and visualization might be slow. You can follow the regular analysis flow directly on the target Intel Xeon Phi processor.

Prerequisites

It is recommended to [install the sampling driver](#) for hardware event-based sampling collection types such as HPC Performance Characterization, Memory Access, Microarchitecture Exploration, or Hotspots (hardware event-based sampling mode). If the sampling driver is not installed, Intel VTune Profiler can work on Linux Perf*. Be aware of the following system configuration settings:

- To enable system-wide and uncore event collection that allows the measurement of DRAM and MCDRAM memory bandwidth that is a part of the Memory Access and HPC Performance Characterization analysis types, use root or sudo to set `/proc/sys/kernel/perf_event_paranoid` to 0.

```
echo 0>/proc/sys/kernel/perf_event_paranoid
```

- To enable collection with the Microarchitecture Exploration analysis type, increase the default limit of opened file descriptors. Use root or sudo to increase the default value in `/etc/security/limits.conf` to `100 * <number_of_logical_CPU_cores>`.

```
<user> hard nofile <100 * number_of_logical_CPU_cores>
<user> soft nofile <100 * number_of_logical_CPU_cores>
```

1. Configure and run analysis on the target system with an Intel Xeon Phi processor

There are two ways to configure and run the analysis on the target system:

- Finalization on host system (recommended): Use a command to run the analysis on the system with the Intel Xeon Phi processor without finalizing. This option results in the best performance.

From a command prompt, run the collection with the deferred finalization option to calculate the binary check sum for proper symbol resolution on the host system. For example, to run a Memory Access analysis: `vtune -collect memory-access -finalization-mode=deferred -r <my_result_dir> ./my_app`

For more information, see [vtune Command Syntax](#) and [finalization-mode](#) topics.

Tip

You can also generate a command using the VTune Profiler GUI as described below. After generating the command, add the `-finalization-mode=deferred` option to the command to delay finalization.

- Finalization on target system: Use the VTune Profiler GUI on the host system to generate a command for the target system with the Intel Xeon Phi processor. Run and finalize the analysis on the target system. This method may not provide the fastest results.

1. In the **WHERE** pane, select **Arbitrary Host** button, set the processor architecture to **Intel® Processor code named Knights Landing**, and specify the operating system type.

2. In the **WHAT** pane, select **Launch Application** and configure the analysis:

- Enter the application name and parameters.
- Select the **Use MPI Launcher** checkbox and provide the launcher name, number of ranks, ranks to profile, and result location.

3. In the **HOW** pane, select and configure an analysis type.

- [Hotspots](#)
- [HPC Performance Characterization](#)
- [Microarchitecture Exploration](#)
- [Memory Access](#)

4. Click the **Command Line** button at the bottom of the window to generate the command.

5. Copy the generated command to a command prompt on the target system and run the analysis.

Finalization begins after the analysis completes. Finalization may take several minutes.

2. Open the result on the host system

Copy the result to the host system (if the results collected on the target system are not available on the host via a share). Finalize the result if your command specified deferred finalization.

- Copy the result to the host system using SSH or a similar method.
- [Optional] Finalize the result by providing the result file and search directories to the binaries of interest if the module paths are different from the target system. For example: `vtune -finalize -r <my_result_dir> -search-dir <my_binary_dir>`

3. Open and interpret analysis results

There are two ways to view the results:

- View results in the command line by running a command to generate a report based on the data collected. For example, the following command creates a hotspots report: `vtune -report hotspots -r <my_result_dir>`
- Launch Intel VTune Profiler on the host system and view the result file.
 - Open Intel VTune Profiler.
 - Use the open result action on the toolbar or from the menu button to browse to the result file.
 - Analyze the results and make optimizations to your application.
 - [HPC Performance Characterization Data View](#)

- Memory Usage Data View
- Hotspots Data View
- Microarchitecture Exploration Data View

See Also

[Custom Analysis](#)

[Dialog Box: Binary/Symbol Search](#)

[Dialog Box: Source Search](#)

Targets in Virtualized Environments

Configure your system to use the Intel® VTune™ Profiler for targets running in such virtualization environments as Hyper-V* on Windows*, KVM* or VMWare ESXi* on Linux*, and others.

Virtual machines are made up of the following components:

- *Host operating system*: system from which the virtual machine is accessed. Supported host systems: Linux*, Windows*
- *Virtual machine manager (VMM)* or *cloud service provider*: tool used to access and manage the virtual machine.
- *Guest operating system*: system accessed via the VMM and profiled using Intel VTune Profiler. Supported guest systems: Linux*, Windows*

In most cases, the VTune Profiler is installed on the guest operating system and analysis is run on the guest system. The guest system may not have full access to the system hardware to collect performance data. Analysis types that require access to system hardware, such as those that require uncore event counters, will not work on a virtual machine.

NOTE

Typically the host operating system has access to the system hardware to collect performance data, but there are cases in which the host system may also be virtualized. If this is the case and you want to collect performance data on the host system, treat the host system as you would a guest system and assume that it no longer has the same level of access to the system hardware.



Analysis Type Support

Support for VTune Profiler analysis types varies depending upon which counters have been virtualized by the VMM. You can refer to the documentation for your VMM to get a list of virtualized counters.

If you run an analysis type that cannot be run in a virtualized environment, VTune Profiler displays a warning message.

VTune Profiler uses the two sampling-based collection modes for analysis:

- [User-Mode Sampling](#)

In general, the [Hotspots](#) analysis type in this mode will work on every supported VMM because the analysis type does not require access to the system hardware.

- [Hardware Event-Based Sampling](#)

Analysis types that use this mode ([Hotspots](#) and [Microarchitecture Exploration](#)) have limited reporting functionality. For example, they may not include accurate results for stacks because this data relies on information provided by [precise events](#). Running analysis types that rely on precise events will return results, but the collected data will be incomplete or distorted. That is, the result may not point to the actual instruction that caused the event, which can be difficult to differentiate from correct events.

To enable performance analysis in the hardware event-based sampling mode on a virtual machine, additional configuration steps are required. As soon as you installed VTune Profiler, you need to enable the vPMU for your hypervisor:

- VMware*
- Hyper-V*
- KVM*
- Xen* Project
- Parallels* Desktop

NOTE

Analysis types based on uncore events ([Memory Access](#), [Input and Output analysis](#), and others) and related performance metrics (Memory Bandwidth, PCIe Bandwidth, and others) are not supported on virtual machines.

Virtual Machine Host/Guest Support

A typical virtualized environment includes a host operating system, which boots first and from which the VMM is loaded, and virtual machines (VMs) running guest operating systems. There are multiple combinations of each and support varies based on each component.

	Linux Host	Windows Host
Linux Guest	KVM Hyper-V VMware	VMware
Windows Guest	VMware	VMware Hyper-V

VTune Profiler supports profiling host and guest OS from the host system. This type of analysis is only available for virtual machines with [KVM hypervisor](#) as a preview feature.

See Also

[Profile Targets on a KVM* Guest System](#)

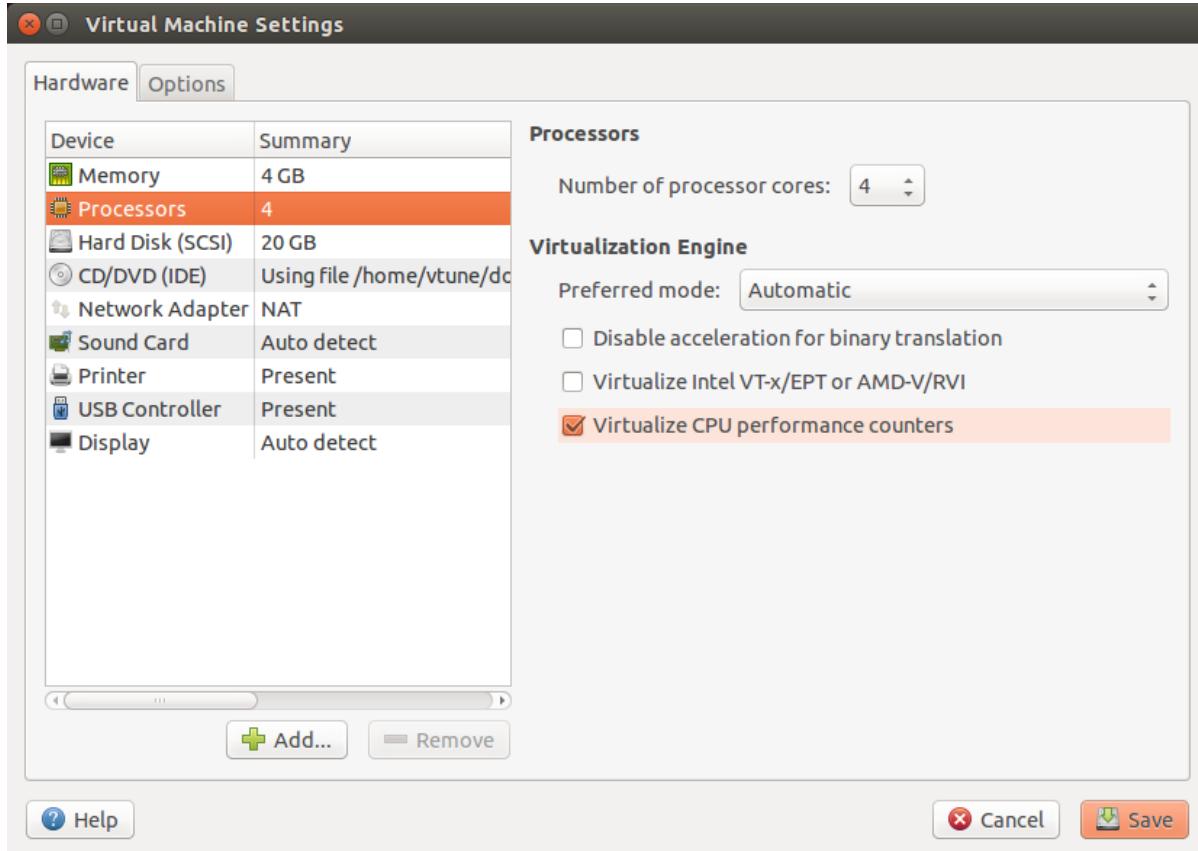
Profile Targets on a VMware* Guest System

[Configure the Intel® VTune™ Profiler to analyze performance on a VMware* guest system.](#)

VMware users can use the VTune Profiler to analyze a Windows* or Linux* virtual guest system. VTune Profiler is installed and run on the guest system. Additional information about installing VTune Profiler is available from the installation guides. Refer to the installation guide for the guest system operating system (Windows or Linux).

Use the following steps to enable event-based sampling analysis on the VMware virtual machine. Refer to the VMware documentation for the most up-to-date information.

1. From the host system, open the configuration settings for the virtual machine.
2. Select the **Processors** device on the left.
3. Select the **Virtualize CPU performance counters** checkbox.
4. Click **Save** to apply the change.



See Also

[Hardware Event-based Sampling Collection](#)

Profile Targets on a Parallels* Guest System

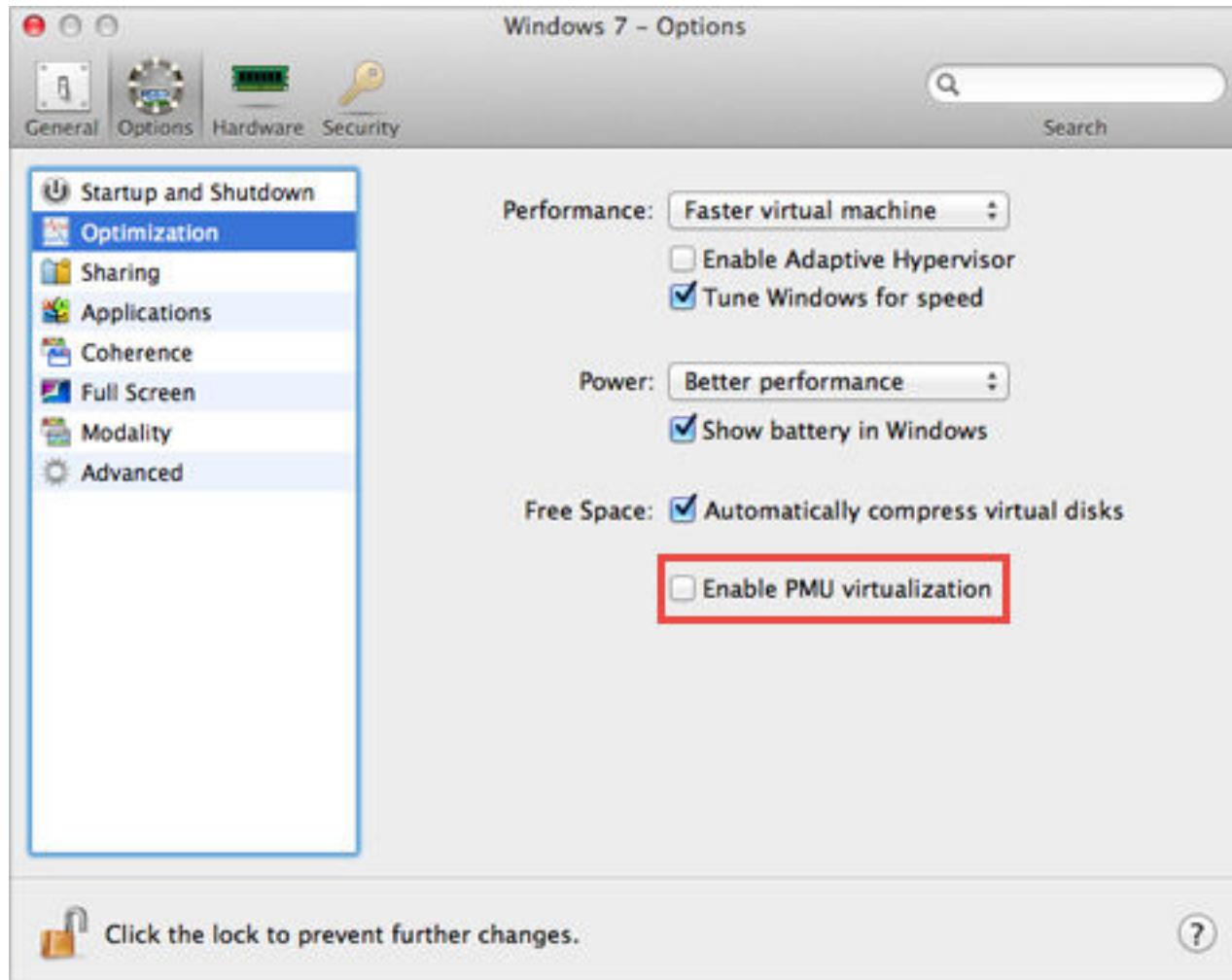
Configure the Intel® VTune™ Profiler to analyze performance on a Parallels guest system.*

Parallels* Desktop users can use Intel® VTune™ Profiler to analyze a Windows* or Linux* virtual guest system using a macOS* host. Intel VTune Profiler is installed and run on the guest system. Additional information about installing VTune Profiler is available from the installation guides. Refer to the installation guide for the guest system operating system (Windows or Linux).

Use the following steps to enable event based sampling analysis inside Parallels virtual machines. Refer to the Parallels documentation for the most up-to-date information.

1. Open the configuration options:
 - Click the Parallels icon in the menu bar, press and hold the Option (Alt) key, and choose **Configure**.

- Choose **Virtual Machine > Configure** from the Parallels Desktop menu bar at the top of the screen.
2. Select the **Options** tab.
3. Select **Optimization**.
4. Select the **Enable PMU virtualization** checkbox.



See Also

[Install Intel® VTune™ Profiler](#)

[Hardware Event-based Sampling Collection](#)

Profile Targets on a KVM* Guest System

Configure the Intel® VTune™ Profiler to analyze performance on a KVM guest system.

Performance analysis for the host and virtual machine(s) in cloud environments helps identify such issues as resource contention (for example, CPU/vCPU time) and network/IO activity. VTune Profiler uses Perf*-based driverless collection to enable performance analysis of the guest Linux* operating system via Kernel-based Virtual Machine (KVM) from the host system.

Unlike other virtual machine systems, systems using KVM on a Linux* host to access a Linux guest can have VTune Profiler installed on either the host system to analyze performance on the guest system or installed directly on the guest system to analyze the guest system. Additional information about installing VTune Profiler is available from the Linux installation guides.

Depending on your analysis target, you may choose any of the supported usage modes for KVM guest OS profiling.

Profiling Modes

Currently, the VTune Profiler supports the following usage modes for KVM guest OS profiling, and each of them has some limitations:

Profiling System	KVM Guest OS (User Apps)	KVM Guest OS (User and Kernel Space)	Host and KVM Guest OS (User and Kernel Space) (preview feature)
Supported analysis	User-mode sampling: Hotspots and Threading	Event-based sampling: Hotspots and limited Microarchitecture Exploration	Event-based sampling: all types with accurate attribution of user-space activity to the user processes on the guest
Target type	Applications in the Launch and Attach modes	<ul style="list-style-type: none"> Applications in the Launch and Attach modes System-wide analysis 	System-wide analysis (host and guest OS)
VTune Profiler installation mode	On the guest OS	On the guest OS	On the host and guest OS (VTune Profiler custom collector)
Limitations	No system-wide analysis for user-mode sampling	<ul style="list-style-type: none"> Limited event-based sampling analysis due to a limited set of virtualized PMU events and unavailable uncore events No information from the host 	<ul style="list-style-type: none"> Additional debugfs and custom collector configuration is required Access to the host system running VM is required Not applicable to cloud environments
Configuration	Learn more	PMU event virtualization required for Event-based sampling Learn more	Analyze KVM guest OS option Learn more

See Also

[Install Intel® VTune™ Profiler](#)

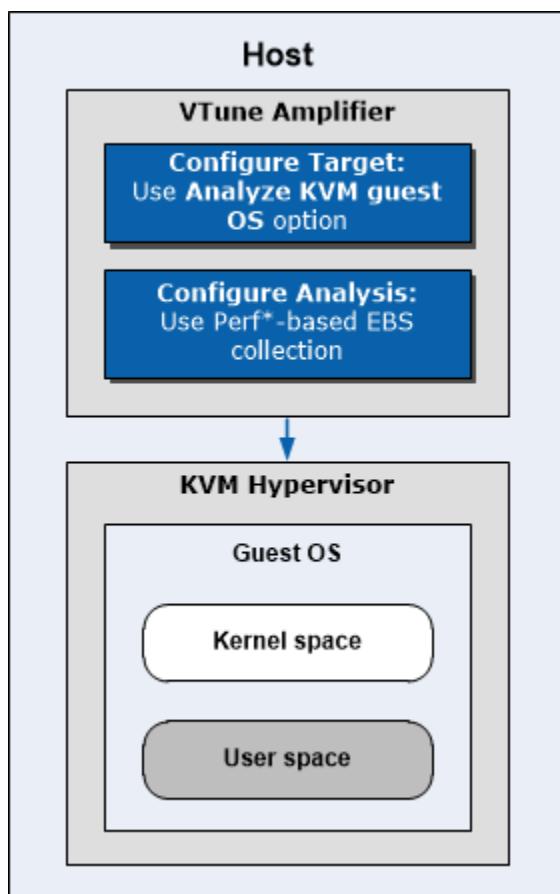
[analyze-kvm-guest](#)

[kvm-guest-kallsyms](#)

[kvm-guest-modules](#)

Profile KVM Kernel Modules from the Host

If you are a system developer and interested in the performance analysis of a guest Linux* system including KVM modules, consider using this usage mode:



1. Prepare your system for analysis:
 - a. Copy the `/proc/kallsyms` and `/proc/modules` files from a guest OS to a host file system to have KVM guest OS symbols resolved.
 - b. Copy any guest OS's modules of interests (`vmlinuz` and any `*.ko` files) from a guest OS and save them to a `[guest]` folder on the host file system.
2. Click the



Configure Analysis button on the VTune Profiler toolbar.

- The **Configure Analysis** window opens.
3. Make sure to select the **Local Host** target system in the **WHERE** pane and configure the required target type in the **WHAT** pane.

By default, the **Launch Application** target type is selected.

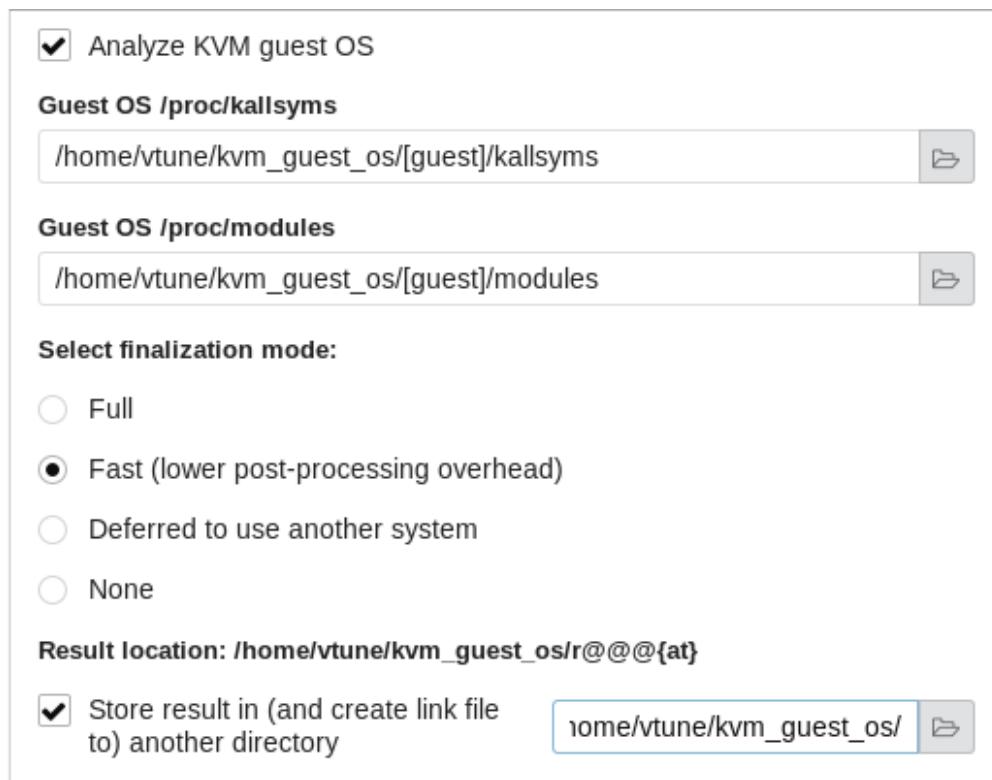
If you select the **Attach to Process** target type, specify the `qemu-kvm` process to attach to.



Alternatively, you may specify the PID of the `qemu-kvm` process. To determine the PID, enter:

```
$ ps aux | grep kvm
```

4. In the **Advanced** section of the **WHAT** pane, select the **Analyze KVM guest OS** option and enter paths to the local copies of the guest /proc/kallsyms and /proc/modules files; for example:



5. Click the

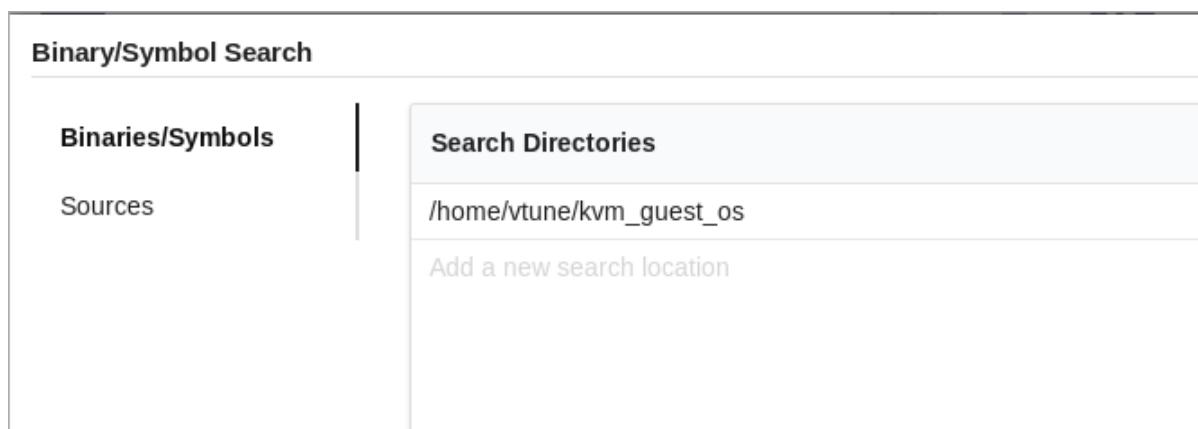


Search Binaries button on the bottom right.

The **Binary/Symbol Search** dialog box opens.

6. Add a local path to a [guest] folder where all modules copied from the guest OS reside.

For example, if your [guest] folder is located in /home/vtune, specify /home/vtune as a search directory:



7. Click **OK** to save your changes.

8. In the **HOW** pane, select a required analysis type.

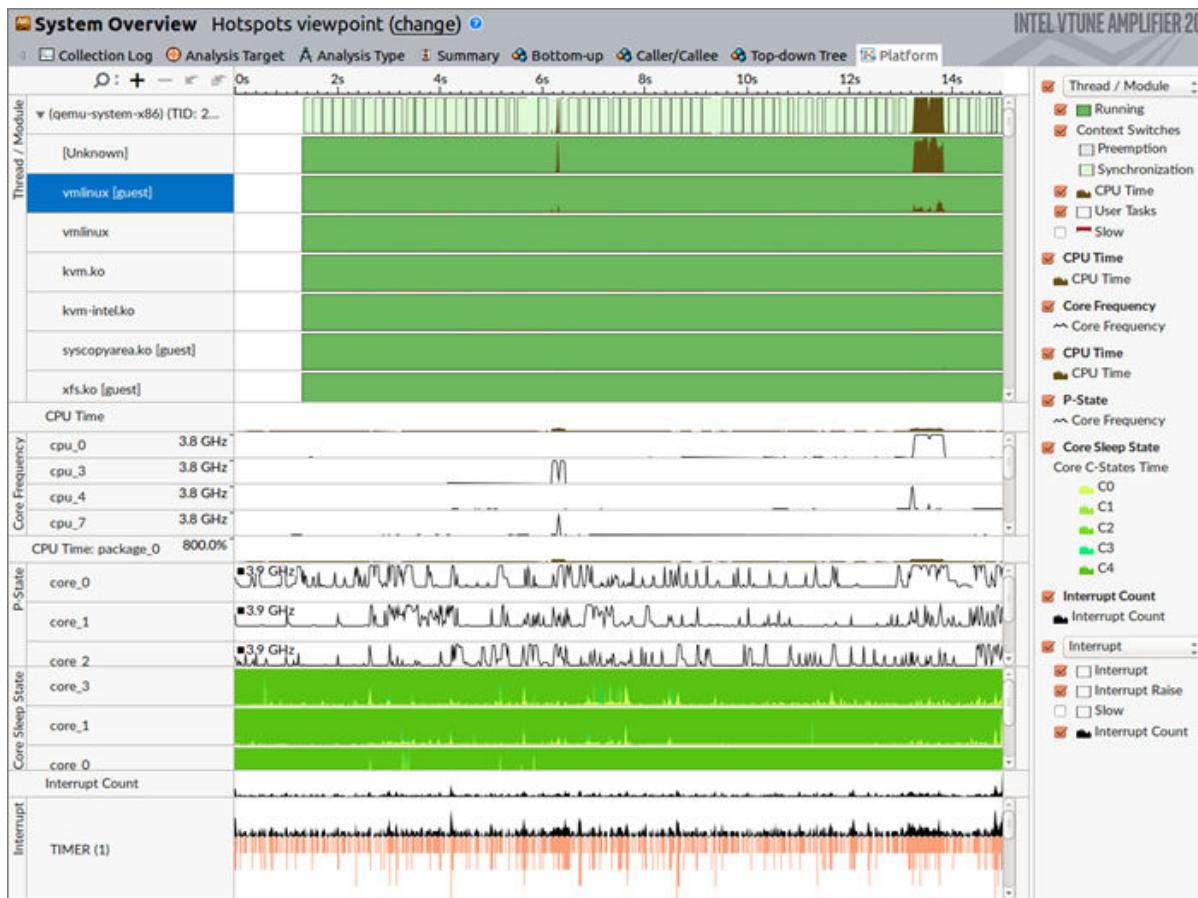
For KVM guest OS profiling, you may choose analysis types using Perf*-based EBS data collection: Hotspots (hardware event-based sampling mode), System Overview, or configure your own custom analysis.

- Click the **Start** button at the bottom to run the analysis.

When you run the analysis, the VTune Profiler collects the data on both host and guest OS and displays merged statistics in the result. Guest OS modules have the [guest] postfix in the grid. For example:

Module / Function / Call Stack	CPU Time			Instructions Retired	CPI Rate
	Effective Time by Utilization	Spin Time	Overhead Time		
▶ [Unknown]	776.136ms	0ms	0ms	3,845,800,000	0.785
▶ vmlinux [guest]	152.350ms	0.400ms	0ms	246,050,000	2.395
▶ copy_user_generic_string	16.584ms	0ms	0ms	2,450,000	25.286
▶ avtab_search_node	11.589ms	0ms	0ms	5,600,000	8.313
▶ memcpy	8.592ms	0ms	0ms	700,000	50.500
▶ async_page_fault	6.194ms	0ms	0ms	1,400,000	17.750
▶ clear_page_c	5.794ms	0ms	0ms	1,400,000	16.500

Focus on the **Platform** tab to analyze your code performance on the guest OS and correlate this data with CPU, GPU, power, hardware event metrics and interrupt count at each moment of time. If you enabled the [kvm Ftrace event collection](#) for your target, you can also monitor the statistics for KVM kernel module:



Limitations

- In this mode, the VTune Profiler collects data only on the kernel space modules on the KVM guest OS. Data on user space modules shows up in the [Unknown] node and includes only high-level statistics.

- Call stack data is not collected for this type of profiling.

See Also

[analyze-kvm-guest](#)

vtune option

[kvm-guest-kallsyms](#)

vtune option

[kvm-guest-modules](#)

vtune option

Profile KVM Kernel and User Space on the KVM System

Install the VTune Profiler on the KVM system and configure your target for the KVM guest OS profiling.

For application analysis, you need to install the Intel® VTune™ Profiler directly on your guest OS. VTune Profiler installation detects a virtual environment and disables sampling drivers installation to avoid system instability. When the product is installed, proceed with [project configuration](#) by specifying your application as an analysis target and selecting an analysis type:



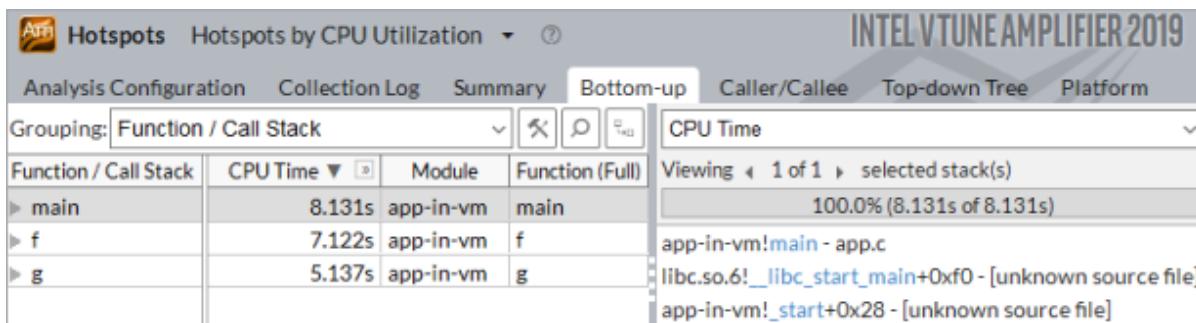
This profiling type supports two usage modes:

- [Guest OS \(user apps\)](#)
- [Guest OS \(kernel and user space\)](#)

Both profiling modes are applicable to cloud environments but introduce some limitations.

Guest OS (User App) Profiling Mode

In this mode, the VTune Profiler supports user-mode sampling and tracing analysis types, Hotspots and Threading, for the applications running in the Launch or Attach mode. System-wide analysis is not supported.

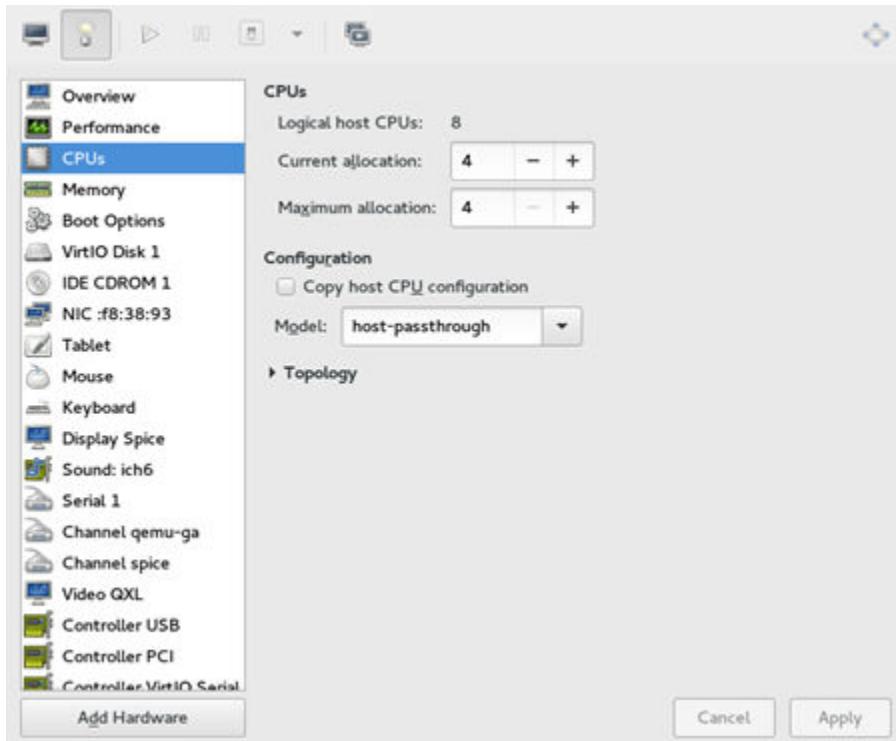


Guest OS (Kernel and User Space) Profiling Mode

In this mode, the VTune Profiler provides limited event-based collection options for the Hotspots and Microarchitecture Exploration analyses and requires additional host system configuration to virtualize PMU counters.

To enable event-based sampling analysis on the KVM system:

1. From the host system, open the configuration settings for the virtual machine.
2. Select the **CPUs** or **Processor** option on the left.
3. Enter host-passthrough into the **Model** field to pass through the host CPU features without modifying the guest system.
4. Click **Apply** to save the changes.



When you select a hardware event-based analysis type (for example, Microarchitecture Exploration), the VTune Profiler automatically enables a driverless event-based sampling collection using the Linux Perf* tool. For this analysis, the VTune Profiler collects only architectural events. See the [Performance Monitoring Unit Sharing Guide](#) for more details on the supported architectural events.

Limitations

- User-mode sampling limitations:
 - Only Hotspots and Threading analyses are supported.
 - No system-wide analysis is available.
- Hardware event-based sampling limitations:
 - Only Hotspots and limited Microarchitecture Exploration analyses are supported.
 - PEBS counters are not virtualized.
 - Uncore events are not available.
- KVM modules and host system modules do not show up in the analysis result.
- Data on the guest OS and your application modules show up as locally collected statistics with no [guest] markers.

See Also

[analyze-kvm-guest](#)

vtune option

[kvm-guest-kallsyms](#)

vtune option

kvm-guest-modules

vtune option

Dialog Box: Binary/Symbol Search**Profile KVM Kernel and User Space from the Host**

In this mode, Intel® VTune™ Profiler collects two traces in parallel: system-wide performance data trace on the host and OS-level event trace on the guest system. These traces get merged into one VTune Profiler result and provide:

- simultaneous analysis of user space activity (processes, threads, functions) from the host on the guest system;
- accurate attribution of collected data to the user processes running on the guest, based on the timestamp synchronization.

This usage mode provides the following advantages:

- VMs are not required to virtualize performance counters. All performance analysis features are available to VM users out of the box.
- Sampling drivers (VTune Profiler sampling driver or Perf*) do not need to be installed on a guest VM.

To enable KVM kernel and user space profiling from the host:

1. Install the VTune Profiler on the host and virtual machine.

NOTE

You do not need to install sampling drivers.

2. On both host and guest systems, run the script from the bin64 folder as a root:

```
$ prepare-debugfs.sh -g <user_group>
$ echo 0 > /proc/sys/kernel/perf_event_paranoid
```

3. [Configure a password-less SSH access](#) from the host to the KVM guest system.

4. If your host system is multi-socket, export the environment variable to set the time source to TSC before starting the VTune Amplifier:

```
VTUNE_RUNTOOL_OPTIONS=--time-source=tsc
```

5. [Create a project](#).

6. From the **WHAT** pane in the **Configure Analysis** window, expand the **Advanced** section and enter the following string to the **Custom collector** field:

```
python <vtune_install_dir>/bin64/kvm-custom-collector.py --kvm-ssh-
login=<username>@<kvm_ssh_ip> --vtune-dir-on-kvm=<vtune-install-dir>
```

NOTE

For additional details on particular options, see the `kvm-custom-collector.py` script help.

7. To collect data from the guest kernel space, select the **Analyze KVM Guest OS** option.

Copy `/proc/kallsyms` and `/proc/modules` files from the virtual machine to the host.

NOTE

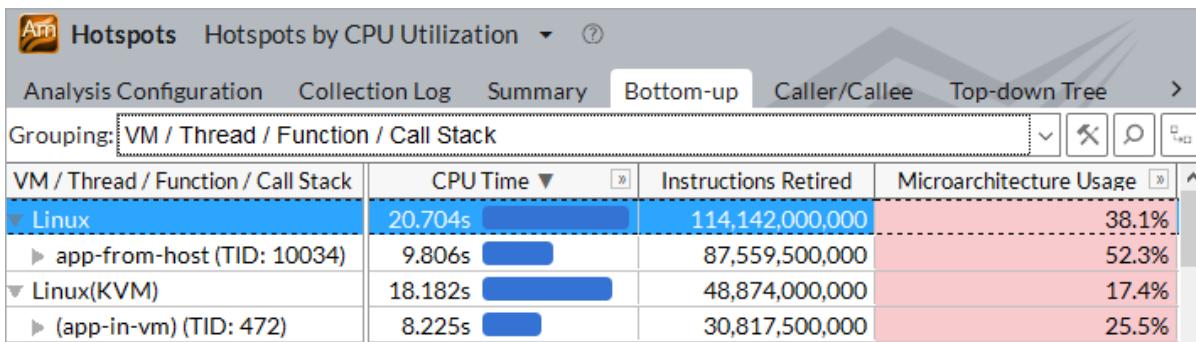
Since these are pseudo-files, you are recommended to `cat` their content into a regular file and then copy it to the host. Specify paths to the copied files in the project properties.

8. From the **HOW** pane, select any hardware event-based sampling analysis (for example, General Exploration) and run the analysis from the host.

Explore the collected data by enabling all the grouping levels containing a VM component to differentiate the host and target data.

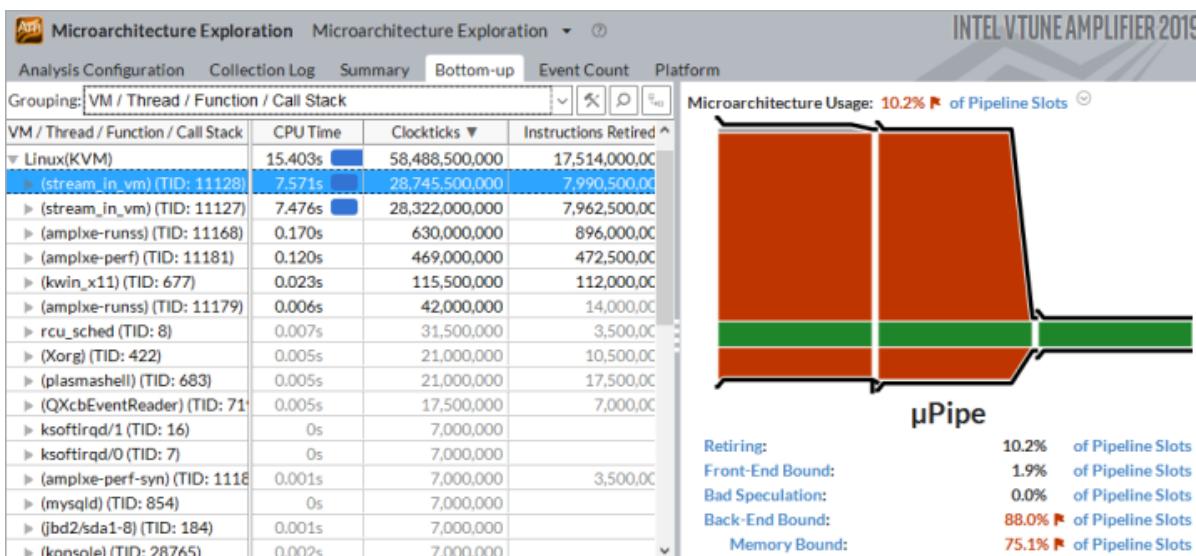
Example 1: Hotspots Analysis (Hardware Event-Based Sampling Mode)

Analyze hotspots for both an application launched from the Linux host, app-from-host, and an application launched on the KVM guest system, app-in-vm:



Example 2: Microarchitecture Exploration Analysis

Analyze the efficiency of the Microarchitecture Usage for the application launched on the KVM guest system. The context summary on the right pane shows the hardware metrics for the thread (launched inside the KVM) selected in the grid:



System Requirements and Limitations

- Minimum Linux kernel version for host system is 4.9.
- debugfs is mounted on both host and guest system.
- Irrespective of the number of KVM/Qemu processes running, only one running VM instance can be profiled.
- In the result view, threads with the same name may be grouped into one process (ftrace).
- In the result view, samples before the first context switch may be attributed to the hypervisor thread on the host.

See Also

[Use a Custom Collector](#)

[analyze-kvm-guest](#)

vtune option

[kvm-guest-kallsyms](#)

vtune option

[kvm-guest-modules](#)
[vtune option](#)

Profile Targets on a Xen* Virtualization Platform

Configure Intel® VTune™ Profiler and your system with a Xen virtualization platform for performance profiling.

You can use the VTune Profiler for hardware event-based analysis either for a guest OS (DomU), a privileged OS (Dom0), or all the domains at once.

Configure a Target System for Analysis

Before running a VTune Profiler analysis on a system with a Xen virtualization platform, enable full-platform CPU monitoring required for event-based sampling analysis:

```
$ echo "all" > /sys/hypervisor/pmu/pmu_mode
```

To get CPU profiling data on a virtualized system (Dom0 and the hypervisor only), enter:

```
$ echo "hv" > /sys/hypervisor/pmu/pmu_mode
```

NOTE

- Some configurations do not support the `all` mode.
- CPU events virtualization requires root privileges.
- Unlike CPU profiling, GPU profiling in the `hv` mode is available for all domains (Dom0 and DomU).

Configure VTune Profiler for Xen Platform-Wide Analysis

Prerequisites: Make sure the Dom0 remote analysis target is accessible via the Ethernet/SSH connection from your host [without any password](#).

Create a VTune Profiler [project](#) and specify options for your remote target as follows:

1. Select the **remote Linux (SSH)** type of the target system on the **WHERE** pane.
2. Specify **SSH destination** details for your Dom0 remote target system.
3. Select the **Profile System** target type to enable platform-wide performance monitoring (**WHAT** pane).

As soon as you set up the target options, the VTune Profiler attempts to automatically install required components on the specified remote system. If, for some reason, the system cannot be reached, VTune Profiler displays an error message. To troubleshoot this potential problem, make sure the default path specified as the **VTune Profiler installation on the remote system** in the **WHERE** pane is accessible, writable, and has 200Mb of available space. If not, specify another location, for example: `/tmp`.

As soon as the connection is established and the target is configured, select an analysis type supported on the Xen virtualization platform from the **HOW** pane:

- [Microarchitecture Exploration](#)
- [GPU Rendering \(preview\)](#)

See Also

[Set Up Analysis Target](#)

[Set Up Remote Linux* Target](#)

[target-system](#)
[vtune option](#)
[target-tmp-dir](#)

vtune option

Profile Targets in the Hyper-V® Environment

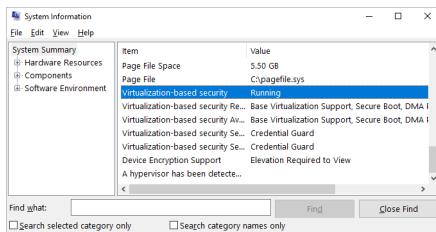
Configure your Windows® system to enable hardware event-based performance analysis in the Hyper-V virtualization environment.

VTune Profiler supports performance profiling in the Hyper-V environment with some limitations applicable to the [event-based sampling collection](#). So, before you start the analysis, make sure your system configuration satisfies the requirements.

Verify Your System Configuration for Hardware Analysis

- For the hardware analysis in your Hyper-V environment, make sure your system runs on:
 - Intel microarchitectures code named Skylake, Goldmont, or later;
 - Windows 10 RS3 operating system (version 1709) or later. To check the system version, use the `winver` command.
- Run the `msinfo32` command to make sure the Hyper-V is enabled and running.

The **System Summary** in the **System Information** dialog box should show the **Virtualization-based security** item as Running:



NOTE

If your system does not meet the profiling requirements but you plan to run hardware event-based sampling analysis with VTune Profiler, make sure to [disable the Hyper-V feature in the system settings](#).

Disable the Credential Guard and Device Guard on Hyper-V

The Hyper-V has optional security features: Device Guard and Credential Guard. When either or both of them are enabled, accessing non-architectural PMU MSRs triggers (required for the driver-based hardware event sampling analysis) a general protection fault. For example, offcore response MSRs and uncore related MSRs are non-architectural MSRs. To collect these events, you must disable the security features as follows:

- Make sure the security features are running on your system:
 - Run the `msinfo32` command to open the **System Information** dialog.
 - In the **System Summary**, check whether the **Virtualization-based Security Services Running** item includes **Hypervisor enforced Code Integrity** and/or **Credential Guard** values.
- Disable these security features by running the Microsoft® DG-CG-Readiness-Tool, available at <https://www.microsoft.com/en-us/download/details.aspx?id=53337>:
 - Open Powershell as an administrator and go to the tool installation directory.
 - Run the tool as follows:


```
. \DG_Readiness_Tool_v2.1.ps1 -Disable -CG -DG
```
 - Reboot the system.
 - Make sure the device guard is turned off. The output from `msinfo32` should NOT include either **Hypervisor enforced Code Integrity** or **Credential Guard**.

See Also

[Error Message: Cannot Enable Event-Based Sampling Collection](#)

Targets in a Cloud Environment

You can use Intel® VTune™ Profiler to run application performance analysis in the user-mode sampling mode on Windows* or Linux* virtual machine based instances or any analysis type on a bare-metal cloud instance.

These cloud service providers are supported:

- Amazon Web Services* (AWS)
- Google Cloud Platform*
- Microsoft Azure*

You can install VTune Profiler either directly on the cloud instance or on a Windows, Linux, or macOS* host system and target a Linux cloud instance for remote analysis.

Prerequisites:

- Existing account with one of the supported cloud service providers
- Existing Linux or Windows instance in the cloud
- Linux instance: Root or sudo privileges to enable user-mode sampling Hotspots analysis by setting /proc/sys/kernel/yama/ptrace_scope to 0. See the [Intel VTune Profiler Release Notes](#) for instructions on enabling it permanently.
- If installing in the cloud: At least 25GB of instance storage

To install VTune Profiler on the cloud instance, copy the VTune Profiler installer to the cloud instance and run the installer. For more information, see the [VTune Profiler Install Guide](#).

A use case with steps for installing and configuring VTune Profiler on an Amazon Web Services instance and running a Hotspots analysis on that instance is available from the [VTune Profiler Cookbook](#).

Arbitrary Targets

Configure and generate a command line for performance analysis on a system that is not accessible from the current host.

Besides targets accessible to Intel® VTune™ Profiler directly on the host or via a remote connection (SSH or ADB), you have an **Arbitrary Host** option to create a command line configuration for a platform not accessible from the current host. You can select any of the supported hardware platforms and operating systems, configure corresponding target and analysis options, and generate a command line by clicking the **Command Line** button. The generated command line will be saved in the buffer and can be used later on the intended host.

NOTE

The option to [generate a command line from GUI](#) via the **Command Line** button is available for both accessible and arbitrary targets.

To configure an analysis for an arbitrary host:

1. Create a new project or click the



Configure Analysis toolbar button for an existing project.

2. From the **Configure Analysis** window, click the



Browse button on the **WHERE** pane and select the **Arbitrary Host (not connected)** type of the target system.

3. Specify a platform for profiling:

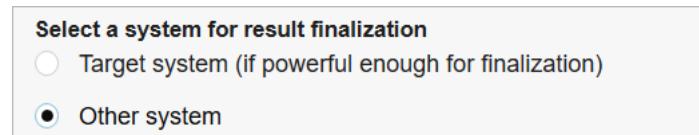
- Select a hardware platform for analysis from the drop-down menu, for example: Intel® processor code named Anniedale.
- Specify either Windows* or GNU*/Linux* operating system.

4. Switch to the **WHAT** pane to configure analysis target options.

For MPI analysis of an arbitrary target, enable the **Use MPI launcher** check box to generate a command line configuration. Configure the following MPI analysis options:

- **MPI launcher:** Select an MPI launcher that should be used for your analysis. You can either enable the **Intel MPI** launcher option (default) or select **Other** and specify a launcher of your choice, for example: `aprun`, `srun`, or `lbrun`.
- **Number of ranks:** Specify the number of ranks used for your application.
- **Profile ranks:** Use **All** to profile all ranks, or choose **Selective** and specify particular ranks to profile, for example: `2-4,6-7,8`.
- **Result location:** Specify a relative or absolute path to the directory where the analysis result should be stored.

If your target system is not powerful enough, consider selecting another system for the result finalization as follows:



In this case, VTune Profiler calculates only binary checksum to be used for finalization on the host machine. This option is recommended for [analysis on the Intel Xeon Phi processor](#) (code name: Knights Landing).

5. Switch to the **HOW** panechoose and configure (if required) an analysis type.

6. Click the



Command Line... button at the bottom to generate a command line for your configuration.

For example, VTune Profiler generates the following command line for a `test` MPI application that will be launched on a GNU/Linux system via Intel MPI launcher and analyzed for Memory Access issues on ranks `2-4,6-7,8`:

```
$ mpirun -n 14 -gtool "vtune -collect memory-access:2-4,6-7,8" /temp/vtune/test
```

7. Click the **Copy** button to copy the generated command line to the buffer and use it later on the intended host.

See Also

[Analysis Target](#)

[Set Up Analysis Target](#)

[Finalization](#)

[MPI Code Analysis](#)

Embedded System Targets

Use the Analysis Communication Agent to profile embedded systems running real-time operating systems supporting the TCP/IP protocol suite, as well as their applications.

Intel® VTune™ Profiler offers the **Communication Agent (TCP/IP)** connection type that enables you to profile embedded systems running real-time operating systems and their applications. Using the Analysis Communication Agent and the sampling driver, you can configure your operating system to enable remote performance profiling using VTune Profiler.

This analysis configuration requires an implementation of the sampling driver and the Analysis Communication Agent for your system. An open reference solution for the Linux* OS kernel is available through the Analysis Communication Agent [GitHub* repository](#). You can use this reference solution to create custom implementations of the driver and the Analysis Communication agent. Detailed implementation information and instructions are available in the [Analysis Communication Agent documentation](#).

You can profile your operating system via the Analysis Communication Agent using the [Hotspots](#) and [Microarchitecture Exploration](#) analysis types in the Profile System mode.

NOTE

This connection type uses the TCP/IP protocol suite. This connection is not secure, and it is recommended to use this connection type in a secure lab environment.

This analysis configuration includes the following components:

- **Target side:**

- **Sampling Driver**

The sampling driver is a module that is loaded into the kernel of your operating system that enables the collection of performance data.

- **Analysis Communication Agent**

The Analysis Communication Agent is a software agent that runs on the target system which serves as a connection between the VTune Profiler collector running on the host side and the sampling driver running on the target system.

- **Host side:**

- **Communication Agent (TCP/IP) connection type**

The **Communication Agent (TCP/IP)** connection type is used to connect to the Analysis Communication Agent running on the target system via the TCP/IP protocol suite.

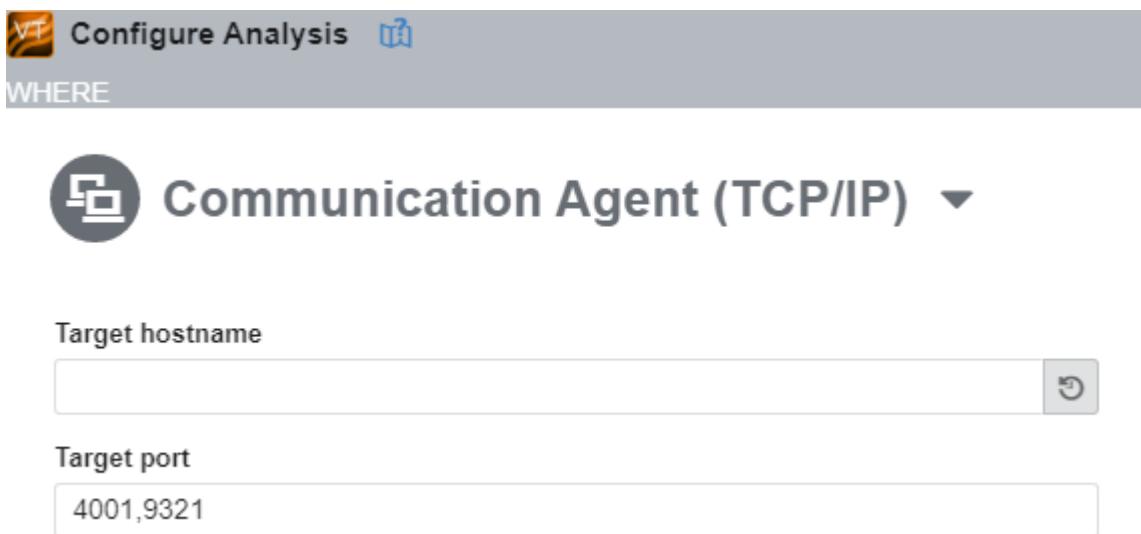
Prerequisites

- Sampling driver and Analysis Communication Agent implementations for your target system. You can use the [reference solution](#) to help implement and build these components.
- A TCP/IP capable operating system with the sampling driver loaded and Analysis Communication Agent launched.
- A host system with VTune Profiler installed.

Run Analysis

Once the target system is ready, follow these steps to run an analysis:

1. Launch VTune Profiler on the host system.
2. (Optional) Click the **New Project** button to create a new project.
3. Click **Configure Analysis** and select the **Communication Agent (TCP/IP)** connection type in the **WHERE** pane.



4. Specify the target hostname and port.
5. Configure any desired options in the **WHAT** pane.
6. Select the analysis type in the **HOW** pane.
7. In the **Binary/Symbol Search** window, browse to the location of the kernel and application target modules on the host system.
8. Click the  **Start** button to run the analysis.
9. Analyze the result using the VTune Profiler GUI to identify any performance bottlenecks in the kernel or applications.

Analyze Performance

After you [create a project](#) and [specify a target for analysis](#), you are ready to run your first analysis.

Performance Snapshot

Click **Configure Analysis** on the Welcome page. By default, this action opens the **Performance Snapshot** analysis type. This is a good starting point to get an overview of potential performance issues that affect your application. The snapshot view includes recommendations for other analysis types you should consider next.

Analysis Groups

Click anywhere on the analysis header that contains the name of the analysis type. This opens the **Analysis Tree**, where you can see other analysis types grouped into several categories. See [Analysis types](#) to get an overview of these predefined options.

Advanced users can create [custom analysis types](#) which appear at the bottom of the analysis tree.

Analysis Group	Analysis Types
Algorithm analysis	<ul style="list-style-type: none"> • Hotspots • Anomaly Detection • Memory Consumption
Microarchitecture analysis	<ul style="list-style-type: none"> • Microarchitecture Exploration • Memory Access
Parallelism analysis	<ul style="list-style-type: none"> • Threading

Analysis Group	Analysis Types
I/O analysis	<ul style="list-style-type: none"> HPC Performance Characterization Input and Output
Accelerators analysis	<ul style="list-style-type: none"> GPU Offload GPU Compute/Media Hotspots (Preview) CPU/FPGA Interaction NPU Exploration Analysis (Preview)
Platform Analyses	System Overview

Aspects of Analysis Types

- You can run an analysis type using the graphical interface (`vtune-gui`) or from the [command line interface](#) (`vtune`).
- All analysis types in VTune Profiler are based on one of these data collection types:
 - [User-mode sampling and tracing collection](#)
 - [Hardware event-based sampling collection](#) (driver-based or [driverless mode](#)), optionally extended with the stack collection
- Each analysis type provides a set of performance metrics that helps you sort out the problems in your code and understand how to optimize it.

VTune Profiler also supports remote collection modes through the GUI and command line, using the [SSH](#) or [ADB](#) connections.

See Also

[Run Command Line Analysis](#)

[Reference](#)

[Run Energy Analysis](#)

User-Mode Sampling and Tracing Collection

When profiling application execution, the Intel® VTune™ Profiler takes snapshots of how that application utilizes the processors in the system. A thread is considered active at a specific moment if it is ready to execute or is executing (not blocking). The snapshots of the number of running threads at the moment provide a hint to the degree of parallelism of the application as well as how this application utilizes processor resources. VTune Profiler classifies utilization into the ranges: Idle, Poor, Ok, and Ideal.

The user-mode sampling and tracing collector interrupts a process, collects the value of all active instruction addresses and captures a calling sequence for each of these samples. Sampled instruction pointers along with their calling sequences (stacks) are stored in data collection files. Statistically collected IP samples with calling sequences enable the viewer to display a call graph or/and the most time-consuming paths. Use this data to understand the control flow for statistically important code sections.

On Linux* the user-mode sampling and tracing collector embeds an agent library into the profiled application. The agent sets up the OS timer for each thread in the application. Upon timer expiration, the application receives the SIGPROF or [another runtime signal](#) that is handled by the collector.

Average overhead of the user-mode sampling and tracing collector is about 5% when sampling is using the default interval of 10ms.

VTune Profiler uses the user-mode sampling and tracing collector to collect data for the following analysis types:

- [Hotspots](#)

- [Threading](#)
- [Memory Consumption](#)

You can also [create a custom analysis type](#) based on the user-mode sampling and tracing collection.

Collecting Stack Data

When collecting data, the VTune Profiler analyzes no more than one stack per configured interval. It unwinds stacks each 10 milliseconds of thread execution. But the VTune Profiler may decide to skip or emulate stack unwinding for performance reasons. In this case, when processing the collected data during finalization, the VTune Profiler tries to find matching stacks in the history for events without stacks.

This approach reduces stack unwinding overhead but may provide incorrect stacks due to wrong matches. In such cases, the VTune Profiler displays pseudo nodes in the bottom-up/top-down trees marked as [Guessed frame(s)], and [Skipped frame(s)]. See [Troubleshooting](#) to learn how to overcome these problems.

VTune Profiler may also display [Unknown frame(s)] nodes if it could not locate symbol files for system or application modules when unwinding the stack. See [Resolving Unknown Frame\(s\)](#) for more details.

See Also

[Error Message: Application Sets Its Own Handler for Signal](#)

[Hardware Event-based Sampling Collection with Stacks](#)

[Hardware Event-based Sampling Collection](#)

Hardware Event-based Sampling Collection

During the hardware event-based sampling (EBS), also known as Performance Monitoring Counter (PMC) analysis in the sampling mode, the Intel® VTune™ Profiler profiles your application using the counter overflow feature of the Performance Monitoring Unit (PMU).

The data collector interrupts a process and captures the IP of interrupted process at the time of the interrupt. Statistically collected IPs of active processes enable the viewer to show statistically important code regions that affect software performance.

Caution

Statistical sampling does not provide 100% accurate data. When the VTune Profiler collects an event, it attributes not only that event but the entire [sampling interval](#) prior to it (often 10,000 to 2,000,000 events) to the current code context. For a big number of samples, this sampling error does not have a serious impact on the accuracy of performance analysis and the final statistical picture is still valid. But if something happened for very little time, then very few samples will exist for it. This may yield seemingly impossible results, such as two million instructions retiring in 0 cycles for a rarely-seen driver. In this case, you may either ignore hotspots showing an insignificant number of samples or switch to a higher granularity (for example, function).

The average overhead of event-based sampling is about 2% on a 1ms sampling interval.

The number of hardware events (Performance Monitoring Counters) that can be collected simultaneously is limited by CPU capabilities. Usually, it is no more than four events. To overcome this limitation, the VTune Profiler splits the event list into several event groups. Each group consists of events that can be collected simultaneously. VTune Profiler uses one of the following techniques:

- Runs an application several times collecting one event group during each run.
- Runs an application only once and [multiplexes the event groups](#) in a round robin fashion during the run. This technique may not work on some OS/hardware combinations.

During product installation on Linux*, you have an option to install the sampling driver with the per-user filtering enabled. When the filtering is on, the collector gathers data only for the processes spawned by the user who started the collection. When it is off (default), samples from all processes on the system are collected. Consider using the filtering to isolate the collection from other users on a cluster for security reasons. The administrator/root can change the filtering mode by rebuilding/restarting the driver at any time. A regular user cannot change the mode after the product is installed.

By default, the VTune Profiler collector samples your target and does not analyze execution paths. But you can enable the **Collect stacks** option during analysis configuration to make the collector take exact measurements of any hardware performance events or timestamps, as well as collect a call stack to the point where a thread gets activated and inactivated. On Linux* systems, by default, VTune Profiler uses the **driverless Perf collection mode** for the hardware event-based stack analysis.

VTune Profiler uses the hardware event-based sampling collector to collect data for the following analysis types:

- [Anomaly Detection](#)
- [Hotspots](#) (hardware event-based sampling mode)
- [Performance Snapshot](#)
- [Microarchitecture Exploration](#)
- [Memory Access](#)
- [GPU Compute/Media Hotspots](#) (preview)
- [GPU Offload](#) (preview)
- [System Overview](#)
- [Threading](#)
- [HPC Performance Characterization](#)
- [CPU/FPGA Interaction](#) (preview)

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

You can also [create a custom analysis type](#) based on the hardware event-based sampling collection.

Caution

Analysis types that use the hardware event-based sampling collector are limited to only one collection allowed at a time on a system.

Prerequisites:

It is recommended to [install the sampling driver](#) for hardware event-based sampling collection types. For Linux* and Android* targets, if the sampling driver is not installed, VTune Profiler can enable the Perf* driverless collection. Be aware of the following configuration settings for Linux target systems:

- To enable system-wide and uncore event collection, use root or sudo to set `/proc/sys/kernel/perf_event_paranoid` to 0.

```
echo 0>/proc/sys/kernel/perf_event_paranoid
```

- To enable collection with the Microarchitecture Exploration analysis type, increase the default limit of opened file descriptors. Use root or sudo to increase the default value in `/etc/security/limits.conf` to `100*number_of_logical_CPU_cores`.

```
<user> hard nofile <100 * number_of_logic_CPU_cores>
```

```
<user> soft nofile <100 * number_of_logic_CPU_cores>
```

See Also

[Hardware Event-based Sampling Collection with Stacks](#)

User-Mode Sampling and Tracing Collection

Cookbook: Profiling Hardware Without Sampling Drivers

Cookbook: Top-Down Microarchitecture Analysis Method
Intel Processor Events Reference

Allow Multiple Runs or Multiplex Events

Enable multiple runs of the event-based sampling data collection for more accurate analysis results.

Intel® VTune™ Profiler runs the hardware event-based sampling analysis to collect data based on the events defined for the selected analysis type. The number of events it can monitor during a single run is limited by the number of performance counters in your processor. If you enable multiple runs of the data collection, the VTune Profiler runs the hardware [event-based sampling](#) data collector as many times as required to collect data on all the events specified for the analysis type. If you specified an application to launch as an analysis target, the VTune Profiler launches your application each time the hardware event-based sampling collector runs.

VTune Profiler allows to avoid multiple runs of the data collection by multiplexing the use of physical counters within a single sampling run. *Event multiplexing* removes the need for multiple runs of the application, thereby reducing the time needed to complete sampling collection at the cost of lower precision of the result data. Event sample counts collected in the multiplexed mode are extrapolated to the total collection runtime.

Event multiplexing is also useful if the application does not have a long steady state or takes a long time to get to steady state. On the other hand, if application initialization is short and it gets to steady state quickly, then you can do multiple short runs and will not need to do event multiplexing.

To enable/disable multiple runs of the data collection:

1. Click the



Configure Analysis button on the VTune Profiler toolbar.

The **Configure Analysis** window opens.

2. Specify your target system type and select the **Application to Launch** target type.

NOTE

Collecting data in multiple runs is only possible if an application to launch is specified.

3. On the **WHAT** configuration pane, scroll down to the **Advanced** section and select the **Allow multiple runs** option to enable more precise event data collection or deselect the option to use event multiplexing.

If you enable the multiple run mode, the VTune Profiler runs the data collection several times for each event set. You can easily detect these multiple runs on the Timeline pane: they are separated with the grayed out **paused** areas.

The multiple run mode affects the metrics calculation. All "total" types of metrics (Total Time, Elapsed Time) are calculated for the whole analysis session that includes multiple runs while all other metrics are provided per run.

If you want to avoid running the application multiple times but get more accurate multiplexing data, you need to create a custom analysis and enable the **Use precise multiplexing** option available for the custom hardware event-based sampling analysis configuration. This option enables a multiplexing algorithm that

switches event groups on each sample. This mode provides more reliable statistics for applications with a short execution time. You may also consider enabling the precise multiplexing if the [MUX Reliability](#) metric for the Microarchitecture Exploration analysis result is low.

See Also

[allow-multiple-runs](#)

vtune option

[Custom Analysis Options](#)

Problem: 'Events= Sample After Value (SAV) * Samples' Is Not True If Multiple Runs Are Disabled

[Set Up Analysis Target](#)

Hardware Event-based Sampling Collection with Stacks

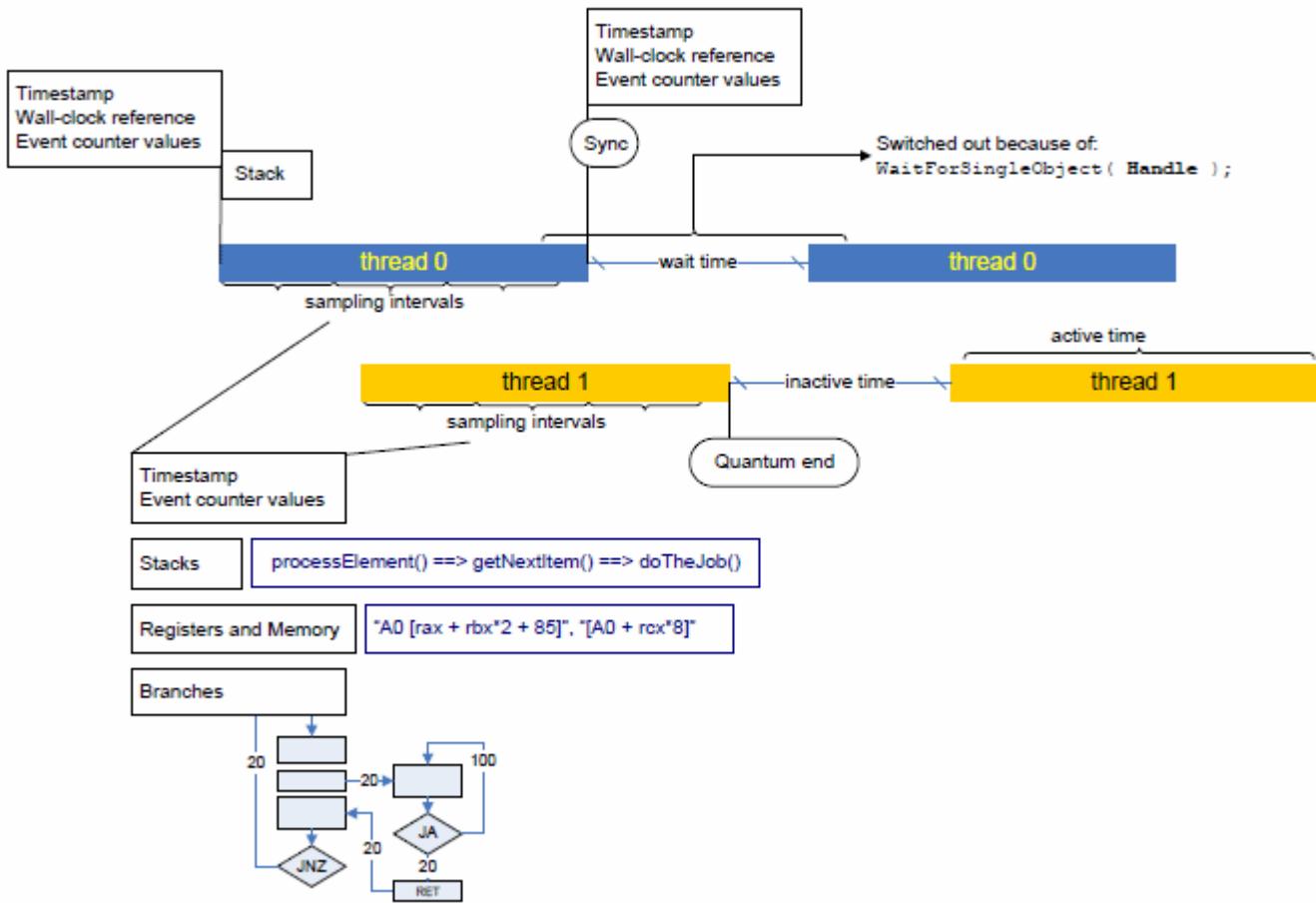
Configure the event-based sampling collector to analyze call stacks for your functions and identify performance, parallelism and power consumption issues.

NOTE

For Linux* targets, make sure your [kernel is configured](#) to support event-based stack sampling collection.

Multitask operating systems execute all software threads in time slices (*thread execution quanta*). Intel® VTune™ Profiler handles thread quantum switches and performs all monitoring operations in correlation with the thread quantum layout.

The figure below explains the general idea of per-thread quantum monitoring:



- The profiler gains control whenever a thread gets scheduled on and then off a processor (that is, at thread quantum borders). That enables the profiler to take exact measurements of any hardware performance events or timestamps, as well as collect a call stack to the point where the thread gets activated and inactivated.
- The profiler determines a reason for thread inactivation: it can either be an explicit request for synchronization, or a so-called thread quantum expiration, when the operating system scheduler preempts the current thread to run another, higher-priority one instead.
- The time during which a thread remains inactive is also measured directly and differentiated based on the thread inactivation reason: inactivity caused by a request for synchronization is called Wait time, while inactivity caused by preemption is called Inactive time.

While a thread is active on a processor (inside a quantum), the profiler employs event-based sampling to reconstruct the program logic and associate hardware events and other characteristics with the program code. Unlike the traditional event-based sampling, the profiler upon each sampling interrupt also collects:

- call stack information
- branching information (if configured so)
- processor timestamps

All that allows for statistically reconstructing program execution logic (call and control flow graphs) and tracing threading activity over time, as well as collecting virtually any information related to hardware utilization and performance.

Configure Stack Collection

- Click the



Configure Analysis button on the VTune Profiler toolbar.

The **Configure Analysis** window opens.

2. Specify your analysis system in the **WHERE** pane and your analysis target in the **WHAT** pane.
3. In the **HOW** pane, choose the required event-based sampling analysis type. Typically, you are recommended to start with the Hotspots analysis in the **hardware event-based sampling** mode.
4. Configure collection options, if required. For call stack analysis, consider enabling the **Collect stacks** option.
5. Click the **Start** button at the bottom to run the selected analysis type.

VTune Profiler collects hardware event-based sampling data along with the information on execution paths. You may see the collected results in the **Hardware Events** viewpoint providing performance, parallelism and power consumption data on detected call paths.

NOTE

- The event-based stack sampling data collection cannot be configured for the entire system. You have to specify an application to launch or attach to.
- By default, on Linux* systems, VTune Profiler uses the **driverless Perf*-based mode** for hardware event-based collection with stacks. To use the driver-based mode, set the **Stack size** option to 0 (unlimited).
- Call stack analysis adds an overhead to your data collection. To minimize the overhead incurred with the stack size, use the **Stack size** option in the custom hardware event-based sampling configuration or **-stack-size** knob from CLI to limit the size of a raw stack. By default, on Linux a stack size of 1024 bytes is collected. On Windows, by default, a full size stack is collected (zero size value). If you disable this option, the overhead will be also reduced but no stack data will be collected.

Analyze Performance

Select the **Hardware Events** viewpoint and click the **Event Count** tab. By default, the data in the grid are sorted by the Clockticks (**CPU_CLK_UNHALTED**) event count providing primary hotspots on top of the list.

Click the plus sign to expand each hotspot node (a function, by default) into a series of call paths, along which the hotspot was executed. VTune Profiler decomposes all hardware events per call path based on the frequency of the path execution.

Hardware Event Count by Hardware Event Type				
Function / Call Stack	INST RETIRED ANY	CPU CLK UNHALTED THREAD	CPU CLK UNHALTED/REF ...	Sample Count
CpuSyscallStub	25,700,419,203	15,966,041,052	5,435,344	
▶ WaitForSingleObjectEx ←	24,850,016,469	15,709,001,086	5,419,254	
▶ ConsoleCallServerGeneric	826,642,385	235,075,235	0	
▶ WriteFile ← write	11,440,420	21,964,731	16,090	
▶ [Unknown stack frame(s)]	8,595,287	0	0	
▶ ReadFile ← read ← filbuf ←	3,724,642	0	0	

The counts of the hardware events of all execution paths leading to a sampled node sum up to the event count of that node. For example, for the **CpuSyscallStub** function, which is the top hotspot of the application, the **INST_RETIRIED.ANY** event count equals the sum of event counts for all 5 calling sequences: 25 700 419 203.

Such a decomposition is extremely important if a hotspot is in a third-party library function whose code cannot be modified, or whose behavior depends on input parameters. In this case the only way of optimization is analyzing the callers and eliminating excessive invocations of the function, or learning which parameters/conditions cause most of the performance degradation.

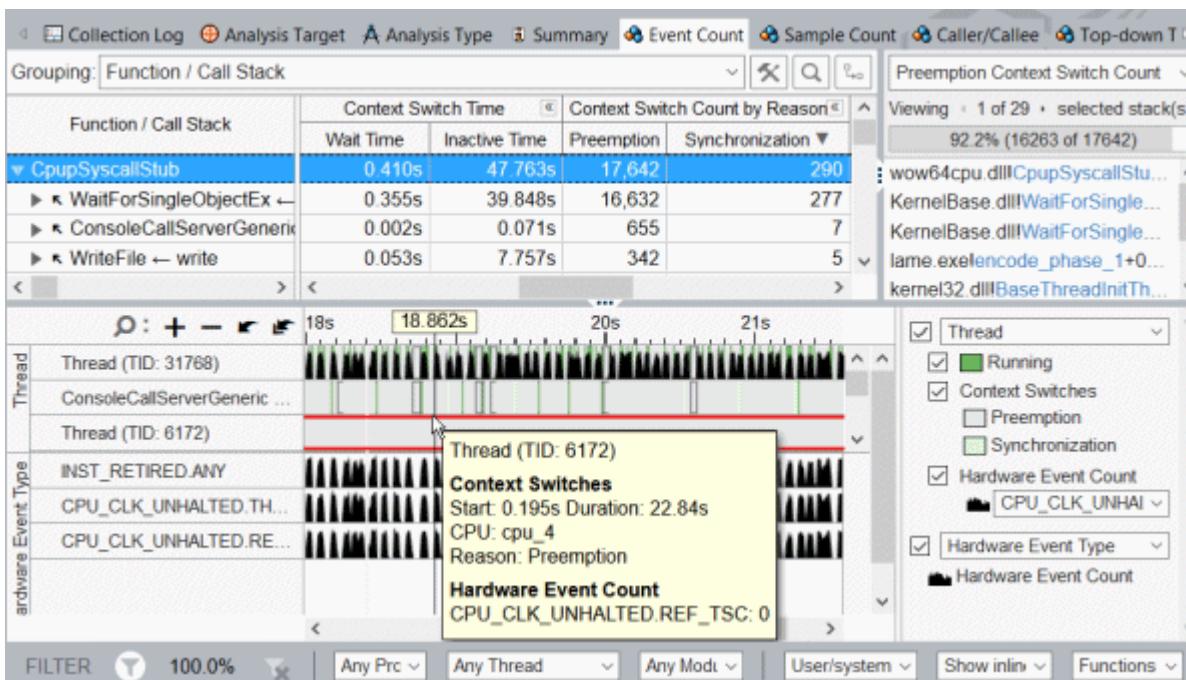
Explore Parallelism

When the call stacks collection is enabled (for example, **Collect stacks** option for the Hotspots in the hardware event-based sampling mode), the VTune Profiler analyzes context switches and displays data on the threads activity using the context switch performance metrics.

Click the **Context Switch by Reason > Synchronization** column header to sort the data by this metric. The synchronization hotspots with the highest number of context switches and high Wait time values typically signals a thread contention on this stack.

Function / Call Stack	Context Switch Time		Context Switch Count by Reason	
	Wait Time	Inactive Time	Preemption	Synchronization ▼
CpupSyscallStub	0.410s	47.763s	17,642	290
▶ ↵ WaitForSingleObjectEx	0.355s	39.848s	16,632	277
▶ ↵ ConsoleCallServerGeneric	0.002s	0.071s	655	7
▶ ↵ WriteFile ← write	0.053s	7.757s	342	5
▶ ↵ SleepEx ← Sleep ← ful	0.000s	0.000s	3	1

Select a context switch oriented type of the stack (for example, the **Preemption Context Switch Count** type) in the drop-down menu of the **Call Stack** pane and explore the **Timeline** pane that shows each separate thread execution quantum. A dark-green bar represents a single thread activity quantum, grey bars and light-green bars - thread inactivity periods (context switches). Hover over a context switch region in the **Timeline** pane to view details on its duration, start time and the reason of thread inactivity.



When you select a context switch region in the **Timeline** pane, the **Call Stack** pane displays a call sequence at which a preceding quantum was interrupted.

You may also select a hardware or software event from the Timeline drop-down menu and see how the event maps to the thread activity quanta (or to the inactivity periods).

Correlate data you obtained during the performance and parallelism analysis. Those execution paths that are listed as the performance hotspots with the highest event count and as the synchronization hotspots are obvious candidates for optimization. Your next step could be analyzing power metrics to understand the cost of such a synchronization scheme in terms of energy.

NOTE

- For analyses using the [Perf*-based driverless collection](#), the types of context switches (preemption or synchronization) may not be identified on kernels older than 4.17 and the following metrics may not be available: Wait time, Wait Rate, Inactive Time, Preemption and Synchronization Context Switch Count.
 - The speed at which the data is generated (proportional to the sampling frequency and the intensity of thread synchronization/contention) may become greater than the speed at which the data is being saved to a trace file, so the profiler will try to adapt the incoming data rate to the outgoing data rate by not letting threads of a program being profiled be scheduled for execution. This will cause paused regions to appear on the timeline, even if no pause was explicitly requested. In ultimate cases, when this procedure fails to limit the incoming data rate, the profiler will begin losing sample records, but will still keep the counts of hardware events. If such a situation occurs, the hardware event counts of lost sample records will be attributed to a special node: **[Events Lost on Trace Overflow]**.
-

See Also

[knob enable-stack-collection=true](#)

Performance Snapshot

VTune Profiler provides several analysis types that are tailored to examine various application types and aspects of performance. Performance Snapshot captures a picture of these aspects and presents an overview of the workings of your application.

Use Performance Snapshot when you want to see a summary of issues affecting your application. This analysis also includes recommendations for other analysis types that you can run next for a deeper investigation.

Run the Analysis

Before running Performance Snapshot, make sure you [Create a project](#).

1. Click **Configure Analysis** on the VTune Profiler welcome screen. This opens the **Performance Snapshot** analysis type by default. You can also select this analysis from the Analysis Tree.
2. In the **WHAT** pane, specify your target application and any application parameters.
3. In the **HOW** pane, click the Start button () to run the analysis.

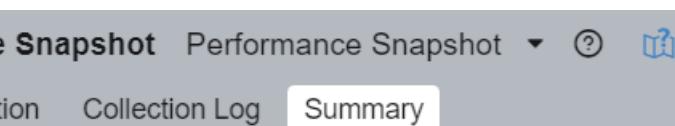
NOTE

To run Performance Snapshot from the command line for this configuration, use the



Command Line button at the bottom.

4. Once the data collection is complete, see a performance overview in the **Summary** tab.
The overview typically includes several metrics along with their descriptions.



Your next analysis type

Get a quick recommendation based on your performance snapshot.

ALGORITHM



Anomaly
Detection
(preview)

PARALLELISM



HPC
Performance
Characterization

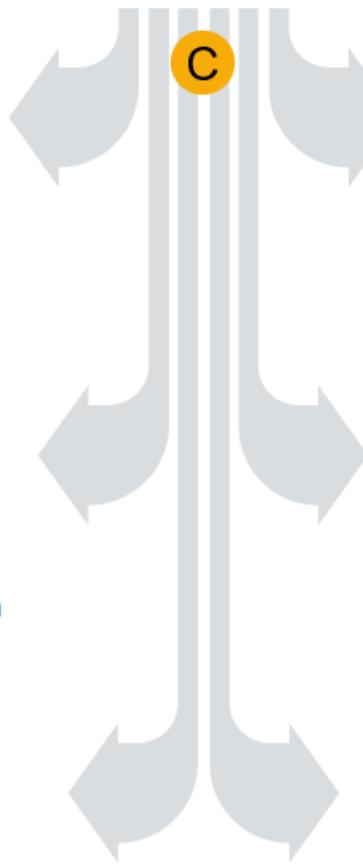
ACCELERATORS



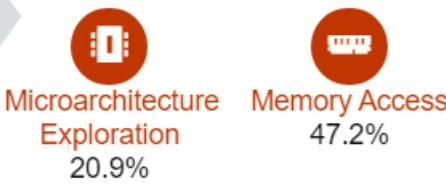
GPU
Compute/Media
Hotspots
(preview)



CPU/FPGA
Interaction
(preview)



MICROARCHITECTURE



I/O



Input and Output
(preview)

PLATFORM ANALYSIS



System
Overview

Throttling
(preview)



Platform Profiler

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

47.2%

A

Elapsed Time ^②: 83.083

IPC ^② :	0.41
SP GFLOPS ^② :	0.00
DP GFLOPS ^② :	0.20
x87 GFLOPS ^② :	0.00
Average CPU Frequency ^② :	2.5 GHz

Effective Logical Core

97.7% (7.815 out of 8)

Microarchitecture Usage of Pipeline Slots

A

Expand each metric for detailed information about contributing factors.

B

A flagged metric indicates a value outside acceptable/normal operating range. Use tool tips to understand how to improve a flagged metric.

Vectorization ^②: 0.3% ↗ of Packed FP Operations

Instruction Mix:	0.3%
SP FLOPs ^② :	0.3%
Packed ^② :	0.3%
128-bit ^② :	0.2%
256-bit ^② :	0.0%



See guidance on other analyses you should consider running next. The **Analysis Tree** highlights these recommendations.

See Also

[Run Command Line Analysis](#)

[Reference](#)

[Run Energy Analysis](#)

Algorithm Group

*The analyses in the **Algorithm** group target software tuning. They help you understand where your application spends the most time. You can also analyze the efficiency of your algorithms.*

The Algorithm group includes these analysis types:

- [Hotspots](#) focuses on a particular target, identifies functions that took the most CPU time to execute, restores the call tree for each function, and shows thread activity.
- [Anomaly Detection](#) analysis helps you identify performance anomalies in frequently recurring intervals of code like loop iterations.
- [Memory Consumption](#) analyzes your Linux* native or Python* targets to explore memory consumption (RAM) over time and identify memory objects allocated and released during the analysis run.

Hotspots Analysis for CPU Usage Issues

Use the Hotspots analysis to understand an application flow and identify sections of code that get a lot of execution time (hotspots). This is a starting point for your algorithm analysis.

Total Time: 133.634s

Collection Time: 472.871s

Thread Count: 5

Critical Time: 0s

Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the **Bottom-up** view for in-depth analysis per function. Or use the **Caller/Callee** view to find critical paths for these functions.

Hotspots

This section lists the most active functions in your application. Optimizing these functions typically results in improving overall application performance.

Module	CPU Time
matrix.exe	472.573s

Hotspots analysis has two sampling-based collection modes:

- **User-mode sampling**, which incurs higher overhead but does not require sampling drivers for collection. Starting with Intel® VTune™ Amplifier 2019, this mode replaced the former Basic Hotspots analysis.
- **Hardware event-based sampling**, which provides minimum collection overhead but needs sampling drivers or Perf* to be installed. Starting with VTune Amplifier 2019, this mode replaced the former Advanced Hotspots analysis.

NOTE

Intel® VTune™ Profiler is a new renamed version of Intel® VTune™ Amplifier.

How It Works: User-Mode Sampling

VTune Profiler uses a low overhead (about 5%) **user-mode sampling and tracing collection** that gets you the information you need without slowing down application execution significantly. The data collector profiles your application using the OS timer, interrupts a process, collects samples of all active instruction addresses with the sampling interval of 10ms, and captures a call sequence (stack) for each sample. VTune Profiler stores the sampled instruction pointer (IP) along with a call sequence in data collection files, and then analyzes and displays this data in a result tab. Statistically collected IP samples with call sequences enable the VTune Profiler to display a top-down tree (call tree). Use this data to understand the control flow for statistically important code sections.

In the user-mode sampling, the collector does not gather system-wide performance data but focuses on your application only. To analyze system performance, use the hardware event-based sampling mode.

VTune Profiler displays a list of functions in your application ordered by the amount of time spent in each function. It also captures the call stacks for each of these functions so you can see how the hot functions are called.

A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hotspots can be removed, while other hotspots are fundamental to the application functionality and cannot be removed.

How It Works: Hardware Event-Based Sampling

The hardware event-based sampling mode is based on the [hardware event-based sampling collection](#) and analyzes all the processes running on your system at the moment, providing CPU time data on whole system performance. VTune Profiler creates a list of functions in your application ordered by the amount of time spent in each function. By default, the Hotspots analysis in the hardware event-based sampling mode does not capture the function call stacks as the hotspots are collected. But you still can analyze stacks for your application modules by selecting the **Collect stacks** option explicitly.

NOTE

- If you cannot run the hardware event-based sampling with stacks, disable the **Collect stacks** option and run the collection. To correlate the obtained hardware event-based sampling data with stacks, run a separate Hotspots analysis in the User-Mode Sampling mode.
- On 32-bit Linux* systems, the VTune Profiler uses a [driverless Perf*-based collection](#) for the hardware event-based sampling mode.

Configure and Run Analysis

To configure and run the Hotspots analysis:

Prerequisites: Create a project.

1. Click the



(standalone GUI)/



(Visual Studio IDE) **Configure Analysis** button on the VTune Profiler welcome screen.

2. In the **HOW** pane, select the **Hotspots** analysis from the Analysis Tree.
3. Configure the following options:

User-Mode Sampling mode	Select to enable the user-mode sampling and tracing collection for hot spots and call stack analysis (formerly known as Basic Hotspots). This collection mode uses a fixed sampling interval of 10ms. If you need to change the interval, click the Copy button and create a custom analysis configuration.
Hardware Event-Based Sampling mode	<p>Select to enable hardware event-based sampling collection for Hotspots analysis (formerly known as Advanced Hotspots).</p> <p>You can configure the following options for this collection mode:</p> <ul style="list-style-type: none"> • CPU sampling interval, ms to specify an interval (in milliseconds) between CPU samples. Possible values for the hardware event-based sampling mode are 0.01-1000. 1 ms is used by default. • Collect stacks to enable advanced collection of call stacks and thread context switches.

NOTE

When changing collection options, pay attention to the **Overhead** diagram on the right. It dynamically changes to reflect the collection overhead incurred by the selected options.

Show additional performance insights check box

Get additional performance insights, such as vectorization, and learn next steps. This option collects additional CPU events, which may enable the multiplexing mode.

The option is enabled by default.

Details button

Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable **additional settings** for the analysis, you need to [create a custom configuration](#) by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

4. Click the



Start button to run the analysis.

NOTE

To [generate the command line](#) for this configuration, click the **Command Line...** button at the bottom.

View Data

When the data is collected, VTune Profiler opens it in the **Hotspots by CPU Utilization** viewpoint providing the following views for analysis:

- [Summary window](#) displays statistics on the overall application execution to analyze CPU time and processor utilization.
- [Bottom-up window](#) displays hotspot functions in the bottom-up tree, CPU time and CPU utilization per function.
- [Top-down Tree window](#) displays hotspot functions in the call tree, performance metrics for a function only (Self value) and for a function and its children together (Total value).
- [Caller/Callee window](#) displays parent and child functions of the selected focus function.
- [Platform window](#) provides details on CPU and GPU utilization, frame rate, memory bandwidth, and user tasks (if corresponding metrics are collected).

What's Next

1. [Identify the most time-consuming function](#) in the grid and double-click it for source analysis.
2. [Analyze the source](#) of the critical function starting with the highlighted hottest code line and moving further with the Hotspot Navigation options.
3. Modify your code to remove bottlenecks and improve the performance of your application.
4. Re-run the analysis and verify your optimization with the [comparison mode](#).
5. Fix vectorization and get code-specific recommendations with the **Vectorization and Code Insights perspective** in Intel® Advisor.

For further steps, explore the **Insights** section provided in the **Summary** window. This section contains information on your target performance against metrics collected in addition to standard hotspots metrics. If there are any performance issues detected, the VTune Profiler flags such a metric value and provides an insight on potential next steps to fix the problem.

Information provided by Hotspots analysis is important for tuning serial applications as well as the serial sections of parallel applications. The Hotspots analysis data helps you understand what your application is doing and identify the code that is critical to tune. For parallel applications running on multi-core systems, consider running the **Threading** or **HPC Performance Characterization** analyses in addition..

See Also

collect

hotspots vtune option

[Offload and Optimize OpenMP* Applications with Intel Tools](#)

[Tutorial: Analyze Common Performance Bottlenecks on Linux* - C++ Sample Code](#)

[Tutorial: Analyze Common Performance Bottlenecks on Windows* - C++ Sample Code](#)

[Tutorial: Analyze Common Performance Bottlenecks on Windows* - C++ Sample Code](#)

[Fix Vectorization and Get Code Insights](#)

Hotspots View

Identify program units that took the most CPU time.

These are recognized as hotspots. The Hotspots viewpoint is available for all analysis results.

Follow these steps to interpret performance data available in the Hotspots viewpoint:

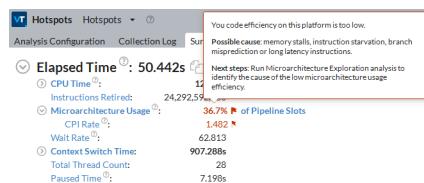
1. Define a performance baseline.
2. Identify the hottest function.
3. Identify algorithm issues.
4. Analyze source.
5. Explore other analysis types.

Define a Performance Baseline

Start your analysis in the [Summary window](#). Here you see general information about the execution of your application. Note that the Elapsed time is different from the application CPU time. The Elapsed time is the application time from start to termination. The application CPU time is the sum of the active processor time for all the threads that run the application. It does not include waiting times.

Use the Elapsed time value as a baseline to compare versions before and after optimization. When tuning the application, as you add more threads, the Elapsed time tends to decrease whereas the CPU time may increase.

If you ran the Hotspots analysis in the hardware event-based sampling mode, the analysis metrics in the [Summary](#) window display the [Microarchitecture Usage](#) metric. Use this metric to estimate the code efficiency on your hardware platform:



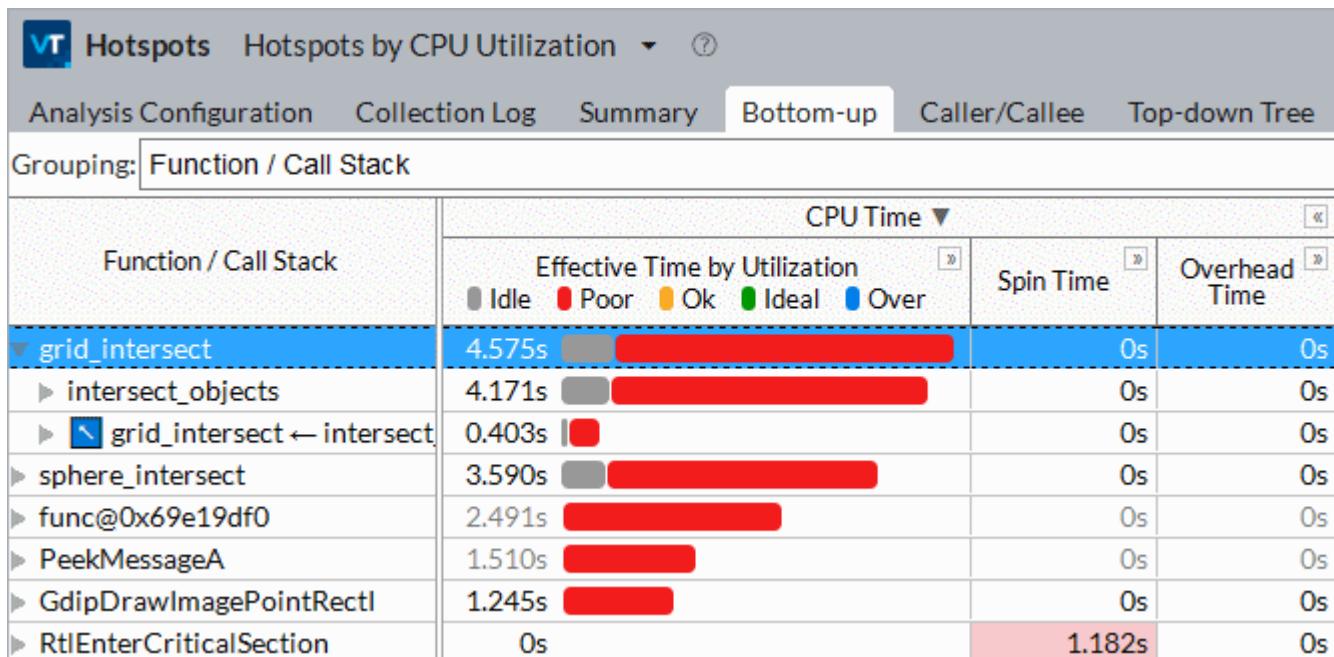
If this metric value is flagged as critical, consider running the [Microarchitecture Exploration](#) analysis to dive deeper into hardware metrics.

Identify the Hottest Function

Get a list of the most time-consuming functions in the [Top Hotspots](#) section of the [Summary](#) window. Click on a hotspot function to explore its call flow and other related metrics in the [Bottom-up view](#).

By default, the **Bottom-up** view presents a sorted display of CPU Time in descending order, starting with the most time-consuming functions. Start optimizing the functions with the largest CPU time.

Expand the **CPU Time** column to get more details on how effectively the CPU time was used:



Next, focus your tuning efforts on the program units with the largest **Poor** value. This means that your application underutilized the CPU time during the execution of these program units. The overall goal of optimization is to achieve **Ideal** (green)



) or **OK** (orange)

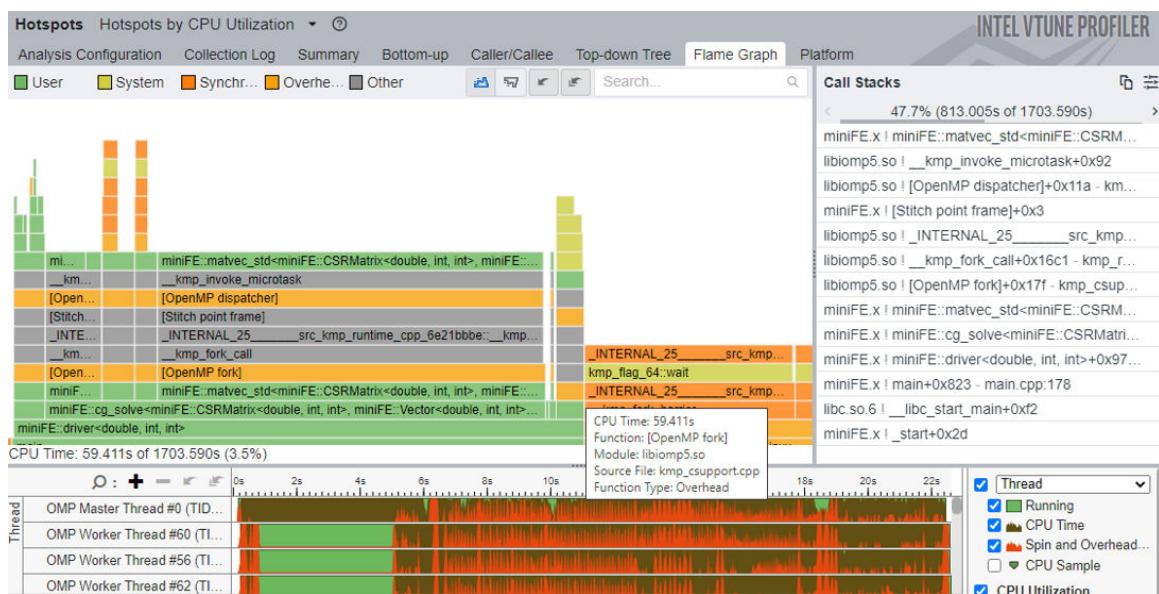


) CPU utilization state and shorten the **Poor** and **Over** CPU utilization values.

Identify Hot Code Paths

Switch to the [Flame Graph window](#) to quickly identify the hottest code paths in your application. Analyze the CPU time spent on each program unit and its related callee functions.

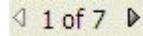
The flame graph plots stack profile population (sorted alphabetically) on the horizontal axis. The vertical axis shows stack depth, starting from zero at the bottom. The width of each element in the flame graph indicates the percentage of CPU time of the function (and its callees) to the total CPU time.



Identify Algorithm Issues

If you identify issues with the calling sequences in your application, you can improve performance by revising the order in which functions are called. Use these methods:

- **Top-down Tree pane:** Analyze the Total and Self time data for callers and callees of the hotspot function to understand whether this time can be optimized.
- **Call Stack pane:** Identify the highest contributing stack for the program unit(s) selected in the **Bottom-up** or **Top-down Tree** panes. Use the navigation buttons



to see the different stacks that called the selected program unit(s). The contribution bar shows the contribution of the currently visible stack to the overall time spent by the selected program unit(s). You can also use the drop-down list in the **Call Stack** pane to view data for different types of stacks.

NOTE

Stack data is available by default for the user-mode sampling mode. To have this data for the hardware event-based sampling mode, you need to enable the **Collect stacks** option in the Hotspots analysis configuration.

Analyze Source

Double-click the hottest function to view its related source code in the Source/Assembly window. Open the code editor directly from Intel® VTune™ Profiler and improve your code (for example, minimizing the number of calls to the hotspot function).

What's Next

If you ran the analysis with the default **Show additional performance insights** option, the **Summary** view will include the **Insights** section that provides additional metrics for your target such as efficiency of the hardware usage and vectorization. This information helps you identify potential next steps for your performance analysis and understand where you could focus your optimization efforts.

INTEL VTUNE PROFILER

INSIGHTS

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the [Bottom-up](#) view for in-depth analysis per function. Otherwise, use the [Caller/Callee](#) view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism  : **14.2%** 

Use [Threading](#) to explore more opportunities to increase parallelism in your application.

Microarchitecture Usage  : **0.0%** 

Use [Microarchitecture Exploration](#) to explore how efficiently your application runs on the used hardware.

Related information

- An explanation of Flame Graphs
- Flame Graph Window
- Source Code Analysis
- View Stacks
- Reference

Anomaly Detection Analysis (preview)

Use Anomaly Detection to identify performance anomalies in frequently recurring intervals of code like loop iterations. Perform fine-grained analysis at the microsecond and nanosecond level.

Application performance can occasionally be hampered by the presence of performance anomalies. A performance anomaly is any short-lived, sporadic issue that causes unrecoverable consequences. These issues may not be statistically discernible but they create a poor user experience and can be very expensive to fix. When the performance of your application requires varying amounts of work for instances of the same task or when it displays variations in a single/few iterations of a loop, these are symptoms of anomalous behavior in your application.

Use Anomaly Detection analysis to identify performance anomalies in your application that are otherwise difficult to isolate. This analysis type uses Intel® Processor Trace (Intel® PT) technology to perform trace data collection and fine-grained time and event measurement. Intel® PT is an extension of Intel® Architecture that captures information about software execution using dedicated hardware. The hardware causes only minimal performance perturbation to the software being traced.

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

The control flow trace feature in Intel® PT generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks). For descriptions of key concepts in Intel® PT, see [Chapter 35 of the Intel Software Developer's Manual \(Volume 3C\):System Programming Guide](#).

To detect software performance anomalies using VTune Profiler, you use the [Instrumentation and Tracing Technology \(ITT\) API](#) to designate specific code regions of interest and then run Anomaly Detection analysis.

Common Performance Anomalies

These are typical examples of performance anomalies in a software application.

- Financial transactions that take an unusually long time to process.
- Glitches in the UI of a video game like slow or skipped video frames.
- Packet losses in large applications that have SPDK/DPDK loops.
- High frequency applications where processing speed is critical and some iterations run slower than others.

Run Anomaly Detection in one of these situations where observed application behavior deviates from expected behavior in some iterations.

Causes for Performance Anomaly

- **Change in control flow:** Different instances of the same task require different amounts of work.
- **Uncommon observations:** Expensive handling of errors or memory/storage reallocation.
- **Context switches:** Synchronization or preemption.
- **Unexpected kernel activity:** Interrupts or page faults.
- **Micro-architectural issues:** Cache misses or incorrect branch predictions.
- **Frequency drops:** Low CPU utilization, cooling issues, or the inclusion of Intel® Advanced Vector Extensions (Intel® AVX) instructions in the code.

The Anomaly Detection Analysis Workflow

When you observe anomalies in your application performance, use Anomaly Detection for a detailed investigation.

1. Prepare your application for analysis.
2. Define parameters that break your code into smaller regions of interest. Decide how long you want to simulate each region.
3. Run Anomaly Detection.

4. Review anomalies in detail:

- a.** Load trace data for the processor for each anomaly in the Bottom-up view to examine code regions of interest.
- b.** Open trace data to see frequency information for a specific region.
- c.** Examine source and assembly views to see the number of loop iterations.

Configure and Run Analysis

Prepare your Application

Large applications can generate huge volumes of data through a profiling run. This in turn can cause significant delay in processing results. You may only want to focus on anomalies in a particular operation in your code. Mark this section by defining it as a Code Region of Interest. Use the ITT API for this purpose.

1. Register the name of the code region you plan to profile:

```
_itt_pt_region region = _itt_pt_region_create("region");
```

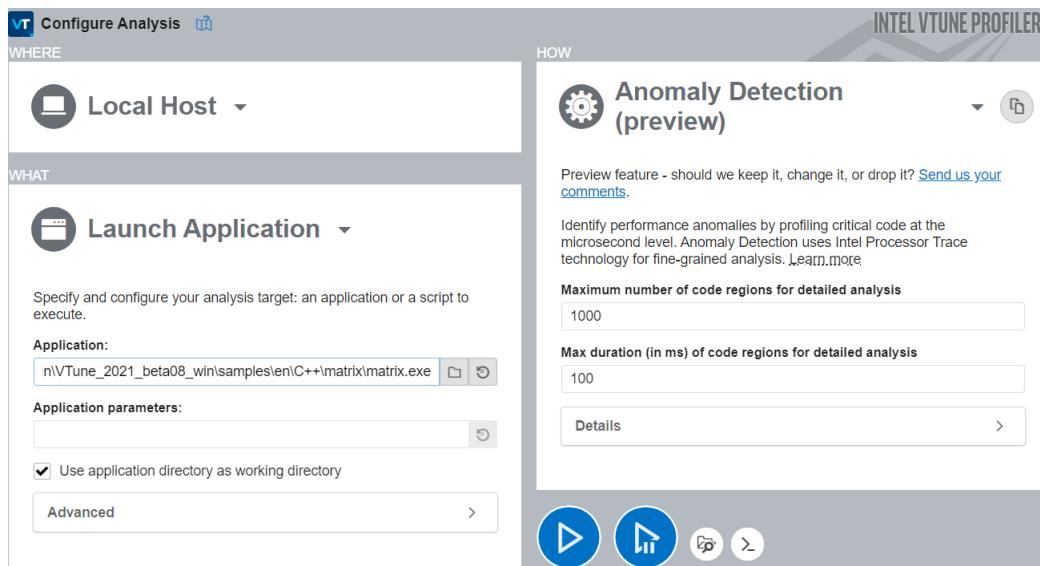
2. Mark the target loop in your application with this name:

```
for(...;...;...)
{
    _itt_mark_pt_region_begin(region);
    <code processing your task>
    _itt_mark_pt_region_end(region);
}
```

Run the Analysis

- 1.** On the Welcome screen, click **Configure Analysis**.
- 2.** In the Analysis Tree, select the **Anomaly Detection** analysis type in the **Algorithm** group.
- 3.** In the **WHAT** pane, specify your application and any relevant application parameters.
- 4.** In the **HOW** pane, specify parameters for the analysis.

Parameter	Description	Range	Recommended Value
<i>Maximum number of code regions for detailed analysis</i>	Specify number of code regions for your application.	10-5000	For faster loading of details, pick a value not more than 1000.
<i>Maximum duration of analysis per code region</i>	Specify the duration of analysis time (ms) to be spent on each code region.	0.001-1000	Any value under 1000 ms.



5. Click the



Start button to run the analysis.

NOTE

To [run Anomaly Detection from the command line](#), use the



Command Line button at the bottom.

View Data

Once the analysis is complete, VTune Profiler displays results in the **Summary** window.

- **Elapsed Time** indicates the total time spent on all code regions of interest.
- **Code Region of Interest Duration Histogram** plots the number of instances of performance-critical tasks against specified duration (or latency). See specific code regions in the Fast and Slow regions to understand why the duration changed.
- **Collection and Platform Info** displays relevant details about the system, data about the collection platform, and the resulting set size.

View Data on a Different System

The above procedure is useful when you process analysis results on the same system where you collected data. If you want to transfer the collected data onto a different system before you view it, run the archive command after data collection to copy essential binaries to the results folder. You must complete this step before transferring results to the new system to load collection details without problems.

To run the archive command:

1. Collect results as described above.
2. At the command line, type:

```
vtune.exe --archive -r r001ad
```

where r001ad is an example of an analysis result.

NOTE

To view collected data on a different system, you must copy all binaries including system and compiler runtime binaries that are linked to your main binary and were accessed during the collection. The archive command is useful for this purpose since it is not easy to copy these binaries manually.

Next Steps

See the [Anomaly Detection view](#) for information on interpreting collected data in these ways:

- Load trace details for each analysis in the **Bottom-up** window.
- Look for unexpected kernel activity. See if applications entered certain kernels that should not have been activated during the analysis.
- Use the source and assembly views to compare code regions of interest in fast and slow regions of the histogram.

See Also

[Anomaly Detection View](#) Interpret results after performing Anomaly Detection analysis on your application. Identify performance anomalies by examining code regions of interest.

Anomaly Detection View

Interpret results after performing Anomaly Detection analysis on your application. Identify performance anomalies by examining code regions of interest.

Use the Anomaly Detection view to interpret the results of an Anomaly Detection analysis. A typical workflow involves an examination in these areas:

View Data

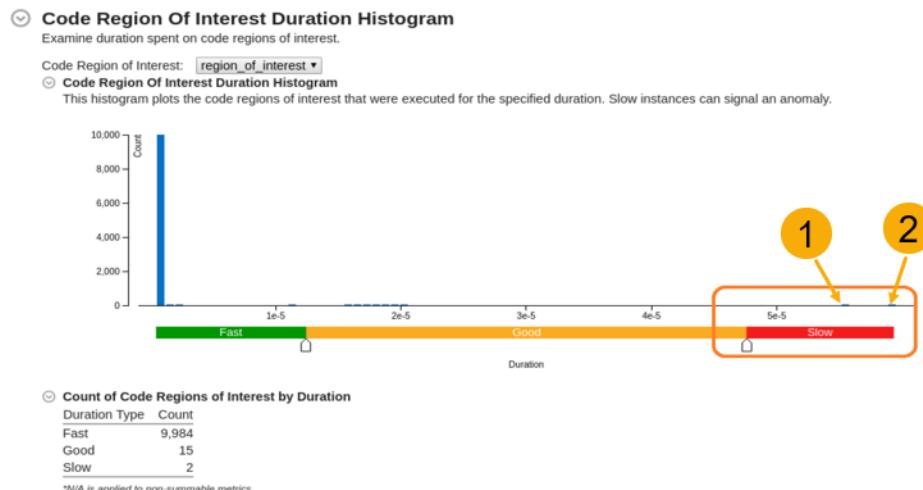
Once you complete running [Anomaly Detection](#) on your application, the collected data displays in the **Summary** window.

Start with the **Code Region of Interest Duration Histogram**. This shows the number of instances of a performance-critical task for a specific duration or latency (in ms).

Examine the histogram to see:

- Code regions of interest
- Information about regions where simulations executed faster or slower than normal

This diagram identifies unexpected performance outliers in the Slow region.



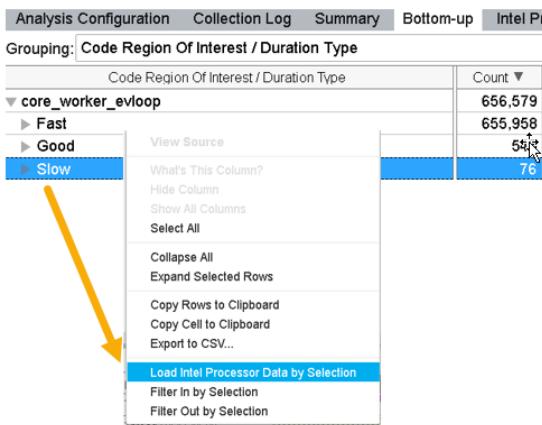
NOTE If necessary, use the sliders on the X-axis to adjust the thresholds for Fast, Good, and Slow latencies.

Load Details for Slow Region

In the **Bottom-up** window, load details for the slow code regions of interest:

1. Switch to the **Bottom-up** window.
2. Group results by **Code Region of Interest / Duration Type**.
3. To further examine the outliers in the Slow region, right click on the Slow field and select **Load Intel Processor Data by Selection**.

This loads details about the code regions of interest in the **Intel Processor Trace Details** window.



Compare Processor Trace Details

Once you load trace data in the **Intel Processor Trace Details** window, you can compare trace details of individual instances of marked code regions by placing them side by side. The top of a stack represents the **kernel entry point**.

Metric	Interpretation
Instructions Retired, Call Count, Total Iteration Count	Control flow metrics. Instructions Retired refers to the number of entries into a kernel.
CPU Time (Kernel and User)	Active time on the CPU.
Wait Time, Inactive Time	Duration for which a thread was idle because of synchronization or preemption.
Elapsed Time	Latency (Wall-clock time of the code region execution).

Use this window as a hub to detect the following types of performance anomalies.

- Context Switch Anomaly
- Kernel-Induced Anomaly
- Frequency Drops
- Control Flow Deviation Anomaly

Context Switch Anomaly

1. In the **Intel Processor Trace Details** window, check the **Inactive Time** and **Wait Time** metrics. The **Wait Time** indicates the duration for which a thread was idle due to synchronization issues.
 - a. If the metrics are zero, the application had no context switches. Proceed to check for a different type of anomaly.
 - b. If the metrics are non-zero, continue with this procedure to check for context switches.
2. Sort the data by **Wait Time**.
3. For the instances that had significant **Wait Time**, compare the **Wait Time** with **Elapsed Time**. If the thread was idle for a considerable portion of elapsed time, this was due to a context switch synchronization issue. In this example, thread 25883 was idle for 1.269 out of 1.318 milliseconds, which is about 96% of the time.

Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time ▼	Inactive Time
					Kernel	User		
core_worker_evloop	20,268,375	329,210	489,045	98.659ms	42.631ms	16.882ms	4.724ms	0ms
▶ 25883	3,082	54	64	1.318ms	0.029ms	0.002ms	1.269ms	0ms
▶ 60215	3,110	55	65	1.240ms	0.030ms	0.004ms	1.209ms	0ms
▶ 276245	3,082	54	64	1.175ms	0.014ms	0.002ms	1.143ms	0ms
▶ 498819	3,082	54	64	1.005ms	0.016ms	0.003ms	0.988ms	0ms
▶ 558496	448,129	10,543	16,527	1.009ms	0.024ms	1.328ms	0.080ms	0ms
▶ 26851	447,762	10,530	16,480	1.057ms	0.014ms	0.769ms	0.035ms	0ms
▶ 484307	252,503	3,872	5,682	1.088ms	0.619ms	0.181ms	0ms	0ms
▶ 484306	452,450	10,641	16,607	1.049ms	0.020ms	0.750ms	0ms	0ms

4. Expand the instance to drill down to a function or a stack. Identify the stack(s) that brought the thread to an idle state.

Kernel-Induced Anomaly

1. In the **Intel Processor Trace Details** window, sort the data by **Kernel Time**. The topmost element of the stack points to the entry point into the kernel. Where the ratio of kernel time to Elapsed Time is high, a significant amount of time was spent in the kernel. In this example, 566 out of 997 microseconds were spent in the kernel for the highlighted thread.

Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time	Inactive Time
					Kernel ▼	User		
core_worker_evloop	19,724,614	320,878	476,973	93.977ms	37.874ms	16.347ms	4.724ms	0ms
▶ 436053	242,474	3,706	5,450	0.997ms	0.566ms	0.176ms	0ms	0ms
▶ 412197	242,907	3,702	5,455	1.244ms	0.519ms	0.134ms	0ms	0ms
▶ 551160	227,194	3,549	5,224	1.088ms	0.519ms	0.144ms	0ms	0ms
▶ 471743	227,295	3,538	5,218	1.077ms	0.519ms	0.138ms	0ms	0ms
▶ 527220	231,793	3,535	5,191	1.145ms	0.516ms	0.132ms	0ms	0ms

2. Expand the thread to see contributing stacks that could be responsible for long kernel times.

Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time	Inactive Time
					Kernel ▼	User		
core_worker_evloop	242,474	3,706	5,450	0.997ms	0.566ms	0.176ms	0ms	0ms
▶ 436053	173	0	0		0.566ms	0ms	0ms	0ms
▶ [kernel activity]								
▶ write ← tcp_send ← buf_tcp_write ← _worker_event_w	84	0	0		0.399ms	0ms	0ms	0ms
▶ read ← tcp_recv ← buf_tcp_read ← _worker_event_r	85	0	0		0.137ms	0ms	0ms	0ms
▶ epoll_pwait ← [Loop at line 218 in event_wait] ← event	1	0	0		0.024ms	0ms	0ms	0ms
▶ [Loop at line 128 in hashtable_get] ← hashtable_get ← i	1	0	0		0.005ms	0ms	0ms	0ms
▶ tz_convert ← _klog_write ← [Loop at line 786 in tmem]	1	0	0		0.002ms	0ms	0ms	0ms
▶ clock_gettime ← gettimeofday ← duration_snapshot ← time	1	0	0		0.000ms	0ms	0ms	0ms

Due to the presence of dynamic code in the kernel and drivers, it is not possible to perform static processing of these binaries. The `kernel_activity` node at the top of the stack aggregates all performance data for kernel activity that happened during a specific instance of the Code Region of Interest.

Since kernel binaries are not processed, VTune Profiler cannot collect code flow metrics like **Call Count**, **Iteration Count**, or **Instructions Retired**. All these metrics are zero, except **Instructions Retired**, which indicates the number of entries into the kernel.

A possible explanation for a kernel-induced anomaly could be network speed. This could cause a slowdown when control goes to the kernel while receiving a request and sending a response over the network.

Frequency Drops

Find information about frequency drops in one of these windows:

- **Bottom-up window:** Shows frequency information for the entire application.
- **Intel Processor Trace Details window:** Shows frequency information only for the loaded region.

Frequency drops can happen due to several reasons:

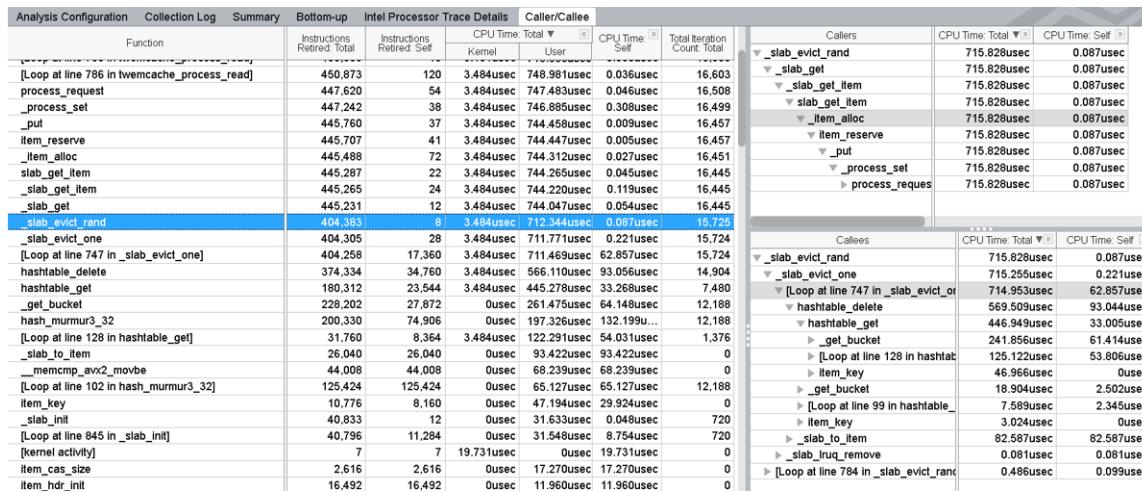
- There are Intel® Advanced Vector Extensions (Intel® AVX) instructions used inside or outside a loaded code region.
- There are underlying hardware issues like cooling.
- Apart from your application, low activity on the core and OS can also cause frequency drops. Look for high numbers of **Inactive Time** or **Wait Time**.

Control Flow Deviation Anomaly

When the **Instructions Retired** metric is unexpectedly huge for some threads, it indicates a control flow anomaly. A code deviation could have happened during execution of the code region.

Code Region Of Interest / Code Region Of Interest (Instance) / Function / Call Stack	Instructions Retired ▾	Call Count	Total Iteration Count	Elapsed Time	CPU Time		Wait Time	Inactive Time
					Kernel	User		
core_worker_evloop	18,142,047	285,483	421,973	92.195ms	40.361ms	12.354ms	4.690ms	0ms
484306	452,450	10,641	16,607	1.049ms	0.020ms	0.750ms	0ms	0ms
558496	448,129	10,543	16,527	1.009ms	0.024ms	1.328ms	0.080ms	0ms
484307	252,503	3,872	5,682	1.088ms	0.619ms	0.181ms	0ms	0ms
348416	238,411	3,645	5,363	1.140ms	0.503ms	0.143ms	0ms	0ms
551698	238,040	3,626	5,349	1.068ms	0.508ms	0.144ms	0ms	0ms
436050	237,851	3,640	5,344	1.150ms	0.508ms	0.144ms	0ms	0ms
524813	236,950	3,620	5,320	1.149ms	0.508ms	0.143ms	0ms	0ms

1. Select a node in the grid where you see a high value for **Instructions Retired**.
2. Right-click and select **Filter In by Selection** from the context menu.
3. Switch to the **Caller/Callee** window.



In the flat profile view, you can see functions annotated with Self and Total CPU Times. The caller view shows the callers of the selected function in a bottom-up representation. The callee view shows a call tree from the selected function in a top-down representation.

In this example, the function call to `_slab_evict_one` function from `_slab_evict_rand` causes a significant delay as evidenced by the Self CPU Time.

Source Code Analysis:

This is an alternative method to identify deviations in the control flow.

1. Check the **Total Iteration Count** to compare the number of loop iterations between a fast and slow iteration.

2. If the slower iteration has a higher iteration count, switch to **Source Assembly** view and examine the source code of the function.
3. Check to see if the slower iteration passed the validation of the cached element.

Both of these methods indicate the presence of a **Cache Eviction**, which can occur infrequently. While you may not be able to eliminate cache evictions entirely, you can minimize them through these ways:

- Increase the cache size.
- Update cache data and repeat the analysis.

See Also

[Anomaly Detection Analysis](#)

[Analyze Performance](#)

Memory Consumption Analysis

Use the Memory Consumption analysis for your Linux native or Python* targets to explore memory consumption (RAM) over time and identify memory objects allocated and released during the analysis run.*

How It Works

Top Memory-Consuming Functions					
Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module	
foo	8 GB	0 B	20,030	test.py	
_nl_load_locale_from_archive	94 MB	94 MB	2	libc.so.6	
list_resize	4 MB	0 B	323	python2.7	
data_stack_grow	3 MB	0 B	3,374	python2.7	
new_arena	1 MB	1 MB	5	python2.7	
[Others]	3 MB	505 KB	3,567		

*N/A is applied to non-summable metrics

During Memory Consumption analysis, the VTune Profiler data collector intercepts memory allocation and deallocation events and captures a call sequence (stack) for each allocation event (for deallocation, only a function that released the memory is captured). VTune Profiler stores the calling instruction pointer (IP) along with a call sequence in data collection files, and then analyzes and displays this data in a result tab.

Configure and Run Analysis

To configure and run the Memory Consumption analysis:

Prerequisites: Create a project.

1. Click the



(standalone GUI)/

(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From **HOW** pane, click the



Browse button and select **Memory Consumption**.

The Memory Consumption analysis is pre-configured to collect data at the memory objects (data structures) granularity, which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.

3. Optionally, you may configure the **Minimal dynamic memory object size to track** option. This option helps reduce runtime overhead of the instrumentation. The default value is 32 bytes.

4. Click the



Start button to [run the analysis](#).

NOTE

Generate the command line for this configuration using the



Command Line button at the bottom.

View Data

By default, the analysis result opens in the [Memory Consumption viewpoint](#). Identify peaks of the memory consumption on the Timeline pane and analyze allocation stacks for the hotspot functions. Double-click a hotspot function to switch to the Source view and analyze the source lines allocating a high amount of memory.

See Also

[Memory Consumption and Allocations View](#)

[Minimize Collection Overhead](#)

collect

memory-consumption vtune option

Memory Consumption and Allocations View

Explore the data collected with the [Memory Consumption analysis](#) for your native or Python* target and identify the most memory-consuming functions, analyze their allocation stacks and source.

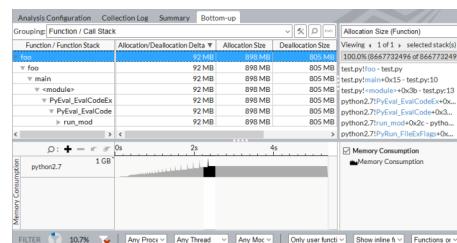
Start with the [Summary window](#) that displays a list of top memory-consuming functions.

For example, the `foo` function has the highest Memory Consumption metric value and could be a candidate for optimization:

Top Memory-Consuming Functions					
Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module	
foo	0 GB	0 B	20,030	test.py	
<code>_ni_load_locale_from_archive</code>	94 MB	94 MB	2	libc.so.6	
<code>list_resize</code>	4 MB	0 B	323	python2.7	
<code>data_stack_grow</code>	3 MB	0 B	3,374	python2.7	
<code>new_arena</code>	1 MB	1 MB	5	python2.7	
[Others]	3 MB	505 KB	3,567		

*N/A is applied to non-summable metrics.

For further investigation, switch to the **Bottom-up** tab and explore the memory consumption distribution over time. Focus on the peak values on the [Timeline](#) pane, select a time range of interest, right click and use the **Filter In by Selection** context menu option to filter in the program units (functions, modules, processes, and so on) executed during this range:



In the example above, the python `foo` function allocated 915 310 048 bytes of memory in a call tree displayed in the **Call Stack** pane on the right but released only 817 830 048 bytes. 92MB is the maximum Allocation/Deallocation delta value that signals a potential memory leak. Clicking the `foo` function opens the [Source view](#) highlighting the code line that allocates the maximum memory. Use this information for deeper code analysis to identify a cause of the memory leaks.

See Also

[Memory Consumption Analysis](#)

[Analyze Performance](#)

Microarchitecture Analysis Group

The **Microarchitecture** analysis group introduces analysis types that help you estimate how effectively your code runs on modern hardware.

- [Microarchitecture Exploration](#) helps identify the most significant hardware issues affecting the performance of your application. Consider this analysis type as a starting point when you do hardware-level analysis.
- [Memory Access](#) measures a set of metrics to identify memory access related issues (for example, specific to NUMA architectures).

Prerequisites:

It is recommended to [install the sampling driver](#) for hardware event-based sampling collection types. For Linux* and Android* targets, if the sampling driver is not installed, VTune Profiler can work on Perf* ([driverless collection](#)). Be aware of the following configuration settings for Linux target systems:

- To enable system-wide and uncore event collection that allows the measurement of DRAM and MCDRAM memory bandwidth that is a part of the Memory Access analysis type, use root or sudo to set `/proc/sys/kernel/perf_event_paranoid` to 0.

```
echo 0>/proc/sys/kernel/perf_event_paranoid
```

- To enable collection with the Microarchitecture Exploration analysis type, increase the default limit of opened file descriptors. Use root or sudo to increase the default value in `/etc/security/limits.conf` to `100*`.

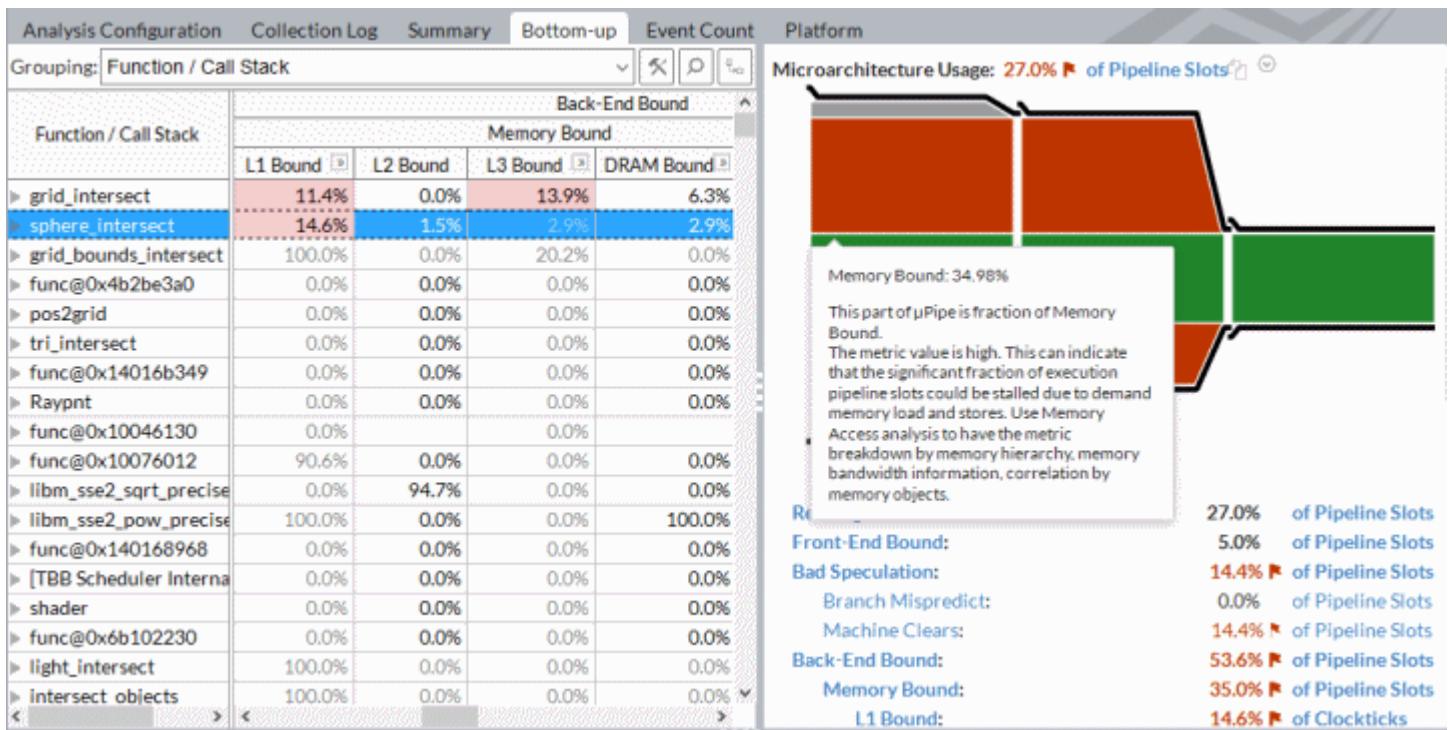
```
<user> hard nofile <100 * number_of_logic_CPU_cores>
```

```
<user> soft nofile <100 * number_of_logic_CPU_cores>
```

Microarchitecture Exploration Analysis for Hardware Issues

Use the [Microarchitecture Exploration analysis](#) (formerly known as [General Exploration](#)) to triage hardware usage issues in your application.

Once you have used Hotspots analysis to determine hotspots in your code, run the Microarchitecture Exploration analysis to understand how efficiently your code is passing through the core pipeline. During Microarchitecture Exploration analysis, VTune Profiler collects a complete list of events for analyzing a typical client application. It calculates a set of predefined ratios used for the metrics and facilitates identifying hardware-level performance problems.



NOTE

Intel® VTune™ Profiler is a new renamed version of Intel® VTune™ Amplifier.

How It Works

The Microarchitecture Exploration analysis strategy varies by microarchitecture. For modern microarchitectures starting with Intel microarchitecture code name Ivy Bridge, the Microarchitecture Exploration analysis is based on the Top-Down Microarchitecture Analysis Method using the [Top-Down Characterization methodology](#), which is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application.

Superscalar processors can be conceptually divided into the front-end, where instructions are fetched and decoded into the operations that constitute them, and the back-end, where the required computation is performed. Each cycle, the front-end generates up to four of these operations. It places them into pipeline slots that then move through the back-end. Thus, for a given execution duration in clock cycles, it is easy to determine the maximum number of pipeline slots containing useful work that can be retired in that duration. The actual number of retired pipeline slots containing useful work, though, rarely equals this maximum. This can be due to several factors: some pipeline slots cannot be filled with useful work, either because the front-end could not fetch or decode instructions in time (Front-end bound execution) or because the back-end was not prepared to accept more operations of a certain kind (Back-end bound execution). Moreover, even pipeline slots that do contain useful work may not retire due to bad speculation. Front-end bound execution may be due to a large code working set, poor code layout, or microcode assists. Back-end bound execution may be due to long-latency operations or other contention for execution resources. Bad speculation is most frequently due to branch misprediction.

Each cycle, each core can fill up to four of its pipeline slots with useful operations. Therefore, for some time interval, it is possible to determine the maximum number of pipeline slots that could have been filled in and issued during that time interval. This analysis performs this estimate and breaks up all pipeline slots into four categories:

- Pipeline slots containing useful work that issued and retired (Retired)

- Pipeline slots containing useful work that issued and cancelled (Bad speculation)
- Pipeline slots that could not be filled with useful work due to problems in the front-end (Front-end Bound)
- Pipeline slots that could not be filled with useful work due to a backup in the back-end (Back-end Bound)

To use Microarchitecture Exploration analysis, first determine which top-level category dominates for hotspots of interest. You can then dive into the dominating category by expanding its column. There, you can find many issues that may contribute to that category.

NOTE

- For a detailed tuning methodology behind the Microarchitecture Exploration analysis and some of the complexities associated with this analysis, see [Understanding How General Exploration Works in Intel® VTune™ Profiler](#).
- For architecture-specific Tuning Guides, see <https://www.intel.com/content/www/us/en/developer/articles/guide/processor-specific-performance-analysis-papers.html>.

Configure and Run Analysis

To configure options for the Microarchitecture Exploration analysis:

Prerequisites: Create a project and specify an analysis target.

1. Click the



(standalone GUI)/

(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From **HOW** pane, click the



Browse button and select **Microarchitecture Exploration**.

3. Configure the following options:

CPU sampling interval, ms spin box

Specify an [interval](#) (in milliseconds) between CPU samples.

Possible values - **1-1000**.

The default value is **1 ms**.

Extend granularity for the top-level metrics selection area

By default, VTune Profiler collects data required to compute top-level metrics ([Front-End Bound](#), [Bad Speculation](#), [Memory Bound](#), [Core Bound](#), and [Retiring](#)) and all their sub-metrics.

You may limit the data collection by selecting particular top-level metrics. In this case, the VTune Profiler extends the level of granularity and collects additional sub-metrics only for the selected top-level metrics. For example, if you select the **Memory Bound** top-level metric, the VTune Profiler collects additional data and provides **Memory Bound** sub-metrics (such as DRAM Bound, Store Bound, and so on), which helps narrow down the analysis to particular microarchitecture levels.

Limiting the amount of data collected simultaneously may also improve profiling accuracy due to less multiplexing. This may be particularly helpful for short-running application or applications with short phases.

Analyze memory bandwidth check box	Collect the data required to compute memory bandwidth. The option is disabled by default.
Evaluate max DRAM bandwidth check box	Evaluate maximum achievable local DRAM bandwidth before the collection starts. This data is used to scale bandwidth metrics on the timeline and calculate thresholds. The option is enabled by default.
Collection mode drop-down menu	Choose the Detailed sampling-based collection mode (default) to view a data breakdown per function and other hotspots. Use the Summary counting-based mode for an overview of the whole profiling run. This mode has a lower collection overhead and faster post-processing time.
Details button	Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable additional settings for the analysis, you need to create a custom configuration by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

NOTE

- For detailed information on events collected for Microarchitecture Exploration on a particular microarchitecture, refer to the [Intel Processor Event Reference](#).
- To [generate the command line](#) for this configuration, use the



Command Line button at the bottom.

4. Click the



Start button to [run the analysis](#).

View Data

To analyze the collected data, use the default [Microarchitecture Exploration viewpoint](#) that provides a high-level performance overview based on the Top-Down Microarchitecture Analysis Method. To easier understand where you could focus your optimization efforts and which part of the microarchitecture pipeline introduces inefficiencies, start with the [Microarchitecture Pipe](#).

See Also

[collect microarchitecture-exploration](#)
vtune option to run the analysis from CLI
[Hardware Event-based Sampling Collection](#)

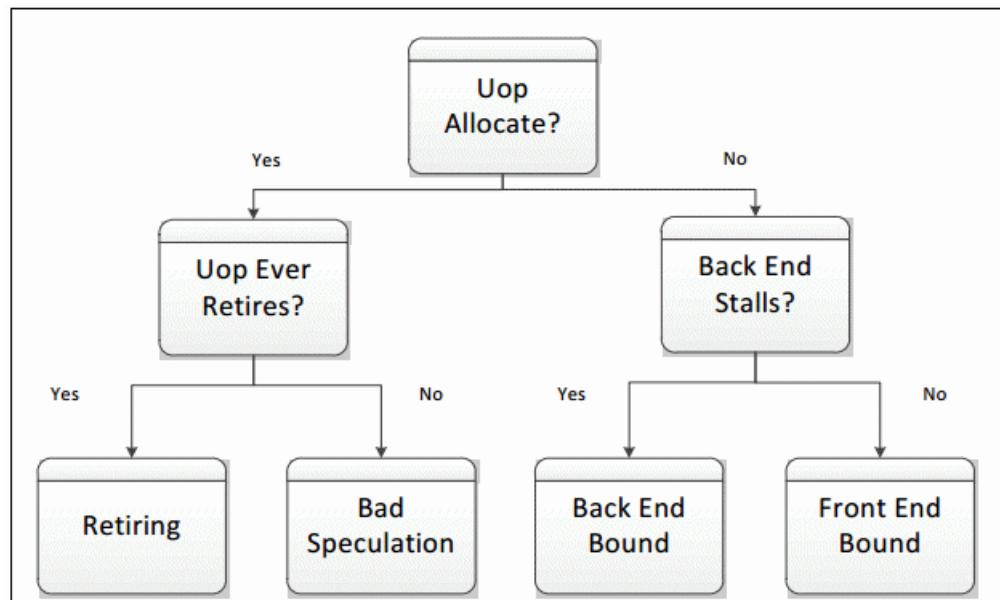
[Set Up Project](#)

Microarchitecture Exploration View

Explore the Intel® VTune™ Profiler Microarchitecture Exploration viewpoint for the PMU analysis based on the top-down microarchitecture analysis method that uses key hardware metrics organized by execution

categories so that you could easily identify what portion of the pipeline is responsible for the majority of execution time.

When the [Microarchitecture Exploration analysis](#) (formerly known as General Exploration) is complete, the VTune Profiler opens the Microarchitecture Exploration viewpoint. The hierarchy of [event-based metrics](#) in this viewpoint depends on your hardware architecture. For example, starting with the Intel microarchitecture code name Ivy Bridge, the VTune Profiler analyzes execution categories based on the [Top-Down Microarchitecture Analysis Method](#):



The four leaf categories serve as high-level performance metrics in the Microarchitecture Exploration viewpoint.

Each metric is an event ratio defined by Intel architects and has its own predefined threshold. VTune Profiler analyzes a ratio value for each aggregated program unit (for example, function). When this value exceeds the threshold and the program unit has more than 5% of CPU time from collection CPU time, it signals a potential performance problem and highlights such a value in pink.

NOTE

- For a detailed tuning methodology behind the Microarchitecture Exploration analysis and some of the complexities associated with this analysis, see [Understanding How General Exploration Works in Intel® VTune™ Profiler](#).
- For architecture-specific Tuning Guides, visit <https://www.intel.com/content/www/us/en/developer/articles/guide/processor-specific-performance-analysis-papers.html>.

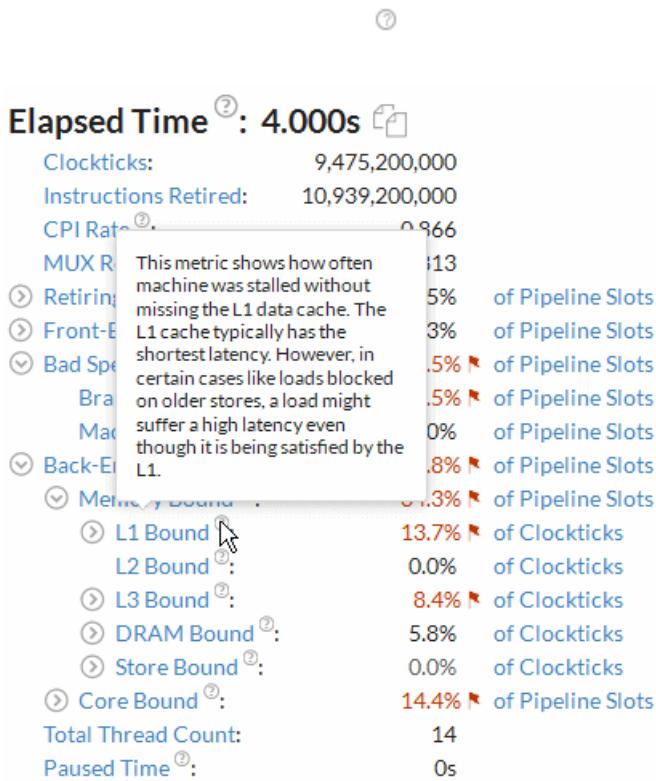
To interpret the performance data provided during the hardware event-based sampling analysis, you may follow the steps below:

1. Learn metrics and define a performance baseline.
2. Identify hardware issues.
3. Analyze source.
4. Explore other analysis types/viewpoints.

Learn Metrics and Define a Performance Baseline

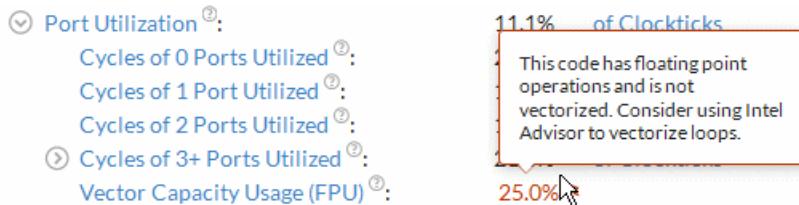
In the Microarchitecture Exploration viewpoint, click the **Summary** tab to switch to the [Summary window](#).

The first section displays the summary statistics on the overall application execution per hardware-related metrics measured in [Pipeline Slots or Clockticks](#). Metrics are organized by execution categories in a list and also represented as a [μPipe diagram](#). To view a metric description, mouse over the help icon



In the example above, mousing over the **L1 Bound** metric displays the metric description in the tooltip.

A flagged metric value signals a performance issue for the whole application execution. Mouse over the flagged value to read the issue description:



You may use the performance issues identified by the VTune Profiler as a baseline for comparison of versions before and after optimization. Your primary performance indicator is the Elapsed time value.

Grayed out metric values indicate that the data collected for this metric is unreliable. This may happen, for example, if the number of samples collected for PMU events is too low. In this case, when you hover over such an unreliable metric value, the VTune Profiler displays a message:



You may either ignore this data, or rerun the collection with the data collection time, sampling interval, or workload increased.

By default, the VTune Profiler collects Microarchitecture Exploration data in the **Detailed** mode. In this mode, all metric names in the Summary view are hyperlinks. Clicking such a hyperlink opens the **Bottom-up** window and sorts the data in the grid by the selected metric. The lightweight **Summary** collection mode is limited to the Summary view statistics.

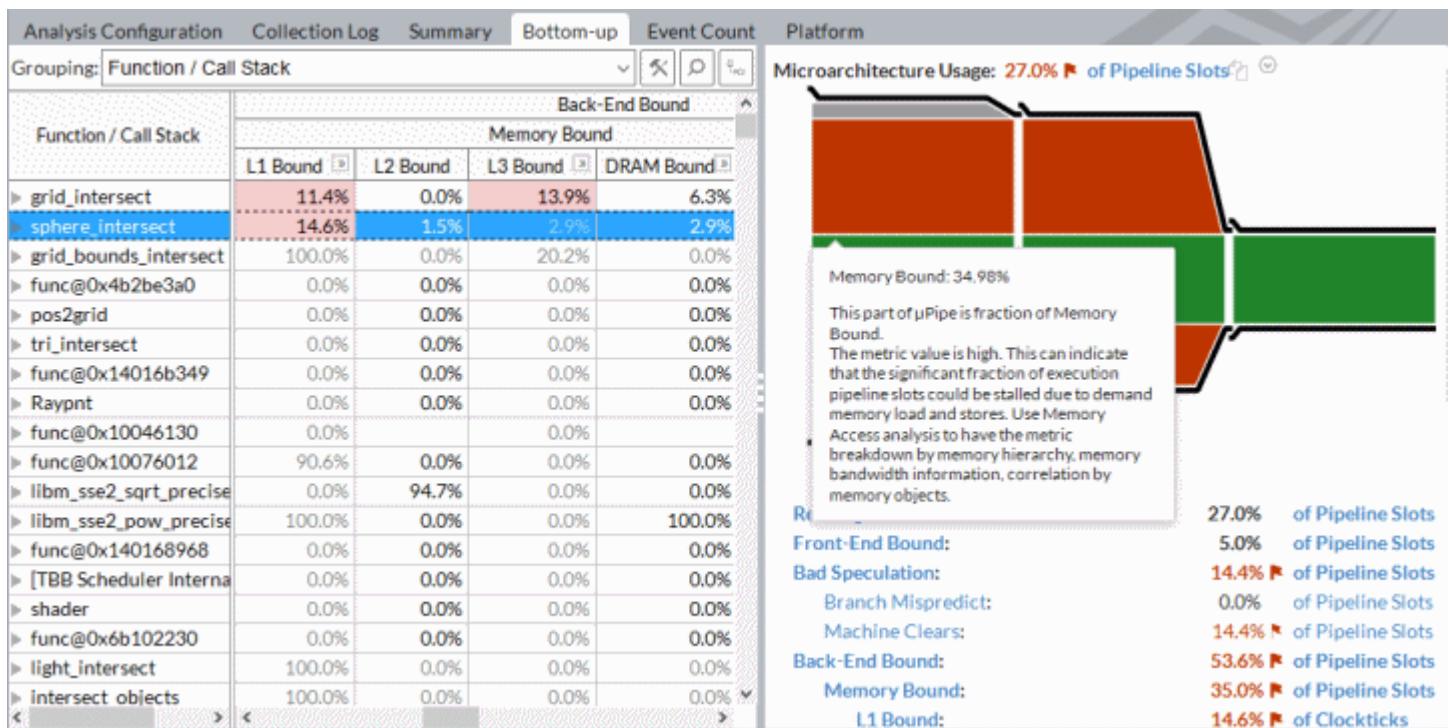
Identify Hardware Issues

To view hardware issues per a program unit, switch to the [Bottom-up pane](#). Each row represents a program unit and percentage of time used by this unit. Program units that take more than 5% of the CPU time are considered as *hotspots*. By default, the VTune Profiler sorts the data in the descending order by Clockticks and provides the hotspots at the top of the list.

Most of the columns in the **Bottom-up** pane represent a hardware performance metric. VTune Profiler calculates a metric based on the formula provided by Intel architects. Mouse over the column header to read the metric description. By default, metric values are represented as numbers. You can change the representation mode with the **Show Data As** context menu option.

The right pane displays a context summary for the selected function. Analyze per-function hardware metrics and their visual representation on the μPipe diagram to estimate the contribution of this particular function to the overall performance.

Each metric has a threshold value. If the metric value exceeds the threshold and the program unit is a hotspot, the VTune Profiler highlights this value in pink as performance-critical. Mouse over each pink cell to read a description of the issue and recommended solution (if any).



In the example above, created on the Intel microarchitecture code name Skylake, the VTune Profiler identified the `sphere_intersect` function as one of the biggest hotspots that took much CPU time. VTune Profiler detected that the back-end portion of the pipeline caused the stalls. For the back-end, the VTune Profiler identified **Memory Bound > L1 Bound** issue as a dominant bottleneck. 14.6% of Clockticks used in this function was stalled missing L1 data cache. This means that if you focus on this function hotspot and optimize it, you can potentially gain ~15% speed-up for this function.

VTune Profiler is able to identify the most common types of pipeline bottlenecks. You may go deeper for more details. If the deeper levels of the metrics do not display any data, it means that the VTune Profiler cannot see a dominant bottleneck on the lower level.

Analyze Source

When you identified a critical function, double-click it to open the **Source/Assembly** window and analyze the source code.



The **Source/Assembly** window displays locator metrics that show what code contributed the most to the issue represented by the metric. For example, if you have the Back-End Bound metric equal to 60% for your function, the source view for this function splits the 60% value across function source lines or instructions to help you identify a source line/instruction with the biggest value contributing the most to the total 60% Back-End Bound metric.

Use the [hotspots navigation](#) toolbar buttons to navigate to the biggest hotspot for each locator metric and identify the code to optimize.

What's Next

- You may view the collected data using the Hotspots viewpoint or run the Hotspots analysis type. Analyzing the source and assembly code for the hotspot function in the Hotspots viewpoint helps identify which instruction contributes most to the poor performance and how much CPU time the hotspot source line takes. Such a code analysis could be useful for the hotspots that do not show any issues in the sub-metrics but do show problems at the upper level of metrics (see the example above).
- Run the [comparison analysis](#) to understand the performance gain you obtained after your optimization.
- You may create your custom analysis configuration and monitor events you are interested in.

NOTE

- For information on processor events, see the [Intel Processor Event Reference](#).
 - Explore [tuning recipes](#) for hardware issues in the *Performance Analysis Cookbook*.
-

See Also

[Analyze Performance](#)

[Custom Analysis](#)

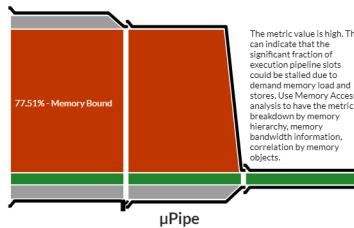
[Cookbook: Top-Down Microarchitecture Analysis Method](#)

[Source Code Analysis](#)

Microarchitecture Pipe

Explore the μPipe diagram of the CPU microarchitecture metrics provided by the Intel® VTune™ Profiler with the Microarchitecture Exploration analysis to identify inefficiencies in the CPU utilization.

When your Microarchitecture Exploration analysis result is collected, the VTune Profiler opens the [Summary window](#) that provides an overview of your target app performance based on the [Top-down Microarchitecture Analysis Method](#) (TMA). Treat the diagram as a pipe with an output flow equal to the ratio: **Actual Instructions Retired/Possible Maximum Instruction Retired** (pipe efficiency). If there are pipeline stalls decreasing retiring, the pipe shape gets narrow.



The μPipe is based on CPU [pipeline slots](#) that represent hardware resources needed to process one micro-operation. Usually there are several pipeline slots available (pipeline width). If pipeline slot does not retire, this is considered as a stall. The fraction of retired pipeline slots represents CPU Microarchitecture efficiency. If there were no stalls on all the CPU cycles, this is considered as 100% efficient CPU execution.

There are usually multiple reasons for stalling pipeline slots, identification of these reasons, as well as their root causes is a CPU Microarchitecture performance analysis process based on the TMA model.

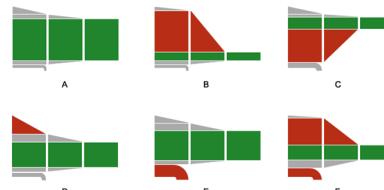
The μPipe in the Microarchitecture Exploration viewpoint visualizes top-level CPU microarchitecture metrics as fractions of the overall number of pipeline slots in a pipe form where all the stalls are represented as obstacles making the pipe narrow.

The pipe is divided into 3 columns and 5 rows where each row represents a pipeline high-level metric:

- [Retiring](#) metric (a fraction of retired pipeline slots) in the middle green row represents the efficiency of the pipe and spans for all 3 columns.
- [Memory Bound](#) metric row above the Retiring metric spans for 2 columns.
- [Core Bound](#) metric row under the Retiring metric spans for 2 columns.
- [Front-End Bound](#) metric is the top row.
- [Bad Speculation](#) metric row at the bottom may have a dedicated representation of a drain meaning wasted CPU work.

The height of the whole pipe is a constant value. The height of every row equals the fraction represented by the corresponding metric.

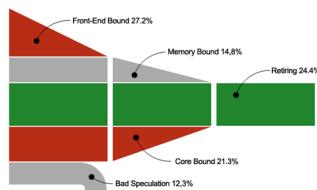
Red color signals a potential performance problem. A fraction of the green color in the diagram helps estimate how good execution efficiency is. So, the pipe form clearly represents existing CPU microarchitecture issues and enables you to recognize the following common patterns:



A	no significant issues
B	Memory bound execution
C	Core bound execution
D	Front End bound execution
E	Bad Speculation issues (for example, branch misprediction)
F	a combination of Memory and Bad Speculation issues

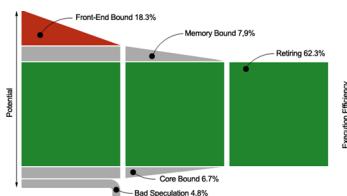
Example 1

This is an example of a pipe representing significant Front-End Bound and Core Bound issues limiting the whole efficiency to 24.4%:



Example 2

This is an example of good CPU execution efficiency with a Front-End issue:



See Also

[Instructions Retired Event](#)

[CPU Metrics Reference](#)

Memory Access Analysis for Cache Misses and High Bandwidth Issues

Use the Intel® VTune™ Profiler's Memory Access analysis to identify memory-related issues, like NUMA problems and bandwidth-limited accesses, and attribute performance events to memory objects (data structures), which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.

NOTE

Intel® VTune™ Profiler is a new renamed version of Intel® VTune™ Amplifier.

How It Works

Grouping:	Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack					
Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
▼DRAM, GB/sec	9.703s	64.3%	6,517,0...	4,141,26...	191,811,508	92
▼High	4.253s	56.8%	2,345,0...	2,111,23...	119,007,140	115
▶ main	4.059s	54.6%	2,170,0...	2,046,83...	119,007,140	108
▶ __intel_ssse3_rep_memcpy	0.177s	100.0%	175,000...	63,000,945	0	223
▶ __do_softirq	0.012s	0.0%	0	0	0	0
▶ run_timer_softirq	0.002s		0	0	0	0
▶ __do_page_fault	0.001s	0.0%	0	0	0	0
▶ numa_migrate_prep	0.001s	0.0%	0	0	0	0
▶ task_cputime	0s	0.0%	0	1,400,021	0	0
▶ Medium	2.880s	70.3%	2,765,0...	981,414,...	52,853,171	83

Memory Access analysis type uses [hardware event-based sampling](#) to collect data for the following metrics:

- **Loads** and **Stores** metrics that show the total number of loads and stores
- **LLC Miss Count** metric that shows the total number of last-level cache misses
 - **Local DRAM Access Count** metric that shows the total number of LLC misses serviced by the local memory
 - **Remote DRAM Access Count** metric that shows the number of accesses to the remote socket memory
 - **Remote Cache Access Count** metric that shows the number of accesses to the remote socket cache
- **Memory Bound** metric that shows a fraction of cycles spent waiting due to demand load or store instructions
 - **L1 Bound** metric that shows how often the machine was stalled without missing the L1 data cache
 - **L2 Bound** metric that shows how often the machine was stalled on L2 cache
 - **L3 Bound** metric that shows how often the CPU was stalled on L3 cache, or contended with a sibling core
 - **L3 Latency** metric that shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited)
 - **NUMA: % of Remote Accesses** metric shows percentage of memory requests to remote DRAM. The lower its value is, the better.
 - **DRAM Bound** metric that shows how often the CPU was stalled on the main memory (DRAM). This metric enables you to identify **DRAM Bandwidth Bound**, **UPI Utilization Bound** issues, as well as **Memory Latency** issues with the following metrics:
 - **Remote / Local DRAM Ratio** metric that is defined by the ratio of remote DRAM loads to local DRAM loads
 - **Local DRAM** metric that shows how often the CPU was stalled on loads from the local memory
 - **Remote DRAM** metric that shows how often the CPU was stalled on loads from the remote memory
 - **Remote Cache** metric that shows how often the CPU was stalled on loads from the remote cache in other sockets
 - **Average Latency** metric that shows an average load latency in cycles

NOTE

- The list of metrics may vary depending on your microarchitecture.
- The UPI Utilization metric replaced QPI Utilization starting with systems based on Intel microarchitecture code name Skylake.

Many of the collected events used in the Memory Access analysis are **precise**. This simplifies understanding the data access pattern. Off-core traffic is divided into the local DRAM and remote DRAM accesses. Typically, you should focus on minimizing remote DRAM accesses that usually have a high cost.

Configure and Run Analysis

To configure options for the Memory Access analysis:

Prerequisites: [Create a project](#).

1. Click the



(standalone GUI)/



(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From **HOW** pane, click the



Browse button and select **Memory Access**.

3. Configure the following options:

CPU sampling interval, ms field	Specify an interval (in milliseconds) between CPU samples. Possible values - 0.01-1000 . The default value is 1 ms .
Analyze dynamic memory objects check box (Linux only)	Enable the instrumentation of dynamic memory allocation/de-allocation and map hardware events to such memory objects. This option may cause additional runtime overhead due to the instrumentation of all system memory allocation/de-allocation API. The option is disabled by default.
Minimal dynamic memory object size to track, in bytes spin box (Linux only)	Specify a minimal size of dynamic memory allocations to analyze. This option helps reduce runtime overhead of the instrumentation. The default value is 1024 .
Evaluate max DRAM bandwidth check box	Evaluate maximum achievable local DRAM bandwidth before the collection starts. This data is used to scale bandwidth metrics on the timeline and calculate thresholds. The option is enabled by default.
Analyze OpenMP regions check box	Instrument and analyze OpenMP regions to detect inefficiencies such as imbalance, lock contention, or overhead on performing scheduling, reduction and atomic operations.

	The option is disabled by default.
Details button	Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable additional settings for the analysis, you need to create a custom configuration by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

4. Click the



Start button to [run the analysis](#).

Limitations:

- Memory objects analysis can be configured for only Linux* targets only and also only for processors based on Intel microarchitectures code named Haswell or newer architectures.

View Data

For analysis, explore the [Memory Usage viewpoint](#) that includes the following windows:

- Summary** window displays statistics on the overall application execution, including the application-level bandwidth utilization histogram.
- Bottom-up** window displays performance data per metric for each hotspot object. If you enable the [Analyze memory objects](#) option for data collection, the **Bottom-up** window also displays memory allocation call stacks in the grid and [Call Stack pane](#). Use the **Memory Object** grouping level, preceded with the **Function** level, to view memory objects as the source location of an allocation call.
- Platform** window provides details on tasks specified in your code with the Task API, Ftrace*/Systrace* event tasks, OpenCL™ API tasks, and so on. If corresponding platform metrics are collected, the Platform window displays over-time data as GPU usage on a software queue, CPU time usage, OpenCL™ kernels data, and GPU performance per the Overview group of GPU hardware metrics, Memory Bandwidth, and CPU Frequency.

Support Limitations

Memory Access analysis is supported on the following platforms:

- 2nd Generation Intel® Core™ processors
- Intel® Xeon® processor families, or later
- 3rd Generation Intel Atom® processor family, or later

If you need to analyze older processors, you can create a [custom analysis](#) and choose events related to memory accesses. However, you will be limited to memory-related events available on those processors. For information about memory access events per processor, see the [VTune Profiler tuning guides](#).

For dynamic memory object analysis on Linux, the VTune Profiler instruments the following Memory Allocation APIs:

- standard system memory allocation API: `mmap`, `malloc/free`, `calloc`, and others
- `memkind` - <https://github.com/memkind/memkind>
- `jemalloc` - <https://github.com/memkind/jemalloc>
- `pmdk` - <https://github.com/pmem/pmdk>

See Also

[Memory Usage View](#)

`collect`

`memory-access``vtune` option

[Intel Processor Events Reference](#)

CPU Metrics Reference

Sampling Interval

Memory Usage View

Use the Intel® VTune™ Profiler to analyze cache misses (L1/L2/LLC), memory loads/stores, memory bandwidth and system memory allocation/de-allocation, identify high bandwidth issues and NUMA issues in your memory-bound application.

To analyze memory usage data, run these analysis types:

- [Memory Access](#) analysis
- [Microarchitecture Exploration](#) analysis with the **Analyze memory bandwidth** option enabled
- [HPC Performance Characterization](#) analysis with the **Analyze memory bandwidth** option enabled

When the analysis is complete, VTune Profiler opens the Memory Usage [viewpoint](#). This viewpoint displays data per memory-access-correlated [event-based metrics](#). Each metric is an event ratio defined by Intel architects and may have its own predefined threshold. VTune Profiler analyzes a ratio value for each aggregated program unit (for example, function). When this value exceeds the threshold and the program unit has more than 5% of CPU time from the collection CPU time, it signals a potential performance problem and highlights that value.

To interpret performance data obtained through the analysis, follow this procedure:

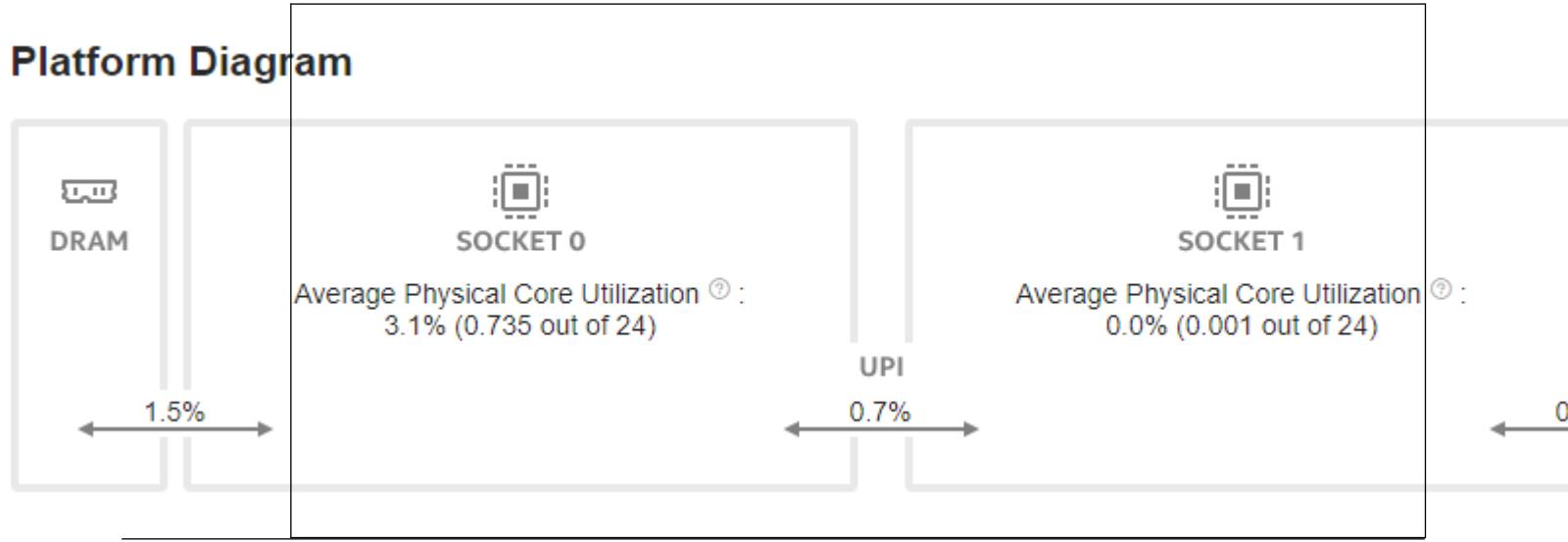
1. [Analyze Topology, Memory, and Cross-Socket Bandwidth](#).
2. [View performance metrics by memory objects \(Linux* targets only\)](#).
3. [Identify code sections and memory objects inducing bandwidth](#).
4. [Analyze bandwidth issues over time](#).
5. [Identify code and memory objects with NUMA issues](#).
6. [Analyze source](#).

Analyze Topology, Memory, and Cross-Socket Bandwidth

Start your performance analysis in the **Summary** window of the Memory Usage viewpoint. Here, the **Platform Diagram** displays system topology and utilization metrics for DRAM, Intel® UPI links, and physical cores.

Sub-optimal application topology can result in induced DRAM and Intel® QuickPath Interconnect (Intel® QPI) or Intel® Ultra Path Interconnect (Intel® UPI) cross-socket traffic. These incidents can limit performance.

Platform Diagram



NOTE

The platform diagram is available for:

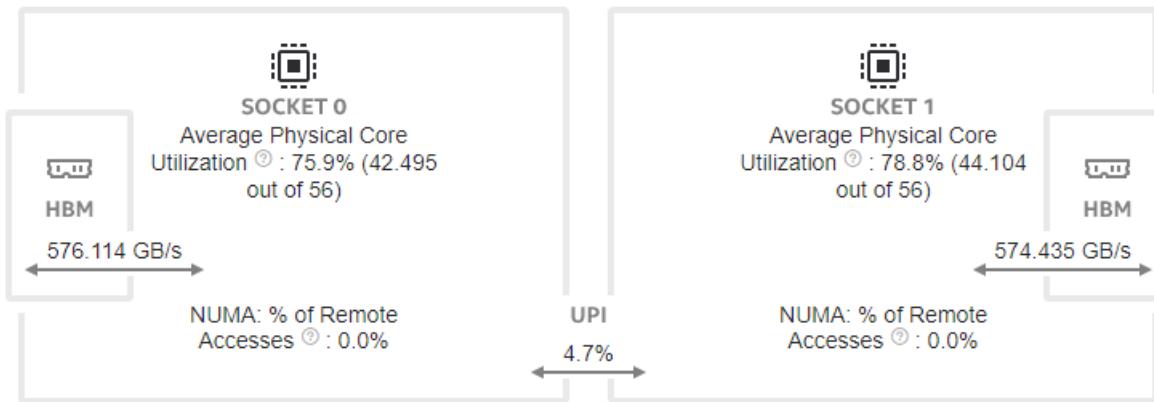
- All client platforms
- Server platforms based on Intel® microarchitecture code name Skylake, with up to four sockets.
- Server platforms based on Intel® microarchitecture code named Sapphire Rapids.

High Bandwidth Memory Data in Platform Diagram

For server platforms based on Intel® microarchitecture code named Sapphire Rapids, the **Platform Diagram** also includes information about High Bandwidth Memory (HBM). Use this information to distinguish from DRAM-specific utilization in the diagram.

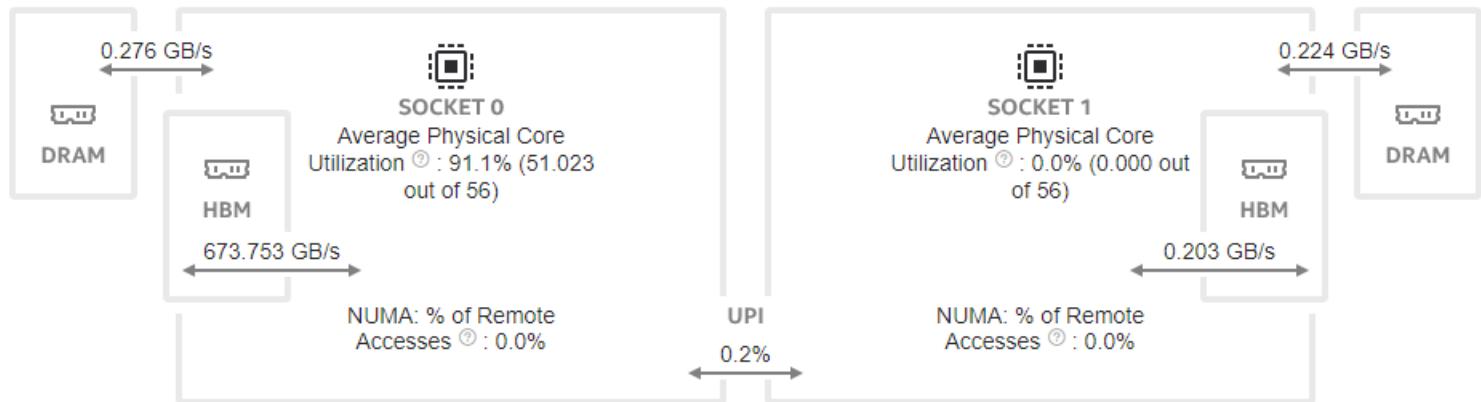
For example, this diagram shows information about HBM mode utilization, where the system has no DRAM.

Platform Diagram



Here is an example of the **Platform Diagram** data in a system that has both HBM and DRAM.

Platform Diagram



If you selected the **Evaluate max DRAM bandwidth** option in your analysis configuration, the Platform Diagram shows the average DRAM utilization. Otherwise, it shows the average DRAM bandwidth.

The **Average UPI Utilization** metric displays UPI utilization in terms of transmit. Irrespective of the number of UPI links that connect a pair of packages, the Platform Diagram shows a single cross-socket connection. If there are several links, the diagram displays the maximum value.

On top of each socket, the **Average Physical Core Utilization** metric indicates the utilization of physical cores by computations of the application under analysis.

Once you examine the topology and utilization information in the diagram, focus on other sections in the Summary window and then switch to the **Bottom-up** and **Platform** windows next.

View Performance Metrics by Memory Objects (Linux* targets only)

If you enabled the **Analyze dynamic memory objects** configuration option for the Memory Access analysis, you can configure the Memory Usage viewpoint to display performance metrics per memory objects (variables, data structures, arrays).

NOTE

Memory objects identification is supported only for Linux targets and only for processors based on Intel microarchitecture code named Haswell and newer architectures. On Windows*, you can [group](#) by Cachelines, see the metrics against the code, and figure out what data structures it accesses.

There are several types of memory objects:

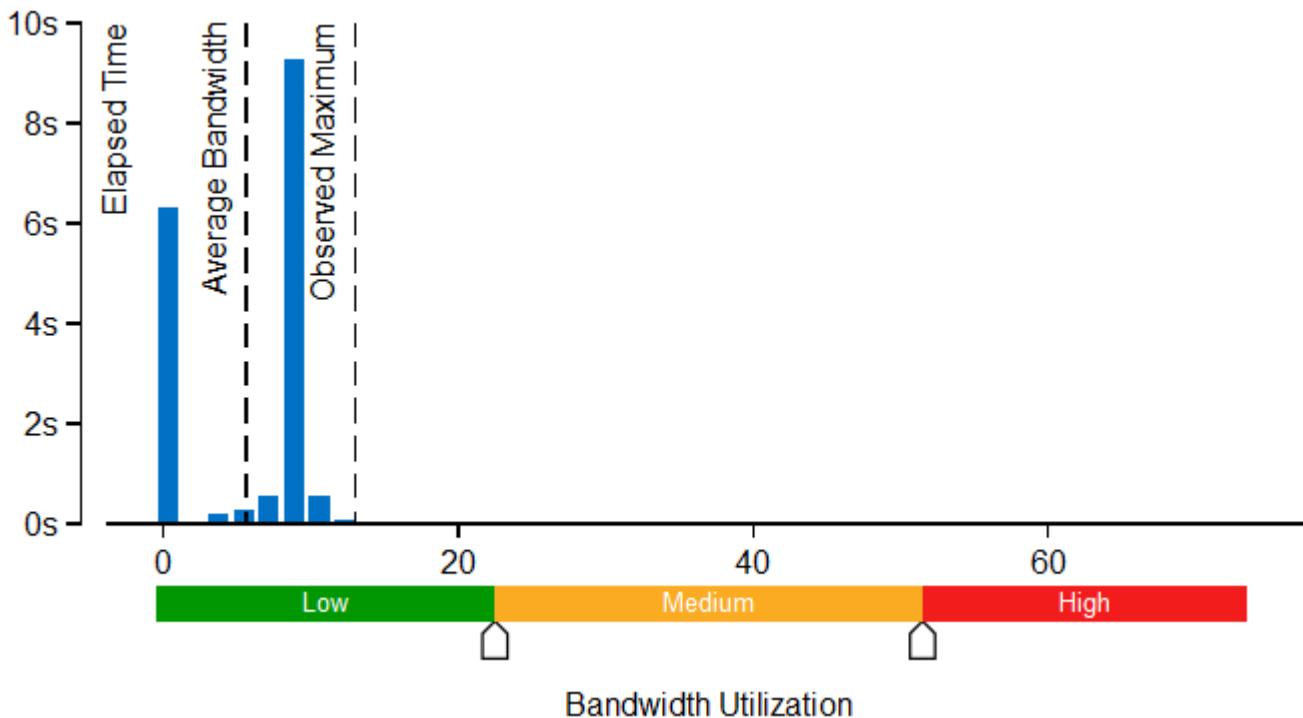
- **Dynamic** memory objects are allocated on heap using the `malloc`, `new`, and similar functions. Such objects are identified by the line where an allocation happened; for example, a source line where the `malloc` function was called.
- **Global** objects are global or static variables. Such objects are identified by the module and variable name, for example: `libiomp5.sp!_kmp_avail_proc (4B)`, where 4B is an allocation size.
- **Stack** objects are local variables. VTune Profiler does not recognize individual variables, so all references to stack memory are associated with one memory object named `[Stack]`.

For memory objects data, click the **Bottom-up** tab and select a grouping level containing **Memory Object** or **Memory Object Allocation Source**. The **Memory Object** granularity groups the data by individual allocations (call site and size) while **Memory Object Allocation Source** groups by the place where an allocation happened.

Only metrics based on DLA-capable hardware events are applicable to the memory objects analysis. For example, the CPU Time metric is based on a non DLA-capable Clockticks event, so cannot be applied to memory objects. Examples of applicable metrics are Loads, Stores, LLC Miss Count, and Average Latency.

Identify Code Sections and Memory Objects Inducing Bandwidth

In the **Bandwidth Utilization** section of the **Summary** window, you can select a bandwidth domain (like DRAM or Interconnect) and analyze the bandwidth utilization over time represented on the histogram:



This histogram shows how much time the system bandwidth was utilized by the selected bandwidth domain and provides thresholds to categorize bandwidth utilization as High, Medium and Low. By default, for Memory Analysis results the thresholds are calculated based on the maximum achievable DRAM bandwidth measured by the VTune Profiler before the collection starts and displayed in the **System Bandwidth** section of the **Summary** window. To enable this functionality for custom analysis results, make sure to select the **Evaluate max DRAM bandwidth** option. If this option is not enabled, the thresholds are calculated based on the maximum bandwidth value collected for this result. You can also set the threshold by moving sliders at the bottom. The modified values will be applied to all subsequent results in this project.

Explore the table under the histogram to identify which functions were frequently accessed while the bandwidth utilization for the selected domain was high. Clicking a function from the list opens the [Bottom-up window](#) with the grid automatically grouped by **Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack** and this function highlighted. Under the **DRAM, GB/sec > High** utilization type, you can see all functions executing when the system DRAM bandwidth utilization was high. Sort the grid by **LLC Miss Count** to see what functions contributed to the high DRAM bandwidth utilization the most:

Grouping: Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack

Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
▼DRAM, GB/sec	9.703s	64.3%	6,517,0...	4,141,26...	191,811,508	92
▼High	4.253s	56.8%	2,345,0...	2,111,23...	119,007,140	115
▶ main	4.059s	54.6%	2,170,0...	2,046,83...	119,007,140	108
▶ __intel_ssse3_rep_memcpy	0.177s	100.0%	175,000...	63,000,945	0	223
▶ __do_softirq	0.012s	0.0%	0	0	0	0
▶ run_timer_softirq	0.002s		0	0	0	0
▶ __do_page_fault	0.001s	0.0%	0	0	0	0
▶ numa_migrate_prep	0.001s	0.0%	0	0	0	0
▶ task_cputime	0s	0.0%	0	1,400,021	0	0
▶ Medium	2.880s	70.3%	2,765,0...	981,414,...	52,853,171	83

In addition to identifying bandwidth-limited code, the VTune Profiler provides a workflow to see the frequently accessed memory objects (variables, data structures, arrays) that had an impact on the high bandwidth utilization. So, if you enabled the memory object analysis for your target, the **Bandwidth Utilization** section includes a table with the top memory objects that were frequently accessed while the bandwidth utilization for the selected domain was high. Click such an object to switch to the **Bottom-up** window with the grid automatically grouped by **Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack** and this object highlighted. Under the **DRAM > High** utilization type, explore all memory objects that were accessed when the system DRAM bandwidth utilization was high. Sort the grid by **LLC Miss Count** to see what memory objects contributed to the high DRAM bandwidth utilization the most:

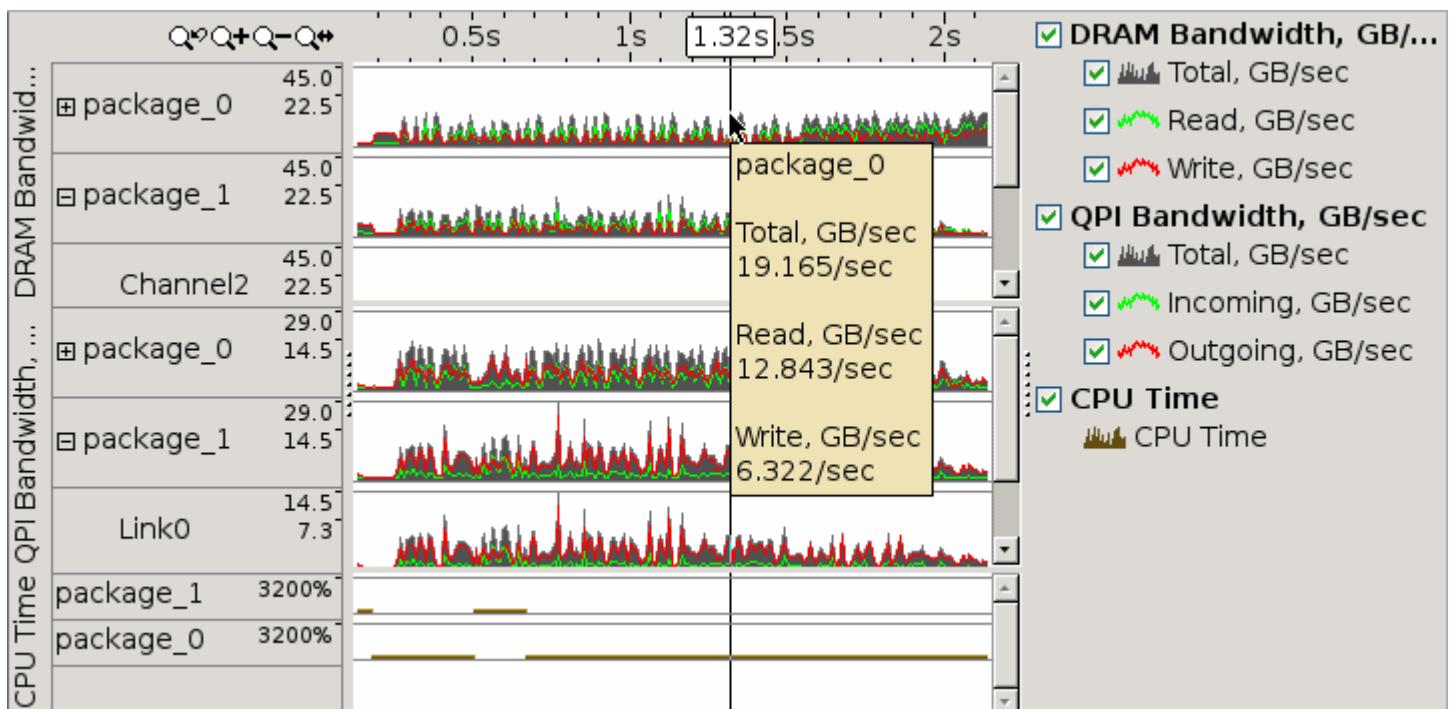
Grouping: Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack

Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
▼DRAM, GB/sec	9.703s	64.3%	6,517,0...	4,141,26...	191,811,508	92
▼High	4.253s	56.8%	2,345,0...	2,111,23...	119,007,140	115
▶ lin_stream.cpp:100 (152 MB)			910,002...	887,613,...	51,453,087	119
▶ lin_stream.cpp:99 (152 MB)			826,002...	770,011,...	39,902,394	91
▶ lin_stream.cpp:98 (152 MB)			609,001...	452,206,...	27,651,659	142
▶ [Unknown]			0	1,400,021	0	0
▶ Medium	2.880s	70.3%	2,765,0...	981,414,...	52,853,171	83
▶ Low	2.571s	71.6%	1,407,0...	1,048,61...	19,951,197	57

Analyze Bandwidth Issues Over Time

To identify bandwidth issues in your application over time, focus on the [Timeline pane](#) provided at the top of the **Bottom-up** window. For Memory Analysis results, the DRAM Bandwidth graph is scaled according to the maximum achievable DRAM bandwidth measured by the VTune Profiler before the collection start. To enable this functionality for custom analysis results, make sure to select the **Evaluate max DRAM bandwidth** option. If this option is not enabled, the thresholds are calculated based on the maximum bandwidth value collected for this result.

Bandwidth events are not associated with any core, but, instead, associated with the *uncore* (iMC, the integrated memory controller). Uncore events happen on structures shared between all CPUs in a package (for example, 10 CPUs on a single package). This makes it impossible to associate any single uncore event with any code context. So, the VTune Profiler may only associate bandwidth uncore event counts with the socket, or package, on which the uncore event happened, and time.



Hover over a bar with high bandwidth value to learn how much data was read from or written to DRAM through the on-chip memory controller. Use time-filtering context menu options to filter in a specific range of time during which bandwidth is notable. Then, switch to the core-based events that correlate with bandwidth in the grid below to determine what specific code is inducing all the bandwidth.

Function / Call Stack	CPU Time ▾	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
▶ main	7.224s	56.3%	5,019,0...	3,264,84...	190,061,403	100
▶ __intel_ssse3_rep_memcpy	2.181s	88.0%	1,435,0...	842,812,...	1,050,063	67
▶ clear_page_c_e	0.106s	95.2%	0	0	0	0
▶ copy_page_rep	0.055s	90.8%	21,000,...	8,400,126	700,042	0
▶ checkSTREAMresults	0.033s		0	0	0	0
▶ __do_softirq	0.026s	0.0%	0	0	0	0

Identify Code and Memory Objects with NUMA Issues

Many modern multi-socket systems are based on the Non-Uniform Memory Architecture (NUMA) where accesses to the memory allocated on the home (local) CPU socket have better latency/bandwidth than accesses to the remote memory. To identify NUMA issues, focus on the following hierarchically organized metrics in the **Bottom-up** view:

- **Memory Bound > DRAM Bound > Local DRAM** metric shows a fraction of cycles the CPU stalled waiting for memory loads from the local memory.
- **Memory Bound > DRAM Bound > Remote DRAM** metric shows a fraction of cycles the CPU stalled waiting for memory loads from the remote memory.
- **Memory Bound > DRAM Bound > Remote Cache** metric shows a fraction of cycles the CPU stalled waiting for memory loads from the remote socket cache.
- **LLC Miss Count > Local DRAM Access Count, LLC Miss Count > Remote DRAM Access Count, LLC Miss Count > Remote Cache Access Count** - metrics show the number of accesses to local memory, remote memory and remote cache respectively.

The performance of your application can be also limited by the bandwidth of Interconnect links (inter-socket connections). VTune Profiler provides mechanisms to identify code and memory objects inducing this type of bandwidth similar to those used to identify DRAM bandwidth problems. In the Summary window, use the **Bandwidth Utilization Histogram** and select **Interconnect** in the **Bandwidth Domain** drop-down menu.

If you select the Interconnect Incoming/Outgoing Non-Data categories in the **Bandwidth Domain** drop-down menu, the histogram displays the bandwidth utilized by hardware generated and system traffic like protocol packet headers, snoop requests and responses, and others:

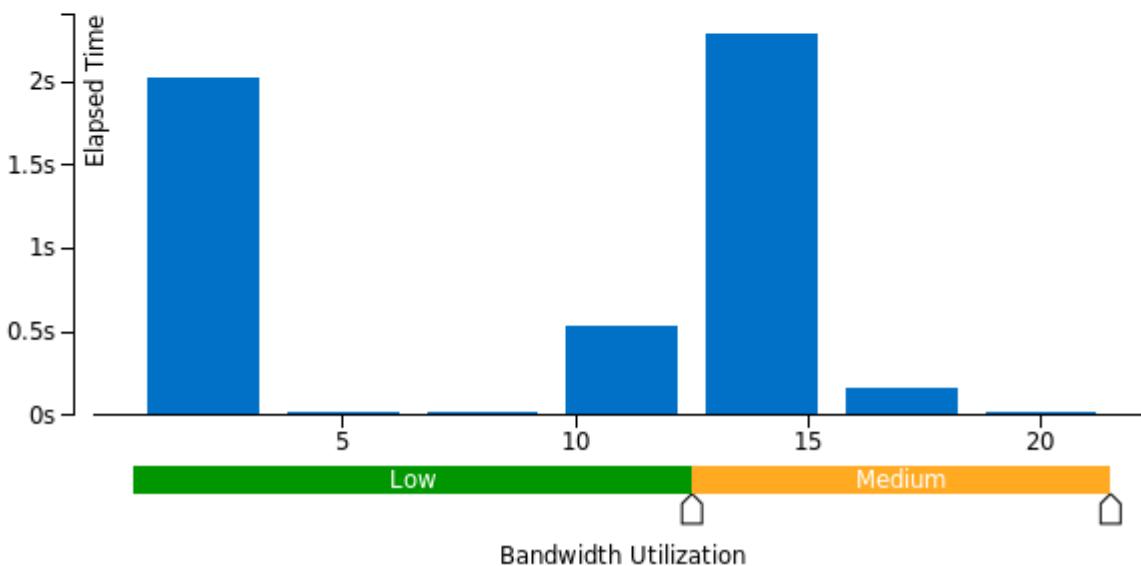
Bandwidth Utilization

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: **[QPI Outgoing Non-Data, GB/sec]** ▾

Bandwidth Utilization Histogram

This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.



NOTE

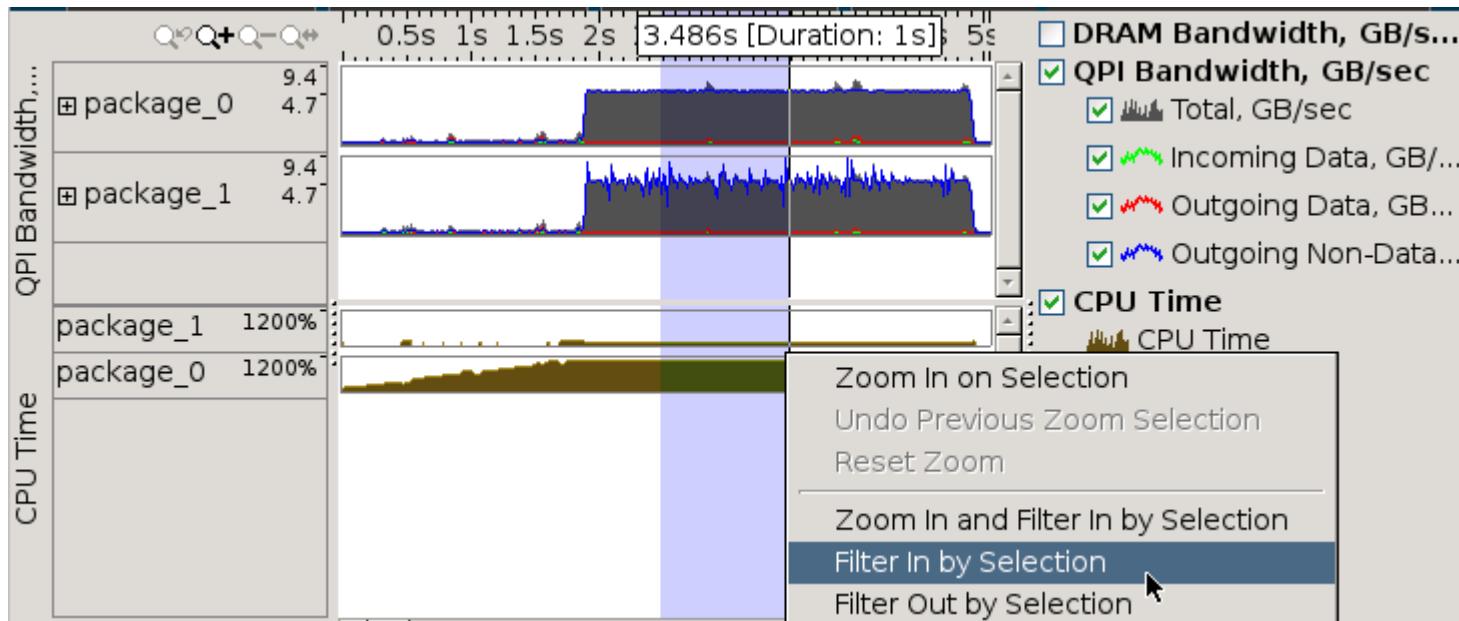
Interconnect bandwidth analysis is supported by the VTune Profiler for Intel microarchitecture code name Ivy Bridge EP and later.

Switch to the **Bottom-up** tab and select the **Bandwidth Domain / Bandwidth Utilization type / Function / Call Stack** grouping level. Expand the **Interconnect** domain grid row and then expand the **High** utilization type row to see all functions that were executing when the system Interconnect bandwidth utilization was high:

Grouping: Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack

Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack	CPU Time ▾	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
▶ DRAM Write, GB/sec	9.703s	64.3%	6,517,0...	4,141,262...	191,811,508	92
▼ QPI, GB/sec	9.703s	64.3%	6,517,0...	4,141,262...	191,811,508	92
▶ Low	6.874s	65.5%	4,403,0...	2,949,844...	133,007,980	95
▶ Medium	2.111s	70.9%	1,673,0...	940,814,1...	46,552,793	72
▼ High	0.719s	32.3%	441,001...	250,603,7...	12,250,735	135
▶ main	0.594s	23.1%	385,001...	215,603,2...	12,250,735	135
▶ __intel_ssse3_rep_memcp	0.119s	76.7%	56,000,...	35,000,525	0	0
▶ __do_softirq	0.003s		0	0	0	0

You can also select areas with the high Interconnect bandwidth utilization in the Timeline view and filter in by this selection:



After the filter is applied, the grid view below the Timeline pane shows what was executing during that time range.

Analyze Source

When you identified a critical function, double-click it to open the **Source/Assembly** window and analyze the source code. The **Source/Assembly** window displays hardware metrics per code line for the selected function.

To view the **Source/Assembly** data for memory objects:

1. Select the **../Function / Memory Object /..** grouping level (the **Function** granularity should precede the **Memory Object** granularity) in the **Bottom-up** window.
2. Expand a function and double-click a memory object under this function.

The **Source/Assembly** window opens displaying metrics per function source lines where accesses to the selected memory object happened.

NOTE

- For information on processor event, see [Intel Processor Event Reference](#).
 - For information on the performance tuning for HPC-computers using the event-based sampling collection, see [Tuning Guides and Performance Analysis Papers](#).
 - For information on performance improvement opportunities with NUMA hardware, see [Optimizing Applications for NUMA](#).
-

See Also

[Source Code Analysis](#)

[VTune Profiler Cookbook: False Sharing](#)

[VTune Profiler Cookbook: Frequent DRAM Accesses](#)

Parallelism Analysis Group

*The **Parallelism** analysis group introduces analysis types based on applications that are compute-sensitive. They can be used as a starting point for overall application performance analysis before moving on to more targeted analysis types.*

Compute-intensive application analysis includes the following analysis types:

- [Threading](#) focuses on a particular target, shows how well your application is threaded for the existing number of logical CPU cores, identifies functions that took the most CPU time to execute and the synchronization objects that might cause ineffective CPU usage.
- [HPC Performance Characterization](#) evaluates compute-sensitive or throughput applications for floating point operation and memory efficiency. It can be used as a starting point for understanding overall application performance.

Threading Analysis

Use the Threading analysis to identify how efficiently an application uses available processor compute cores and explore inefficiencies in threading runtime usage or contention on threading synchronization that makes threads waiting and prevents effective processor utilization.

NOTE

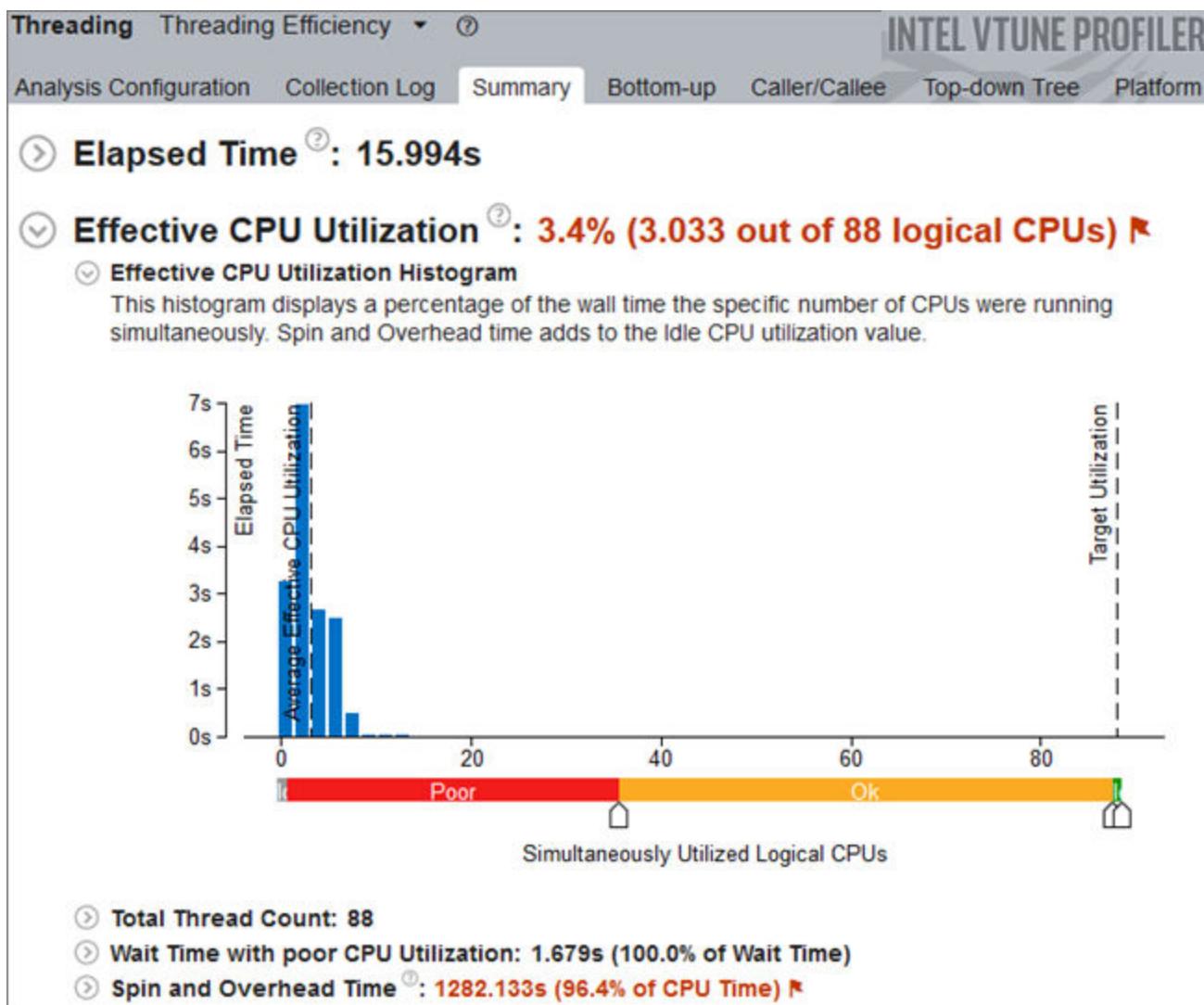
- Threading analysis combines and replaces the Concurrency and Locks and Waits analysis types available in previous versions of Intel® VTune™ Profiler.
 - Intel® VTune™ Profiler is a new renamed version of Intel® VTune™ Amplifier.
-

Intel® VTune™ Profiler uses the **Effective CPU Utilization** metric as a main measurement of threading efficiency. The metric is built on how an application utilizes the available logical cores. For throughput computing, it is typical to load one logical core per physical core.

The following aspects of Threading Analysis provide possible reasons for poor CPU utilization:

- [Thread count](#): a quick glance at the application thread count can give clues to threading inefficiencies, such as a fixed number of threads that might prevent the application from scaling to a larger number of cores or lead to thread oversubscription

- **Wait time** (trace-based or context switch-based): analyze threads waiting on synchronization objects or I/O
- **Spin and overhead time**: estimate threading runtime overhead or the impact of spin waits (busy or active waits)



The Threading Analysis provides two collection modes with major differences in thread wait time collection and interpretation:

- **User-Mode Sampling and Tracing**, which can recognize synchronization objects and collect thread wait time by objects using tracing. This is helpful in understanding thread interaction semantics and making optimization changes based on that data. There are two groups of synchronization objects supported by Intel VTune Profiler: objects usually used for synchronization between threads (such as Mutex or Semaphore) and objects associated with waits on I/O operations (such as Stream).
- **Hardware Event-Based Sampling and Context Switches**, which collects thread inactive wait time based on context switch information. Even though there is not a thread object definition in this case, the problematic synchronization functions can be found by using the wait time attributed with call stacks with lower overhead than the previous collection mode. The analysis based on context switches also shows thread preemption time, which is useful in measuring the impact of thread oversubscription on a system.

How It Works: User-Mode Sampling and Tracing

With [user-mode sampling and tracing collection](#), VTune Profiler instruments threading and blocking API intercepting the calls during runtime and building thread interaction flow detecting synchronization objects. Using User-mode Sampling and Tracing Collection analysis mode you can estimate the impact each synchronization object has on the application and understand how long the application had to wait on each synchronization object, or in blocking APIs. The analysis shows the thread interaction with execution flow transition from one thread to another with releasing and accruing synchronization objects on the timeline view.

If this mode brings significant overhead in the application runtime, try the [Hardware Event-Based Sampling and Context Switches](#) mode, which offers a less intrusive method of wait time collection.

Wait Time with poor CPU Utilization: 1441.885s (100.0% of Wait Time)			
🕒 Top Waiting Objects			
This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on contended synchronizations. A significant amount of Wait time associated with a synchronization object reflects high contention for that object and, thus, reduced parallelism.			
Sync Object	Wait Time with poor CPU Utilization	(% from Object Wait Time)	Wait Count ⓘ
Mutex 0x0ddd4096	1166.157s	100.0%	6,740,163
Thread 0x63f112fa	194.350s	100.0%	4
Condition Variable 0x47dc96d0	41.970s	100.0%	172,710
Condition Variable 0x8bf64680	39.408s	100.0%	245,089
Thread Pool	0.001s	100.0%	8
[Others]	0.000s	100.0%	4

*N/A is applied to non-summable metrics.

How It Works: Hardware Event-Based Sampling and Context Switches

Multitask operating systems execute all software threads in time slices (thread execution quanta). In the [Hardware Event-Based Sampling and Context Switches](#) mode, the profiler gains control whenever a thread gets scheduled on and then off a processor (that is, at thread quantum borders). This mode also determines a reason for thread inactivation, which includes an explicit request for synchronization or thread quantum expiration (when the operating system scheduler preempts the current thread to run a higher-priority thread instead).

The time during which a thread remains inactive is measured and called Inactive Wait Time. Inactive Wait Time is differentiated based on the reason for inactivity:

- Inactive Sync Wait Time is caused by a request for synchronization
- Preemption Wait Time is caused by preemption

Inactive Wait Time with poor CPU Utilization: 47.436s (100.0% from Inactive Wait Time)						
Inactive Sync Wait Time ⓘ: 47.435s *						
Preemption Wait Time ⓘ: 0.001s						
🕒 Top Functions by Inactive Wait Time with Poor CPU Utilization.						
This section lists the functions sorted by the time spent waiting on synchronization or thread preemption with poor CPU Utilization.						
Function	Module	Inactive Wait Time	Inactive Sync Wait Time ⓘ	Inactive Sync Wait Count	Preemption Wait Time ⓘ	Preemption Wait Count
pthread_cond_wait	libpthread-2.23.so	37.068s	37.068s *	2,119,873	0.000s	11
_GL_pthread_mutex_lock	libpthread-2.23.so	7.838s	7.838s *	401,209	0.000s	1
pthread_cond_broadcast	libpthread-2.23.so	2.165s	2.165s	134,802	0s	0
[vtsspp]	vtsspp	0.247s	0.247s	13,428	0.000s	17
pthread_join	libpthread-2.23.so	0.116s	0.116s	2	0s	0
[Others]		0.001s	0.000s	15	0.001s	16

*N/A is applied to non-summable metrics.

Since context switch information is collected with call stacks, it is possible to explore reasons of Inactive Wait Time by wait functions with their call paths. The [Hardware Event-Based Sampling and Context Switches](#) mode shows the places in the code where the wait was induced by a synchronization object or I/O operation.

The Hardware Event-Based Sampling and Context Switches mode is based on the [hardware event-based sampling collection](#) and analyzes all the processes running on your system at the moment, providing context switching data on whole system performance. On Linux* systems, Inactive Wait Time Collection is available in driverless Perf*-based collection usage with kernel version 4.4 or later. Inactive Time reasons are available in kernel 4.17 and later.

NOTE

On 32-bit Linux* systems, the VTune Profiler uses a [driverless Perf*-based collection](#) for the hardware event-based sampling mode.

Configure and Run Analysis

To configure options for the Threading analysis:

Prerequisites: Create a [project](#) and specify an analysis target.

1. Click the



(standalone GUI)/



(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From **HOW** pane, click the



Browse button and select **Threading**.

3. Configure the collection options.

User-Mode Sampling and Tracing mode	Select to enable the user-mode sampling and tracing collection for synchronization object analysis. This collection mode uses a fixed sampling interval of 10ms. If you need to change the interval, click the Copy button and create a custom analysis configuration. For intervals less than 10ms, use the Hardware Event-Based Sampling and Context Switches mode.
Hardware Event-Based Sampling and Context Switches mode	Select to enable hardware event-based sampling and context switches collection. You can configure the CPU sampling interval, ms to specify an interval (in milliseconds) between CPU samples. Possible values for the hardware event-based sampling mode are 0.01-1000 . 1 ms is used by default.
Details button	Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable additional settings for the analysis, you need to create a custom configuration by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

NOTE

When changing collection options, pay attention to the **Overhead** diagram on the right. It dynamically changes to reflect the collection overhead incurred by the selected options.

Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable [additional settings](#) for the analysis, you need to [create a custom configuration](#) by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

NOTE

To run Threading Analysis from the command line for this configuration, use the



Command Line button at the bottom.

4. Click the



Start button to [run the analysis](#).

[View Data](#)

The Threading analysis results appear in the Threading Efficiency viewpoint, which consists of the following windows/panes:

- Summary window displays statistics on the overall application execution, identifying CPU time and processor utilization.
- [Bottom-up window](#) displays hotspot functions in the bottom-up tree, CPU time and CPU utilization per function.
- [Top-down Tree window](#) displays hotspot functions in the call tree, performance metrics for a function only (Self value) and for a function and its children together (Total value).
- [Caller/Callee window](#) displays parent and child functions of the selected focus function.
- [Platform window](#) provides details on CPU and GPU utilization, frame rate, memory bandwidth, and user tasks (if corresponding metrics are collected).

[What's Next](#)

1. Start on the result **Summary** window to explore the Effective CPU utilization of your application and identify reasons for underutilization connected with synchronization, parallel work arrangement overhead, or incorrect thread count. Click links associated with flagged issues to be taken to more detailed information. For example, clicking a sync object name in the **Top Waiting Objects** table takes you to that object in the **Bottom-up** window.
2. Analyze thread integration synchronization objects with wait and signal stacks and transitions on the timeline. Explore CPU time spent in threading runtimes to classify inefficiencies in their use.
3. Modify your code to remove CPU utilization bottlenecks and improve the parallelism of your application. Concentrate your tuning on objects with long Wait time where the system is poorly utilized (red bars) during the wait. Consider adding parallelism, rebalancing, or reducing contention. Ideal utilization (green bars) occurs when the number of running threads equals the number of available logical cores.
4. Re-run the analysis to verify your optimization with the [comparison mode](#) and identify more possible areas for improvement.

For more information and interpretation tips, see [Threading Efficiency View](#).

[See Also](#)

[Threading Efficiency View](#)

[collect](#)

threading vtune option

[HPC Performance Characterization Analysis](#)

Threading Efficiency View

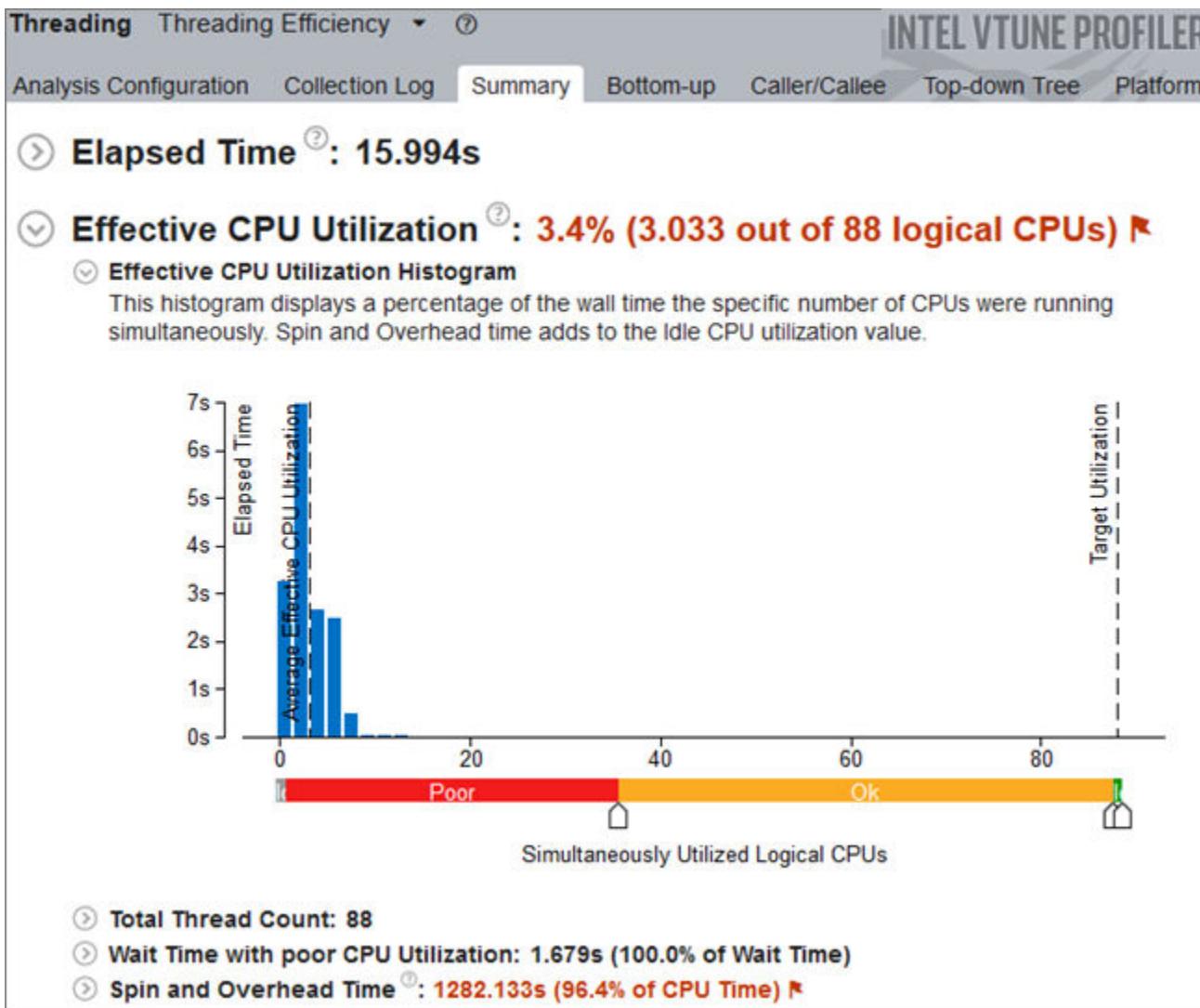
Use the Intel® VTune™ Profiler's Threading Efficiency viewpoint to identify causes of poor CPU utilization such as inefficient synchronization.

Use the following workflow to analyze results collected by the Threading analysis type:

1. Define a performance baseline
2. Examine wait time, spin and overhead time, and thread count metrics
3. Review the timeline
4. Analyze the application source code
5. Explore other analysis types for further diagnosis and optimization

Define a Performance Baseline

Start with analyzing the application-level data provided in the **Summary** window for this analysis result. Use the Elapsed time value as a baseline for comparison of results before and after optimization.



Explore the **Spin Time**, **Overhead Time**, **Wait Time**, and **Total Thread Count** to identify the main cause of performance issues.

Wait Time

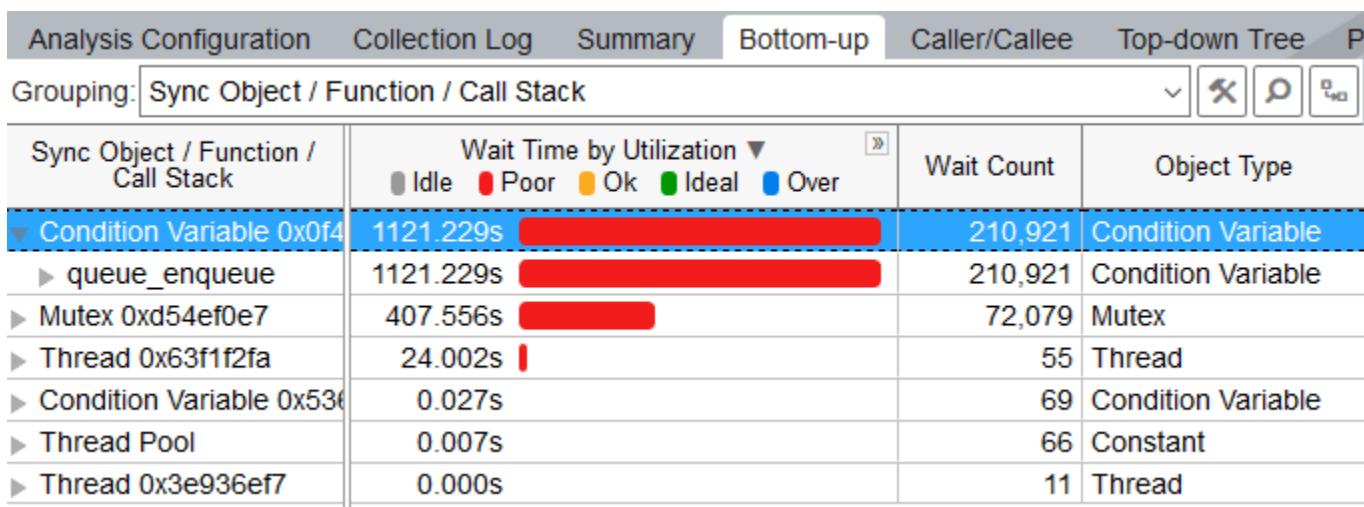
A high thread wait time can cause poor CPU utilization. One common problem in parallel applications is threads waiting too long on synchronization objects that are on the critical path of application execution (for example, locks). Parallel performance suffers when waits occur while cores are under-utilized. Threading analysis helps to analyze thread wait time and find synchronization bottlenecks.

Explore the [Bottom-up window](#) to identify the most performance critical synchronization objects. Although it varies, often there are non-interesting threads waiting for a long time on objects infrequently. Usually you are recommended to focus your tuning efforts on the waits with both high Wait Time and Wait Count values, especially if they have poor utilization/concurrency.

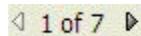
By default, the synchronization objects are sorted by Wait time. You can view the time distribution per utilization level by clicking the



button at the **Wait Time by Utilization** column header to expand the column.



To identify the highest contributing stack for the synchronization objects selected in the **Bottom-up** or **Top-down Tree** panes, use the navigation buttons



on the stack pane. The contribution bar shows the contribution of the currently visible stack to the overall time spent by the selected synchronization objects. You can also use the drop-down list in the **Call Stack** pane to view data for different types of stacks.

You should try to eliminate or minimize the Wait Time for the synchronization objects with the highest Wait Time (or longest red bars, if the bar format is selected) and Wait Count values.

In Hardware Event-based Sampling and Context Switches mode, sort functions by **Inactive Sync Wait Time**. Use the **Caller/Callee** pane to figure out the call sites in the application that calls a wait function with high **Inactive Sync Wait Time**.

Spin and Overhead Time

Threading analysis shows how much time the application spends in threading runtimes either because of busy waits or overhead on parallel work arrangement. The goal is to minimize CPU cycles that are spent either on active wait or task scheduling. Look at the call paths for functions with higher spin and overhead time of application execution and follow the advice of flagged issues to reduce the time.

NOTE

The spin time shown in Spin and Overhead Time section might be included into wait time based on user-level sampling and tracing.

Thread Count

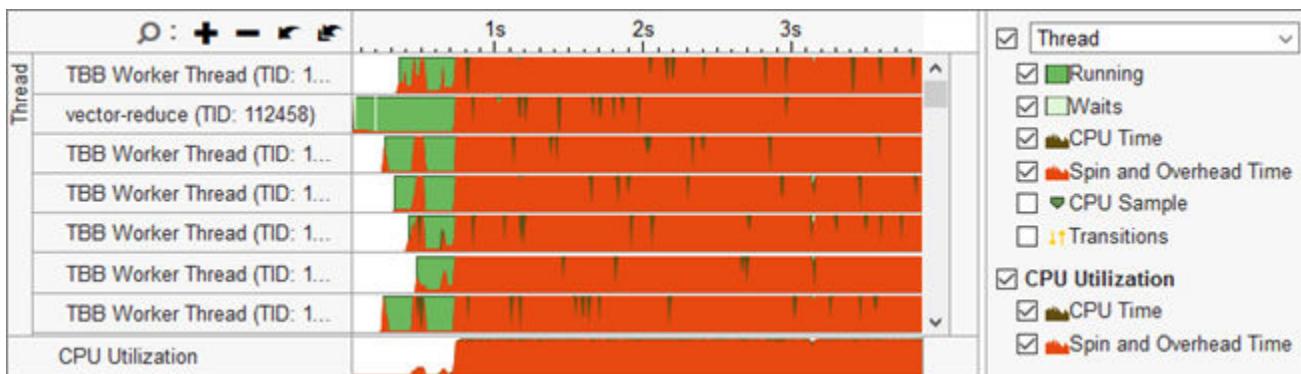
Threading analysis will show time an application spends in oversubscription by flagging when the application is running more threads than the number of logical cores on the machine. Running an excessive number of threads can cause a higher CPU time because some of the threads may be waiting on others to complete or time may be wasted on context switches. Another common issue is running with a fixed number of threads, which can cause performance degradation when running on a platform with a different number of cores. For example, running with a significantly lower number of threads than the number of cores available can cause higher application elapsed time.

Use the **Total Thread Count** metric available on the **Summary** window to determine if your application has thread oversubscription or could benefit from increased threading.

In Hardware Event-based Sampling and Context Switches mode, use the **Preemption Wait Time** metric to estimate the impact of oversubscription. The higher the metric value on worked threads, the higher the impact of oversubscription on the application performance. Note that thread preemption can also be triggered by a conflict with other applications or kernel threads running on a system.

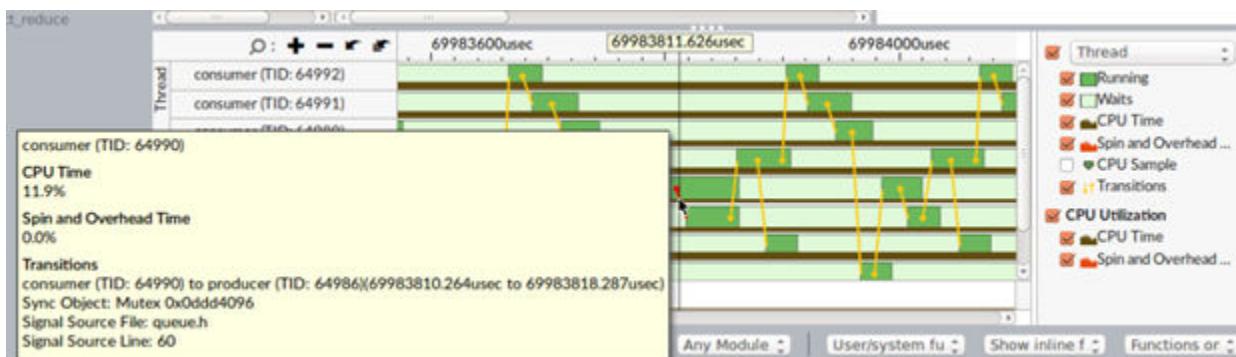
Review the Timeline

The **Timeline** pane at the bottom of the **Bottom-up/Top-down Tree** windows shows the thread behavior in your application and how CPU utilization metrics are changing over time. Analyze the data, select the problem area, and zoom in to selection using the context menu options. VTune Profiler calculates the overall **CPU Utilization** metric as the sum of CPU time per each thread of the Threads area. Maximum **CPU Utilization** value is equal to [number of processor cores] x 100%.

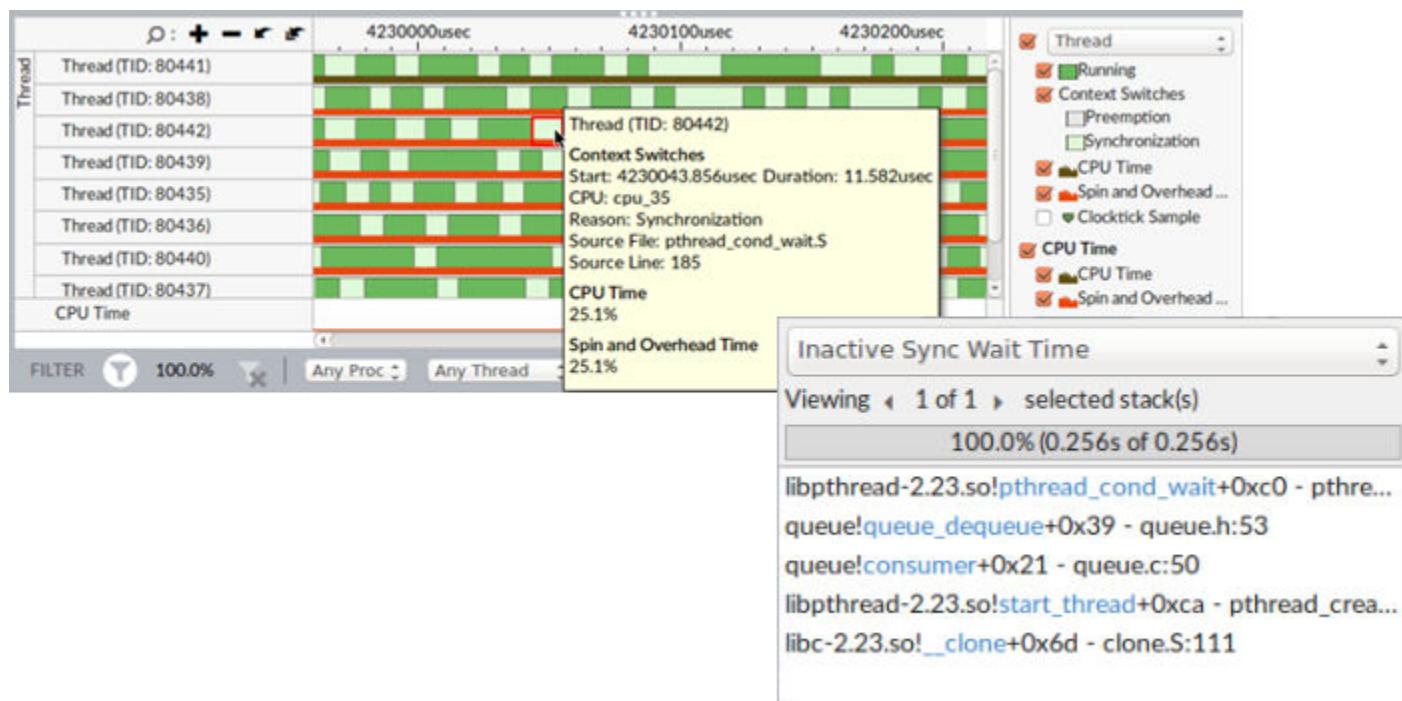


To understand what your application was doing during a particular time frame, select this range on the timeline, right-click and choose **Zoom In and Filter In by Selection**. VTune Profiler will display functions or sync objects used during this time range.

For User-mode Sampling and Tracing collection mode, select the **Transitions** option on the timeline to explore thread interactions.



For Hardware Event-based Sampling and Context Switches mode, the timeline is helpful in exploring inactive waits. Select an inactive time area on the timeline to display the wait stack on the stack pane that corresponds to the context switch.



Analyze Source

Double-click the hottest synchronization object (with the highest Wait Time and Wait Count values) to view its related source code file in the **Source/Assembly** window. From the **Timeline** pane, you can double-click the transition line to open the call site for this transition. You can open the code editor directly from the VTune Profiler and edit your code.

Explore Other Analysis Types

- Run the [comparison analysis](#) to understand the performance gain you obtain after your optimization.
- Run Microarchitecture Exploration analysis to identify hardware issues affecting the performance of your application.

See Also

[Analyze Performance](#)

[Source Code Analysis](#)

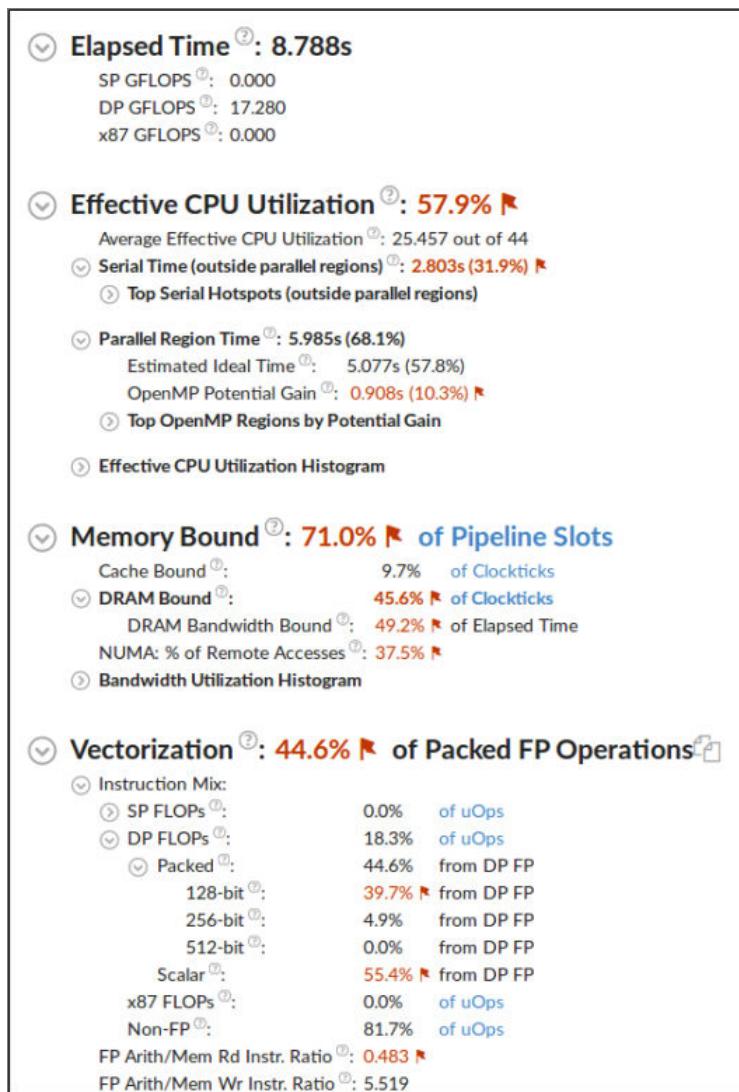
View Stacks

HPC Performance Characterization Analysis

Use the HPC Performance Characterization analysis to identify how effectively your compute-intensive application uses CPU, memory, and floating-point operation hardware resources.

How It Works

The HPC Performance Characterization analysis type can be used as a starting point for understanding the performance aspects of your application. Additional scalability metrics are available for applications that use Intel OpenMP* or Intel MPI runtime libraries.



During HPC Performance Characterization analysis, the Intel® VTune™ Profiler data collector profiles your application using [event-based sampling collection](#). OpenMP analysis metrics for Intel OpenMP runtime library are based on User API instrumentation enabled in the runtime library.

Typically the collector will gather data for a specified application, but it can collect system-wide performance data with limited detail if required.

NOTE

Vectorization and GFLOPS metrics are supported on Intel® microarchitectures formerly code named Ivy Bridge, Broadwell, and Skylake. Limited support is available for Intel® Xeon Phi™ processors formerly code named Knights Landing. The metrics are not currently available on 4th Generation Intel processors. Expand the **Details** section on the analysis configuration pane to view the processor family available on your system.

The analysis can be run from within the VTune Profiler GUI or [from the command line](#).

NOTE

Intel® VTune™ Profiler is a new renamed version of Intel® VTune™ Amplifier.

Configure and Run Analysis

To configure options for the HPC Performance Characterization analysis:

Prerequisites: Create a project.

1. Click the



(standalone GUI)/

(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From **HOW** pane, click the



Browse button and select **HPC Performance Characterization**.

3. Configure the following options:

CPU sampling interval, ms field	Specify an interval (in milliseconds) between CPU samples. Possible values - 0.01-1000 . The default value is 1 .
Collect stacks check box	Enable advanced collection of call stacks and thread context switches. The option is disabled by default.
Analyze memory bandwidth check box	Collect the data required to compute memory bandwidth. The option is enabled by default.
Evaluate max DRAM bandwidth check box	Evaluate maximum achievable local DRAM bandwidth before the collection starts. This data is used to scale bandwidth metrics on the timeline and calculate thresholds. The option is enabled by default.
Analyze OpenMP regions check box	Instrument and analyze OpenMP regions to detect inefficiencies such as imbalance, lock contention, or overhead on performing scheduling, reduction and atomic operations.

The option is enabled by default.

Details button

Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable **additional settings** for the analysis, you need to [create a custom configuration](#) by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

NOTE

You may [generate the command line](#) for this configuration using the



Command Line button at the bottom.

4. Click the



Start button to run the analysis.

View Data

Use the HPC Performance Characterization viewpoint to review the following:

- Effective Physical Core Utilization: Explore application parallel efficiency by looking at physical core utilization by the application code execution. Look for scalability problems involving the use of serial time versus parallel time, tuning potential for OpenMP regions, and MPI imbalance.
- Memory Bound: Evaluate whether the application is memory bound. To understand deeper problems, run the [Memory Access Analysis](#) to identify specific memory objects causing issues.
- Vectorization: Determine if floating-point loops are bandwidth bound or vectorized. For bandwidth bound loops/functions, run the Memory Access Analysis to reduce bandwidth consumption. For vectorization optimization opportunities, use the Intel Advisor to run a vectorization analysis.
- Intel® Omni-Path Fabric Usage: Identify performance bottlenecks caused by reaching the interconnect limits.

Use the Analyzing an OpenMP* and MPI Application tutorial to review basic steps for tuning a hybrid application. The tutorial is available from the Intel Developer Zone at <https://www.intel.com/content/www/us/en/docs/vtune-profiler/tutorial-vtac-mpi-openmp/current/overview.html>.

See Also

[HPC Performance Characterization View](#)

[Cookbook: OpenMP* Code Analysis Method](#)

[Syntax](#)

HPC Performance Characterization View

Use the HPC Performance Characterization viewpoint to estimate CPU usage, memory efficiency, and floating-point utilization for compute-intensive or throughput applications. Compute-intensive or throughput applications should use hardware resources efficiently for the duration of their elapsed time. Use the [HPC Performance Characterization](#) analysis as a starting point for optimizing application performance and runtime.

Follow these steps to interpret the performance data provided in the HPC Performance Characterization viewpoint:

1. [Define a Performance Baseline](#)

- [2. Determine Optimization Opportunities](#)
- [3. Analyze Source](#)
- [4. Analyze Process/Thread Affinity](#)
- [5. Explore Other Analysis Types](#)

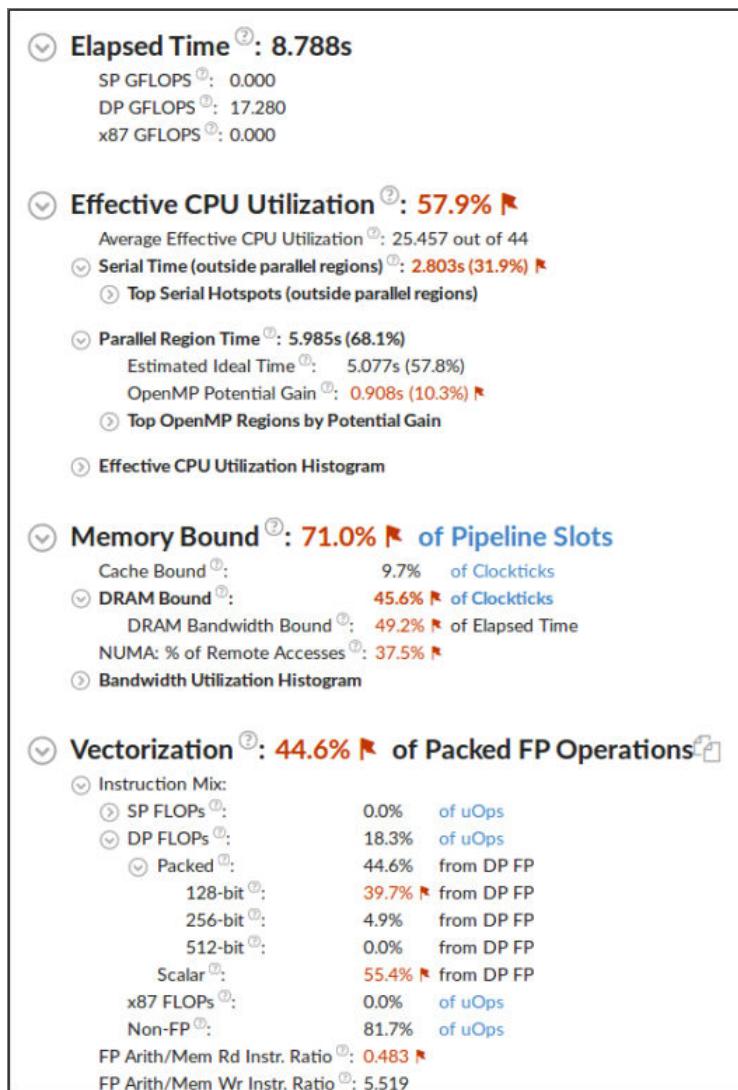
Tip

To review basic steps for tuning a hybrid application, follow the [Analyzing an OpenMP* and MPI Application](#) tutorial .

1. Define a Performance Baseline

Start with exploring the [Summary window](#) that provides general information on your application execution. Key areas for optimization include the elapsed time and floating-point operation per second counts (single precision, double precision, and legacy x87). Red text indicates an area of potential optimization. Hover over a flag to learn more about how to improve your code.

Use the Elapsed Time and GFLOPS values as a baseline for comparison of versions before and after optimization.



2. Determine Optimization Opportunities

Review the **Summary** window to find the key optimization opportunities for your application. Performance metrics that can be improved are marked in red. Issues identified could include Effective Physical Core Utilization, Memory Bound, Vectorization, or a combination of these. The following sections provide suggested next steps for each performance aspect:

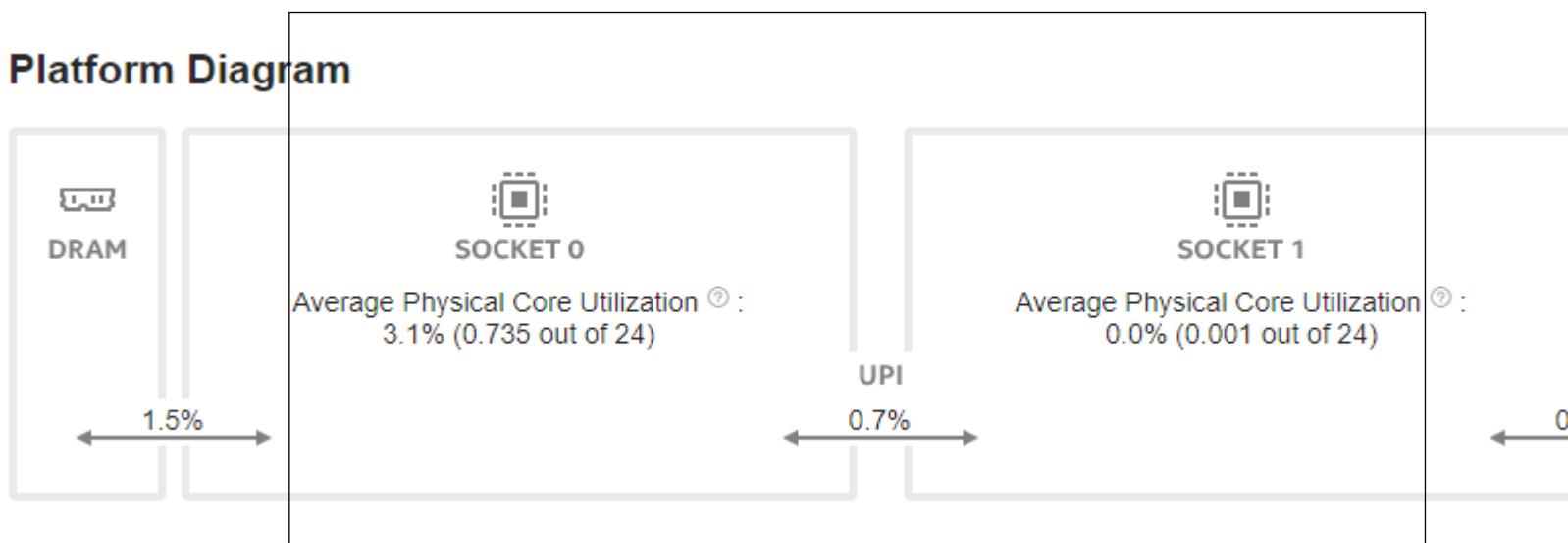
- [Topology, Memory, and Cross-Socket Bandwidth](#)
- [CPU Utilization](#)
- [GPU Utilization](#)
- [Memory Bound](#)
- [Vectorization](#)

Topology, Memory, and Cross-Socket Bandwidth

Start your performance analysis in the **Summary** window of the HPC Performance Characterization viewpoint. Here, the **Platform Diagram** displays system topology and utilization metrics for DRAM, Intel® Ultra Path Interconnect (Intel® UPI) links, and physical cores.

Sub-optimal application topology can result in induced DRAM and Intel® QuickPath Interconnect (Intel® QPI) or Intel® Ultra Path Interconnect (Intel® UPI) cross-socket traffic. These incidents can limit performance.

Platform Diagram



NOTE

The platform diagram is available for:

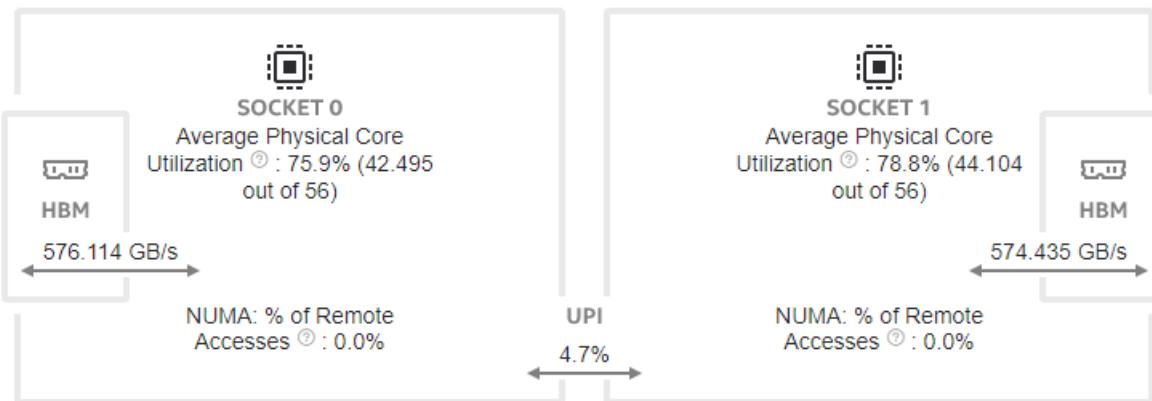
- All client platforms.
- Server platforms based on Intel® microarchitecture code named Skylake, with up to four sockets.
- Server platforms based on Intel® microarchitecture code named Sapphire Rapids.

High Bandwidth Memory Data in Platform Diagram

For server platforms based on Intel® microarchitecture code named Sapphire Rapids, the **Platform Diagram** also includes information about High Bandwidth Memory (HBM). Use this information to distinguish from DRAM-specific utilization in the diagram.

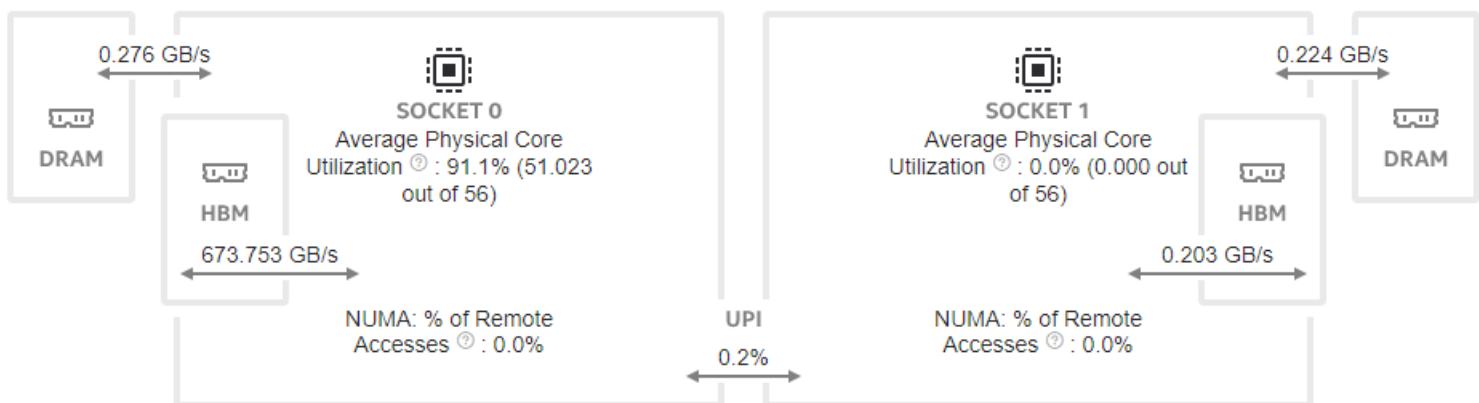
For example, this diagram shows information about HBM mode utilization, where the system has no DRAM.

Platform Diagram



Here is an example of the **Platform Diagram** data in a system that has both HBM and DRAM.

Platform Diagram



If you selected the **Evaluate max DRAM bandwidth** option in your analysis configuration, the **Platform Diagram** shows the average DRAM utilization. Otherwise, the diagram shows the average DRAM bandwidth.

The **Average UPI Utilization** metric displays UPI utilization in terms of transmit. Irrespective of the number of UPI links that connect a pair of packages, the Platform Diagram shows a single cross-socket connection. If there are several links, the diagram displays the maximum value.

On top of each socket, the **Average Physical Core Utilization** metric indicates the utilization of physical cores by computations of the application under analysis.

Once you examine the topology and utilization information in the diagram, focus on other sections in the **Summary** window and then switch to the **Bottom-up** window.

CPU Utilization

Effective Physical Core Utilization ^①: 85.5% (37.640 out of 44)

Effective Logical Core Utilization ^②: 43.2% (38.025 out of 88)

Serial Time (outside parallel regions) ^③: 0.222s (0.8%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

Function	Module	Serial CPU Time ^④
clear_page_c_e	vmlinux	0.089s
init_arr	matrix_multiply_naive.llc	0.048s
init_arr	matrix_multiply_naive.llc	0.010s
func@0x247b0	libltnotify_collector.so	0.003s
release_pages	vmlinux	0.002s
[Others]		0.019s

Parallel Region Time ^⑤: 26.319s (99.2%)

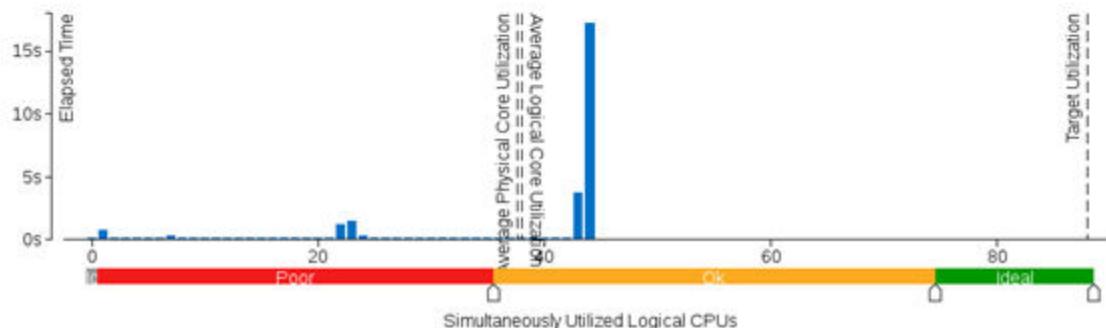
Estimated Ideal Time ^⑥: 23.163s (87.3%)

OpenMP Potential Gain ^⑦: 3.156s (11.9%)

Top OpenMP Regions by Potential Gain

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



- Explore the **Effective Physical Core Utilization** metric as a measure of the parallel efficiency of the application. A value of 100% means that the application code execution uses all available physical cores. If the value is less than 100%, it is worth looking at the second level metrics to discover reasons for parallel inefficiency.
- Learn about opportunities to use the logical cores. In some cases, using logical cores leads to increases in application concurrency and overall performance improvements.

This table provides additional CPU utilization information for specific applications or hardware configurations.

Application or Architecture Type	Notes and Recommendations
Certain Intel® processors like Intel® Xeon Phi™ or Intel Atom®, or systems where Intel Hyper-Threading Technology (Intel HT Technology) is OFF or absent	The metric breakdown between physical and logical core utilization is not available. In these cases, a single Effective CPU Utilization metric is displayed to show parallel execution efficiency.
Applications that do not use OpenMP* or MPI runtime libraries	<ul style="list-style-type: none"> Review the Effective CPU Utilization Histogram, which displays the Elapsed Time of your application, broken down by CPU utilization levels.

Application or Architecture Type	Notes and Recommendations
Applications with Intel OpenMP*	<ul style="list-style-type: none"> • Use the data in the Bottom-up and Top-down Tree windows to identify the most time-consuming functions in your application by CPU utilization. Focus on the functions with the largest CPU time and low CPU utilization level as your candidates for optimization (for example, parallelization). • Compare the serial time to the parallel region time. If the serial portion is significant, consider options to minimize serial execution, either by introducing more parallelism or by doing algorithm or microarchitecture tuning for sections that seem unavoidably serial. For high thread-count machines, serial sections have a severe negative impact on potential scaling (Amdahl's Law) and should be minimized as much as possible. Look at serial hotspots to define candidates for further parallelization. • Review the OpenMP Potential Gain to estimate the efficiency of OpenMP parallelization in the parallel part of the code. The Potential Gain metric estimates the elapsed time between the actual measurement and an idealized execution of parallel regions, assuming perfectly balanced threads and zero overhead of the OpenMP runtime on work arrangement. Use this data to understand the maximum time that you may save by improving OpenMP parallelism. If Potential Gain for a region is significant, you can go deeper and select the link on a region name to navigate to the Bottom-up window employing an OpenMP Region dominant grouping and the region of interest selection. • Consider running Threading analysis when there are multiple locks used in one parallel construct to find the performance impact of a particular lock.
MPI applications	<p>Review the MPI Imbalance metric that shows the CPU time spent by ranks spinning in waits on communication operations, normalized by number of ranks on the profiling node. The metric issue detection description generation is based on minimal MPI Busy Wait time by ranks. If the minimal MPI Busy wait time by ranks is not significant, then the rank with the minimal time most likely lies on the critical path of application execution. In this case, review the CPU utilization metrics for this rank.</p>

Application or Architecture Type	Notes and Recommendations
Hybrid MPI + OpenMP applications	<p>The sub-section MPI Rank on Critical Path shows OpenMP efficiency metrics like Serial Time (outside of any OpenMP region), Parallel Region time, and OpenMP Potential Gain. If the minimal MPI Busy Wait time is significant, it can be a result of suboptimal communication schema between ranks or imbalance triggered by another node. In this case, use Intel® Trace Analyzer and Collector for in depth analysis of communication schema.</p>

GPU Utilization

GPU utilization metrics display when:

- Your application makes use of a GPU.
- Your system is configured to collect GPU data. See [Set Up System for GPU Analysis](#).

Under **Elapsed Time**, the **GPU** section presents an overview of how your application offloads work to the GPU.



- The **GPU Stack Utilization** metric indicates if the GPU was idle at any point during data collection. A value of 100% implies that your application offloaded work to the GPU throughout the duration of data collection. Anything lower presents an opportunity to improve GPU utilization. The representation of GPU utilization as the number of used GPU stacks provides context in terms of hardware.
- The **GPU Accumulated Time** metric indicates the sum total of times spent by GPU stacks which had at least one execution thread scheduled. If there are multiple GPU stacks available in the system, the GPU Accumulated Time may be larger than the **Elapsed Time**.
- The **IPC Rate** metric indicates the average number of instructions per cycle processed by the two FPU pipelines of Intel® Integrated Graphics. To have your workload fully utilize the floating-point capability of the GPU, the IPC Rate should be closer to 2.

Next, look into the **GPU Stack Utilization** section. Here, you can understand if your workload can use the GPU more efficiently.

GPU Stack Utilization: 56.7% ↘

EU State ↗:

Active ↗: 49.0%
 Stalled ↗: 48.4% ↗
 Idle ↗: 2.7%
 Occupancy ↗: 86.1% ↗ of peak value

Offload Time: 70.3% (2.752s) of elapsed time

Compute: 80.8% (2.223s) of offload time
 Data Transfer: 12.3% (0.338s) of offload time
 Overhead: 6.9% (0.191s) of offload time

Top OpenMP Offload Regions

OpenMP Offload Region	Offload Time	Percentage of Elapsed Time	Data Transfer	Overhead	EU Array Active ↗
main\$omp\$target\$region:dvc=0@unknown:16	2.224s	56.8%	0.000s	0.000s	49.0%
main\$omp\$target\$region:dvc=0@unknown:15	0.529s	13.5%	0.338s	0.190s	0.0%

*N/A is applied to non-summable metrics.

Ideally, your GPU stack utilization should be 100%. If the **GPU Stack Utilization** metric is <100%, there were cycles where the GPU had no execution threads scheduled.

- **EU State** breaks down the activity of GPU execution units. Check here to see if they were stalled or idle when processing your workload.
- **Occupancy** is a measure of the efficiency of scheduling the GPU thread. A value below 100% recommends that you tune the sizes of the work items in your workload. Consider running the [GPU Offload Analysis](#). This provides an insight into computing tasks running on the GPU as well as additional GPU-related performance metrics.

If your application offloads code via Intel OpenMP*, check the **Offload Time** section:

- The **Offload Time** metric displays the total duration of the OpenMP offload regions in your workload. If **Offload Time** is below 100%, consider offloading more code to the GPU.
- The **Compute**, **Data Transfer**, and **Overhead** metrics help you understand what constitutes the **Offload Time**. Ideally, the **Compute** portion should be 100%. If the **Data Transfer** component is significant, try to transfer less data between the host and the GPU.

In the **Top OpenMP Offload Regions** section, review the breakdown of offload and GPU metrics by OpenMP offload region. Focus on regions that take up a significant portion of the **Offload Time**.

The names of the OpenMP offload regions use this format:

<func_name>\$omp\$target\$region:dvc=<device_number>@<file_name>:<line_number>

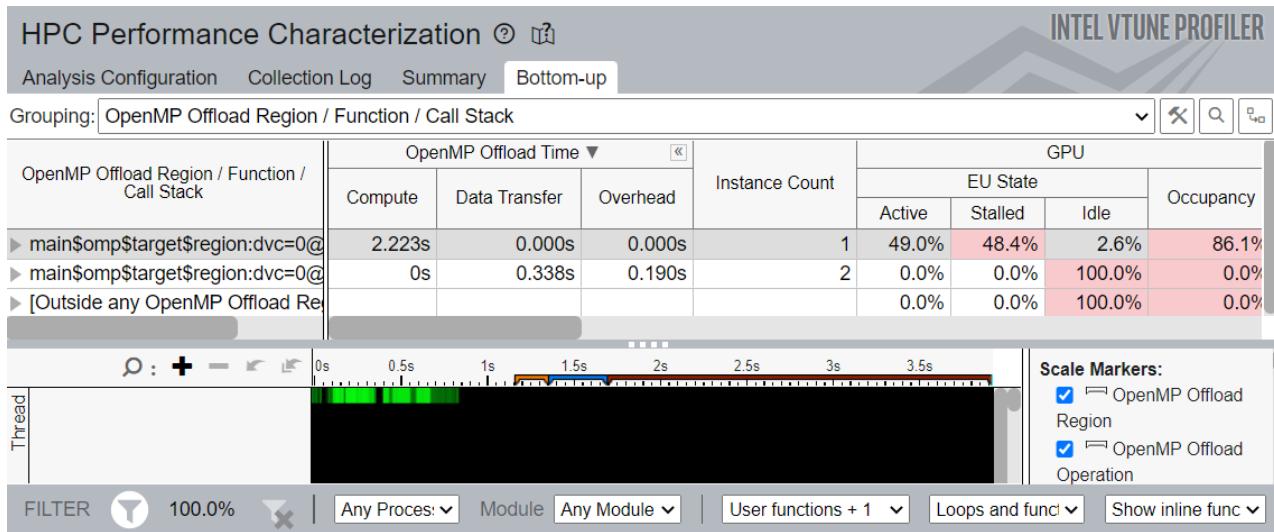
where:

- *func_name* is the name of the source function where the OpenMP target directive is declared.
- *device_number* is the internal OpenMP device number where the offload was targeted.
- *file_name* and *line_number* constitute the source location of the OpenMP target directive.

When you compile your OpenMP application, the *func_name*, *file_name*, and *line_number* fields require you to pass debug information options to the Intel Compiler. If debug information is absent, these fields get default values.

Field	Compiler Options to Enable		Default Value
	Linux OS	Windows OS	
<i>line_number</i>	-g	/Zi	0
<i>func_name</i>	-g	/Zi	unknown
<i>file_name</i>	-g -mllvm -parallel-source-info=2	/Zi -mllvm -parallel-source-info=2	unknown

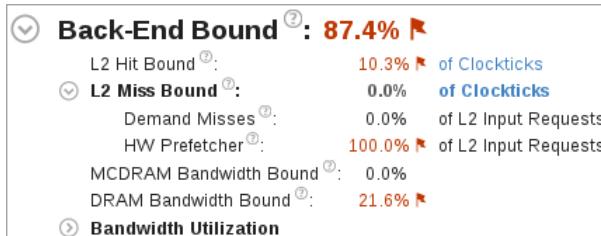
For applications that use OpenMP offload, the **Bottom-up** window displays additional information.



- Group by **OpenMP Offload Region**. In this grouping, the grid displays:
 - **OpenMP Offload Time** metrics
 - **Instance Count**
 - **GPU** metrics
- The timeline view displays ruler markers that indicate the span of **OpenMP Offload Regions** and **OpenMP Offload Operations** within those regions.

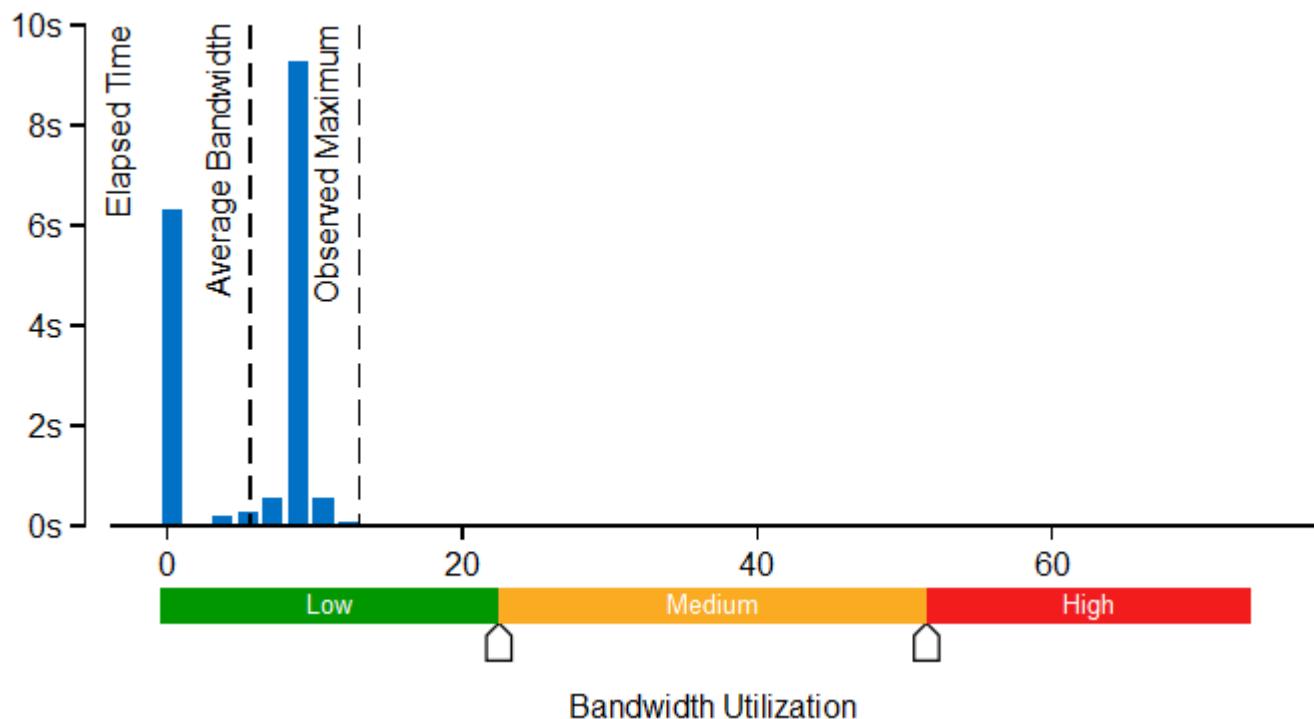
Memory Bound

- A high **Memory Bound** value might indicate that a significant portion of execution time was lost while fetching data. The section shows a fraction of cycles that were lost in stalls being served in different cache hierarchy levels (L1, L2, L3) or fetching data from DRAM. For last level cache misses that lead to DRAM, it is important to distinguish if the stalls were because of a memory bandwidth limit since they can require specific optimization techniques when compared to latency bound stalls. VTune Profiler shows a hint about identifying this issue in the DRAM Bound metric issue description. This section also offers the percentage of accesses to a remote socket compared to a local socket to see if memory stalls can be connected with NUMA issues.



- A high **L2 Hit Bound** or **L2 Miss Bound** value indicates that a high ratio of cycles were spent handling L2 hits or misses.

- The **L2 Miss Bound** metric does not take into account data brought into the L2 cache by the hardware prefetcher. However, in some cases the hardware prefetcher can generate significant DRAM/MCDRAM traffic and saturate the bandwidth. The **Demand Misses** and **HW Prefetcher** metrics show the percentages of all L2 cache input requests that are caused by demand loads or the hardware prefetcher.
- A high **DRAM Bandwidth Bound** or **MCDRAM Bandwidth Bound** value indicates that a large percentage of the overall elapsed time was spent with high bandwidth utilization. A high **DRAM Bandwidth Bound** value is an opportunity to run the [Memory Access](#) analysis to identify data structures that can be allocated in high bandwidth memory (MCDRAM), if it is available.
- The **Bandwidth Utilization Histogram** shows how much time the system bandwidth was utilized by a certain value (Bandwidth Domain) and provides thresholds to categorize bandwidth utilization as High, Medium and Low. The thresholds are calculated based on benchmarks that calculate the maximum value. You can also set the threshold by moving sliders at the bottom of the histogram. The modified values are applied to all subsequent results in the project.



- Switch to the **Bottom-up** window and review the **Memory Bound** columns in the grid to determine optimization opportunities.
- If your application is memory bound, consider running a [Memory Access](#) analysis for deeper metrics and the ability to correlate these metrics with memory objects.

Vectorization

- The Vectorization metric represents the percentage of packed (vectorized) floating point operations. 0% means that the code is fully scalar while 100% means the code is fully vectorized. The metric does not take into account the actual vector length used by the code for vector instructions. As a result, if the code is fully vectorized and uses a legacy instruction set that loaded only half a vector length, the Vectorization metric still shows 100%.

Low vectorization means that a significant fraction of floating point operations are not vectorized. Use Intel® Advisor to understand possible reasons why the code was not vectorized.

The second level metrics allow for rough estimates of the size of floating point work with particular precision and see the actual vector length of vector instructions with particular precision. Partial vector length can provide information about legacy instruction set usage and show an opportunity to recompile the code with modern instruction set, which can lead to additional performance improvement. Relevant metrics might include:

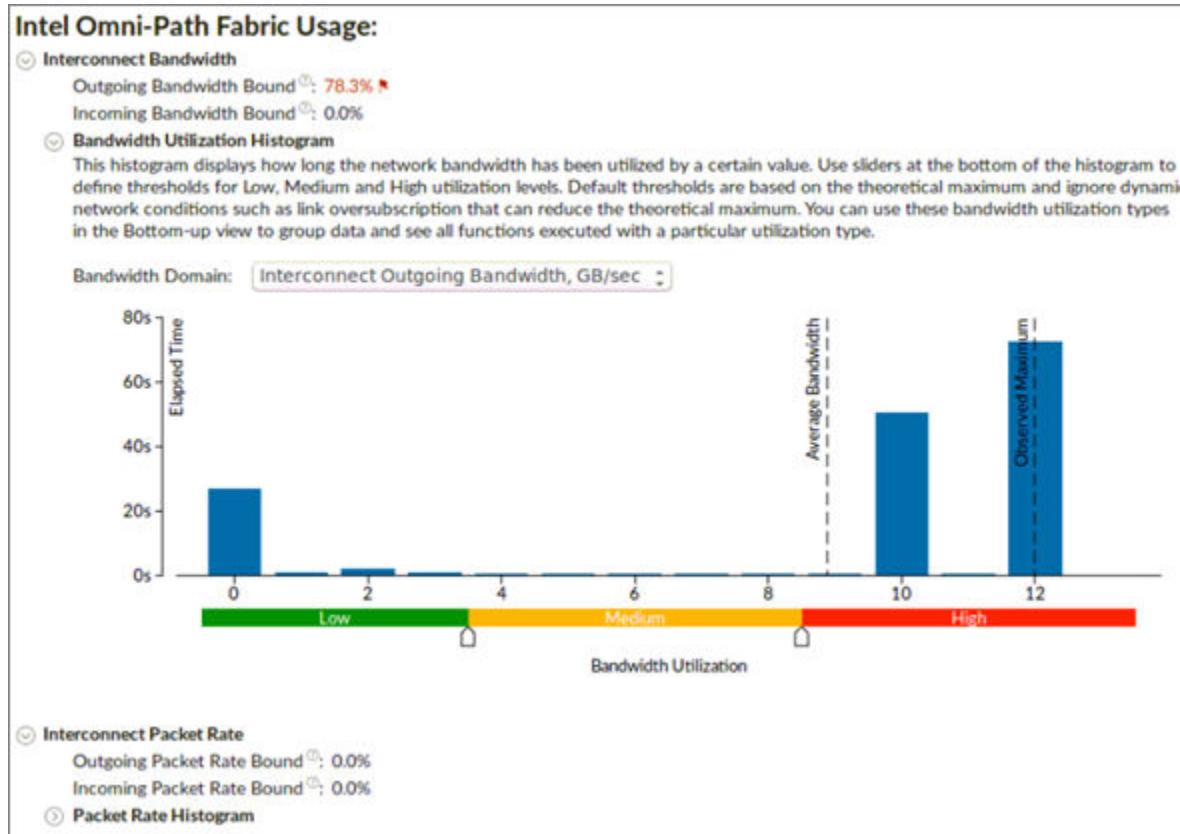
- Instruction Mix
- FP Arithmetic Instructions per Memory Read or Write
- The **Top Loops/Functions with FPU Usage by CPU Time** table shows the top functions that contain floating point operations sorted by CPU time and allows for a quick estimate of the fraction of vectorized code, the vector instruction set used in the loop/function, and the loop type.

Top Loops/Functions with FPU Usage by CPU Time					
This section provides information for the most time consuming loops/functions with floating point operations.					
Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
[Loop at line 526 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec, std::minife::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@518]	119.475s	15.7%	55.8%	44.2%	AVX(128); FMA(128)
[Loop at line 204 in miniFE::daxpy<miniFE::Vector<double, int, int>>>\$omp\$parallel_for@204]	19.962s				AVX(256); FMA(256)
[Loop at line 526 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec, std::minife::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@518]	16.779s	25.3%	98.9%	1.1%	
[Loop at line 519 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec, std::minife::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@518]	10.309s	26.4%	93.2%	6.8%	AVX(128); AVX512F_128(128); FMA(128)
[Loop at line 248 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec, std::minife::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@248]	8.054s	10.7%	100.0%	0.0%	AVX(256); FMA(256)
[Others]	21.160s	58.5%	8.2%	91.8%	

*NA is applied to non-summable metrics.

Intel® Omni-Path Fabric Usage

Intel® Omni-Path Fabric (Intel® OP Fabric) metrics are available for analysis of compute nodes equipped with Intel OP Fabric interconnect. They help to understand if MPI communication has bottlenecks connected with reaching interconnect hardware limits. The section shows two aspects interconnect usage: bandwidth and packet rate. Both bandwidth and packet rate split the data into outgoing and incoming data because the interconnect is bi-directional. A bottleneck can be connected with one of the directions.



- **Outgoing and Incoming Bandwidth Bound** metrics shows the percent of elapsed time that an application spent in communication closer to or reaching interconnect bandwidth limit.
- **Bandwidth Utilization Histogram** shows how much time the interconnect bandwidth was utilized by a certain value (Bandwidth Domain) and provides thresholds to categorize bandwidth utilization as High, Medium, and Low.
- **Outgoing and Incoming Packet Rate** metrics shows the percent of elapsed time that an application spent in communication closer to or reaching interconnect packet rate limit.
- **Packet Rate Histogram** shows how much time the interconnect packet rate was reached by a certain value and provides thresholds to categorize packet rate as High, Medium, and Low.

3. Analyze Source

Double-click the function you want to optimize to view its related source code file in the Source/Assembly window. You can open the code editor directly from the Intel® VTune™ Profiler and edit your code (for example, minimizing the number of calls to the hotspot function).

4. Analyze Process/Thread Affinity

If the results show inefficient core utilization or NUMA effects, it can be helpful to know if and how threads are pinned to processor cores.

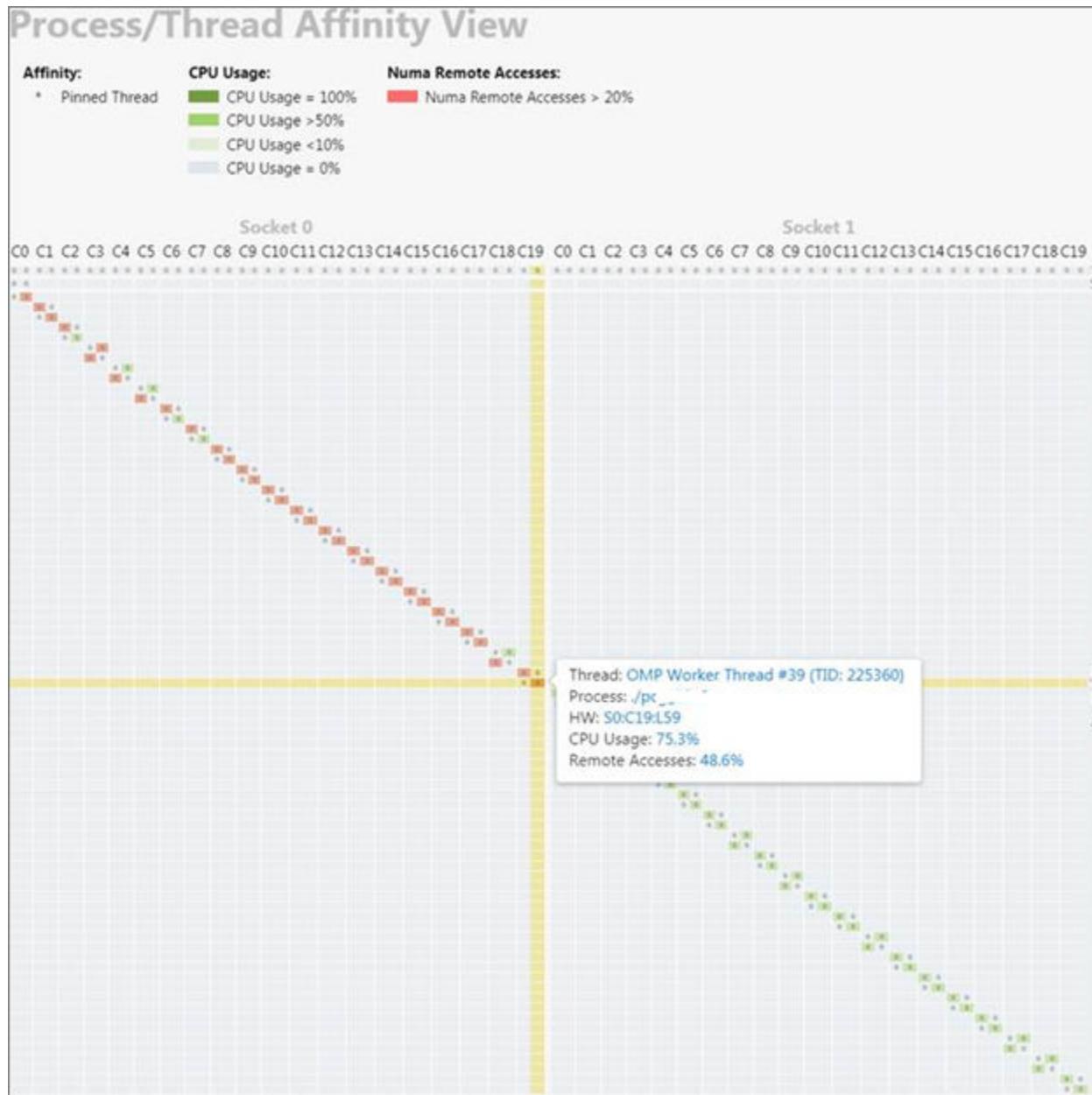
The thread pinning or affinity can be applied by parallel runtimes (such as MPI), by using environment variables, or by using APIs from parallel runtimes or the operating system. Use the knob **Collect thread affinity** in the VTune Profiler GUI or `-knob collect-affinity=true` in the command line to activate affinity collection for the HPC Performance Characterization analysis. With this option enabled it is possible to generate a thread affinity command line report that shows thread pinning to sockets, physical cores, and logical cores. Note that affinity information is collected at the end of the thread lifetime, so the resulting data may not show the whole issue for dynamic affinity that is changed during the thread lifetime.

A preview HTML report is available to see process/thread affinity along with thread CPU execution and remote accesses. Use the following command to generate the preview HTML report:

```
vtune -report affinity -format=html -r <result_dir>
```

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.



5. Explore Other Analysis Types

- Run a [Memory Access analysis](#) to view more detail about cache bound and memory bound issues affecting the performance of your application.
- Use the Intel Advisor to analyze the application for vectorization optimization.

See Also

[Analyze Performance](#)

[Viewing Source](#)

[Reference for Performance Metrics](#)

[Running HPC Performance Characterization Analysis from the Command Line](#)

Input and Output Analysis

Use the Input and Output analysis of Intel® VTune™ Profiler to locate performance bottlenecks in I/O-intensive applications at both hardware and software levels.

The Input and Output analysis of Intel® VTune™ Profiler helps to determine:

- Platform I/O consumption by external PCIe devices and integrated accelerators:
 - I/O bandwidth consumption, including [Intel® Data Direct I/O Technology](#) (Intel® DDIO) and Memory-Mapped I/O traffic.
 - [Utilization efficiency of Intel® DDIO](#)
 - [Memory bandwidth](#) consumption.
 - [Intel® Ultra Path Interconnect \(Intel® UPI\)](#) bandwidth consumption.
 - Software data plane utilization.

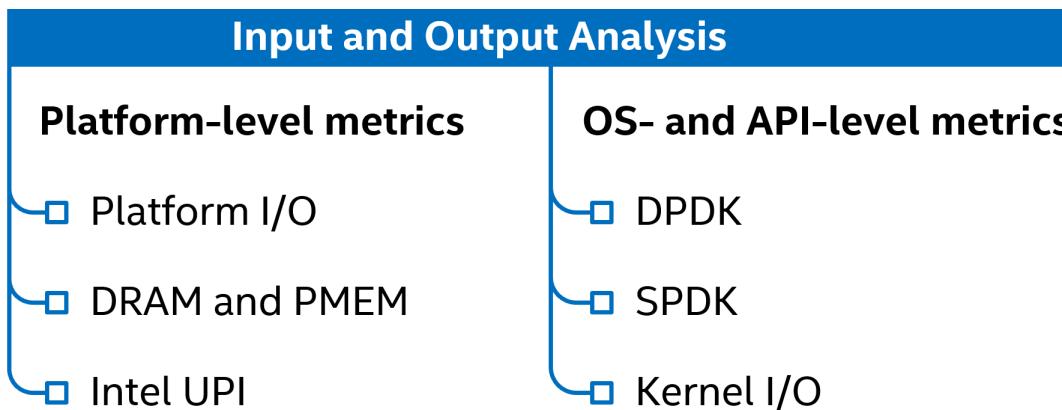
The Input and Output analysis features two main types of performance metrics:

- Platform-level metrics** — application-agnostic hardware event-based metrics.
- OS- and API-specific metrics** — performance metrics for software data planes—[DPDK](#) and [SPDK](#)—and the [Linux* kernel I/O stack](#).

Linux* and FreeBSD* targets are supported.

NOTE

The full set of Input and Output analysis metrics is available on Intel® Xeon® processors only.



Configure and Run Analysis

NOTE

On FreeBSD systems, the graphical user interface of VTune Profiler is not supported. You can still configure and run the analysis from a Linux* or Windows* system using remote SSH capabilities, or collect the result locally from the CLI. For more information on available options, see [FreeBSD Targets](#).

1. Launch VTune Profiler and, optionally, create a new project.
2. Click the **Configure Analysis** button.
3. In the **WHERE** pane, select the target system to profile.
4. In the **HOW** pane, select **Input and Output**.
5. In the **WHAT** pane, specify your analysis target (application, process, or system).
6. Depending on your target app and analysis purpose, choose any of the configuration options described in sections below.
7. Click **Start** to run the analysis.

VTune Profiler collects the data, generates a result, and opens the result with that displays data according to configuration.

To run the Input and Output analysis from the command line, enter:

```
vtune -collect io [-knob <value>] -- <target> [target_options]
```

For details, see the [io command line reference](#).

Platform-Level Metrics

To collect hardware event-based metrics, either [load the Intel sampling driver](#) or [configure driverless hardware event collection](#) (Linux targets only).

IO Analysis Configuration Check Box	Features	Prerequisites/Applicability
Analyze PCIe traffic	Calculate inbound I/O (Intel® Data Direct I/O) and outbound I/O (Memory-Mapped I/O) bandwidth.	<p>Available on server platforms..</p> <p>The granularity of I/O bandwidth metrics depends on CPU model, collector used, and user privileges:</p> <ul style="list-style-type: none"> • Code names: Haswell, Broadwell. • Granularity: by CPU socket (package) in any case. • Code names: Skylake, Cascade Lake, Cooper Lake. • Granularity: <ul style="list-style-type: none"> • With sampling driver: I/O device (external PCIe or integrated accelerator). • Driverless with root: I/O device (external PCIe or integrated accelerator). • Driverless without root: before kernel v5.10—CPU socket; on kernels v5.10 and newer—I/O device.

IO Analysis Configuration Check Box	Features	Prerequisites/Applicability
	<p>Calculate L3 hits and misses of inbound I/O requests (Intel® DDIO hits/misses).</p>	<ul style="list-style-type: none"> • Code names: Snow Ridge, Ice Lake • Granularity: <ul style="list-style-type: none"> • With sampling driver: I/O device (external PCIe or integrated accelerator). • Driverless with root: I/O device (external PCIe or integrated accelerator). • Driverless without root: before kernel v5.14—CPU socket; on kernels v5.14 and newer—I/O device.
	<p>Calculate average latency of inbound I/O reads and writes, as well as CPU/IO conflicts.</p>	<p>Available on server platforms based on Intel® microarchitecture code named Haswell and newer.</p> <p>The granularity of inbound I/O request L3 hit/miss metrics depends on CPU model, collector used and user privileges:</p> <ul style="list-style-type: none"> • Code names: Haswell, Broadwell. <ul style="list-style-type: none"> • Granularity: by CPU socket (package) in any case. • Code names: Skylake, Cascade Lake, Cooper Lake. <ul style="list-style-type: none"> • Granularity: <ul style="list-style-type: none"> • With sampling driver: set of I/O devices¹. • Driverless with root: set of I/O devices¹. • Driverless without root: CPU socket (package). • Code names: Snow Ridge, Ice Lake <ul style="list-style-type: none"> • Granularity: <ul style="list-style-type: none"> • With sampling driver: set of I/O devices¹. • Driverless with root: set of I/O devices¹. • Driverless without root: CPU socket (package). <p>¹—commonly, a set combines all devices sharing the same 16 PCIe lanes.</p> <p>Available on server platforms based on Intel® microarchitecture code named Skylake and newer.</p>

IO Analysis Configuration Check Box	Features	Prerequisites/Applicability
		<p>The granularity of latency and CPU/IO conflicts metrics depends on CPU model, collector used and user privileges:</p> <ul style="list-style-type: none"> • Code names: Skylake, Cascade Lake, Cooper Lake. • Granularity: <ul style="list-style-type: none"> • With sampling driver: set of I/O devices¹. • Driverless with root: set of I/O devices^{1, 2}. • Driverless without root: CPU socket (package)². • Code names: Snow Ridge, Ice Lake • Granularity: <ul style="list-style-type: none"> • With sampling driver: set of I/O devices¹. • Driverless with root: set of I/O devices¹. • Driverless without root: CPU socket (package).
Locate MMIO accesses	Locate code that induces outbound I/O traffic by accessing device memory through the MMIO address space.	<p>Available on server platforms based on Intel® microarchitecture code named Skylake and newer.</p> <ul style="list-style-type: none"> • This option is not available in Profile System mode. • This option is available on Linux systems only.
Analyze Intel® VT-d	Calculate performance metrics for Intel® Virtualization Technology for Directed I/O (Intel VT-d).	<p>Available on server platforms based on Intel® microarchitecture code named Ice Lake and newer.</p> <p>The Intel VT-d metrics granularity depends on collector used and user privileges:</p> <ul style="list-style-type: none"> • Code names: Snow Ridge, Ice Lake • Granularity:

IO Analysis Configuration Check Box	Features	Prerequisites/Applicability
		<ul style="list-style-type: none"> With sampling driver: set of I/O devices¹. Driverless with root: set of I/O devices¹. Driverless without root: before kernel v5.14—CPU socket; on kernels v5.14 and newer—set of I/O devices¹.
Analyze memory and cross-socket bandwidth	Calculate DRAM, Persistent Memory, and Intel® Ultra Path Interconnect (Intel® UPI) or Intel® QuickPath Interconnect (Intel® QPI) bandwidth.	<p>¹—commonly, a set combines all devices sharing the same 16 PCIe lanes.</p> <p>While DRAM bandwidth data is always collected, persistent memory bandwidth and Intel® UPI / Intel® QPI cross-socket bandwidth data is only collected when applicable to the system.</p>
Evaluate max DRAM bandwidth	<p>Evaluate the maximum achievable local DRAM bandwidth before the collection starts.</p> <p>This data is used to scale bandwidth metrics on the Platform Diagram and timeline and to calculate thresholds.</p>	Not available on FreeBSD systems.

OS- and API-Level Metrics

IO Analysis Configuration Check Box	Prerequisites/Applicability
DPDK	<p>Make sure DPDK is built with VTune Profiler support enabled.</p> <p>When profiling DPDK as FD.io VPP plugin, modify the <code>DPDK_MESON_ARGS</code> variable in <code>build/external/packages/dpdk.mk</code> with the same flags as described in Profiling with VTune section.</p> <p>Not available for FreeBSD targets. Not available in system-wide mode.</p>
SPDK	<p>Make sure SPDK is built using the <code>--with-vtune</code> advanced build option.</p> <p>When profiling in Attach to Process mode, make sure to set up the environment variables before launching the application.</p> <p>Not available in Profile System mode.</p>
Kernel I/O	<p>To collect these metrics, VTune Profiler enables FTrace* collection that requires access to <code>debugfs</code>. On some systems, this requires that you</p>

IO Analysis Configuration Check Box	Prerequisites/Applicability
	<p>reconfigure your permissions for the prepare_debugfs.sh script located in the <code>bin</code> directory, or use root privileges.</p> <p>Not available for FreeBSD targets.</p>

[Analyze Platform Performance](#) Understand the platform-level metrics provided by the Input and Output analysis of Intel® VTune™ Profiler.

[Analyze DPDK Applications](#) Use the Input and Output analysis of Intel® VTune™ Profiler to profile DPDK applications and collect batching statistics for polling threads performing Rx and event dequeue operations.

[Analyze SPDK Applications](#) Use the Input and Output analysis of Intel® VTune™ Profiler to profile SPDK applications and estimate SPDK Effective Time and SPDK Latency, and identify under-utilized throughput of an SPDK device.

[Analyze Linux Kernel I/O](#) Use the Input and Output analysis of Intel® VTune™ Profiler to match user-level code to I/O operations executed by the hardware.

io Command Line Analysis

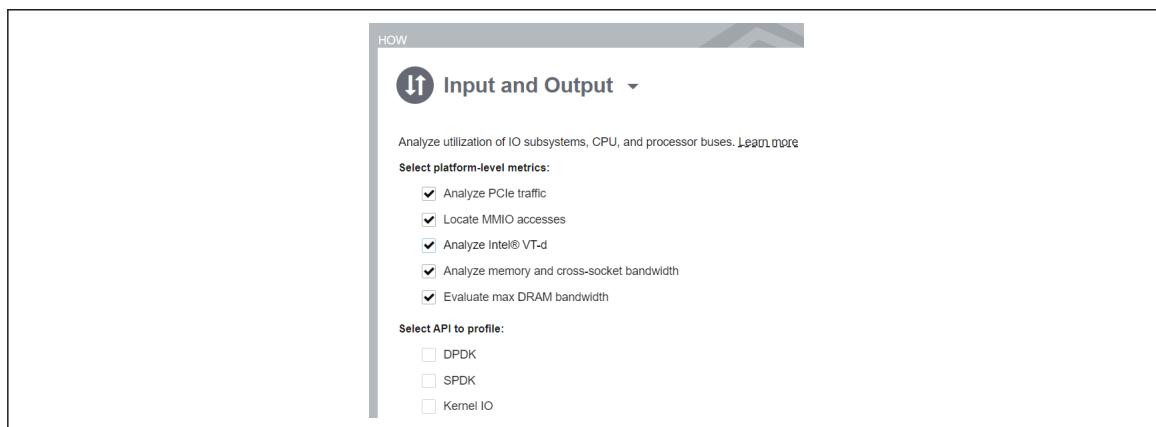
Analyze Platform Performance

Understand the platform-level metrics provided by the Input and Output analysis of Intel® VTune™ Profiler.

The Input and Output analysis provides platform-level metrics designed to:

- Analyze platform I/O traffic on per-I/O-device basis, whether the I/O device is an external PCIe device or an integrated accelerator.
- Analyze efficiency of [Intel® Data Direct I/O technology \(Intel® DDIO\) utilization](#).
- Analyze Intel® Virtualization Technology for Directed I/O (Intel® VT-d) utilization.
- Monitor DRAM and persistent memory bandwidth consumption.
- Identify I/O performance issues potentially caused by inefficient remote socket accesses.
- Identify sources of outbound I/O (MMIO) traffic.

To get this information, start the analysis with these options enabled:

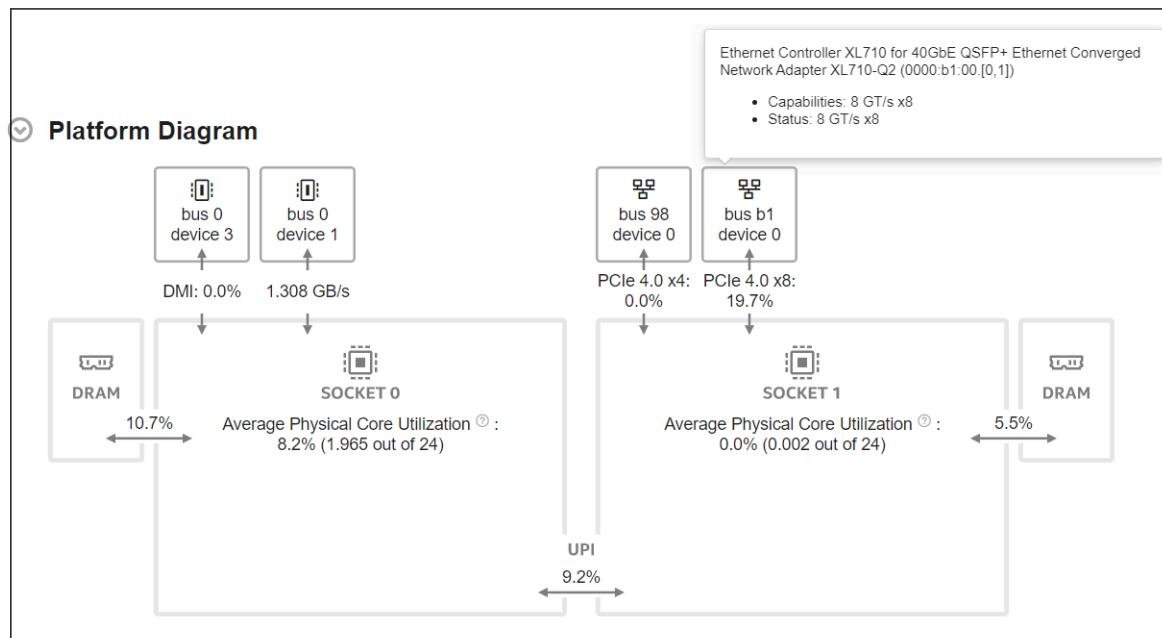


Analyze Topology and Hardware Resource Utilization

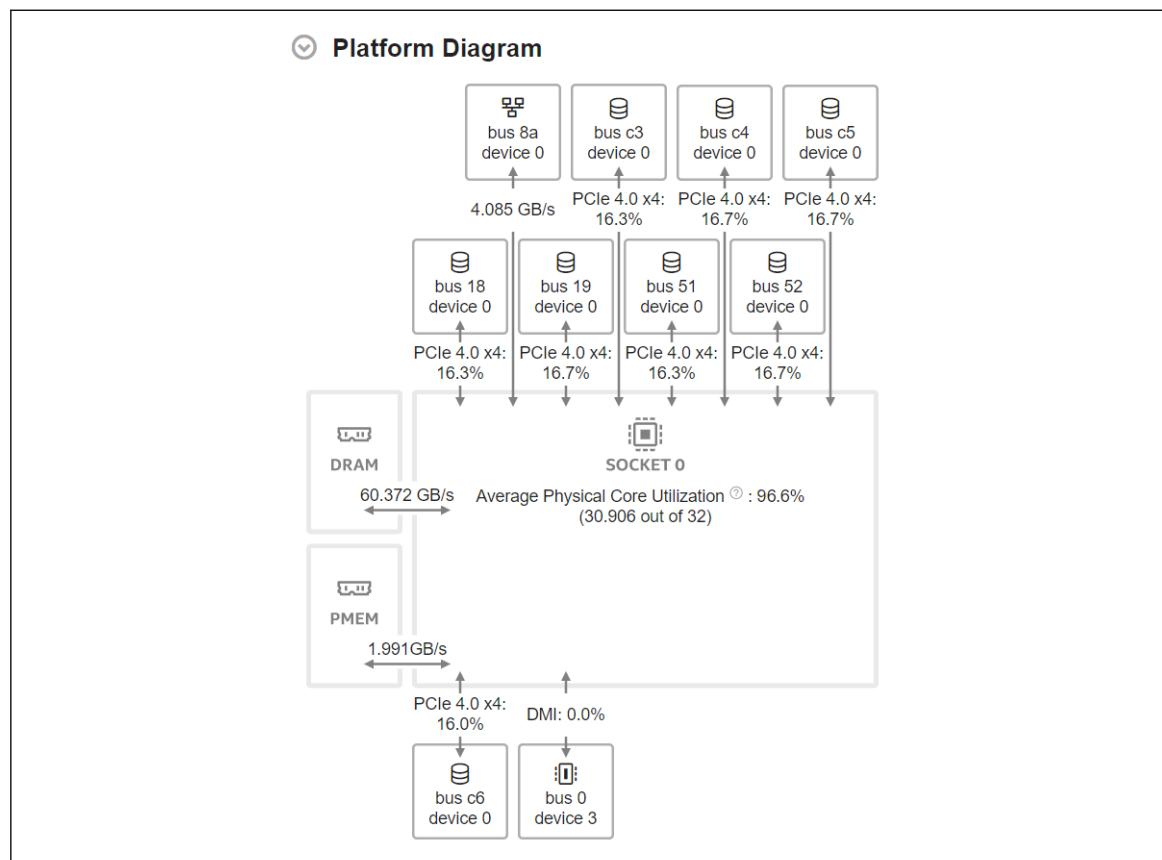
Once the data collection finishes, VTune Profiler opens the default **Summary** window.

Start your investigation with the **Platform Diagram** section of the **Summary** window. The **Platform Diagram** presents system topology and utilization metrics for I/O and Intel® UPI links, DRAM, persistent memory, and physical cores.

Here is an example of a **Platform Diagram** for a two-socket server with an active network interface card (NIC) on socket 1 and active Intel® QuickData Technology (CBDMA) on socket 0:



This is a **Platform Diagram** for a single-socket server with 8 active NVMe SSDs, network interface card, and persistent memory:

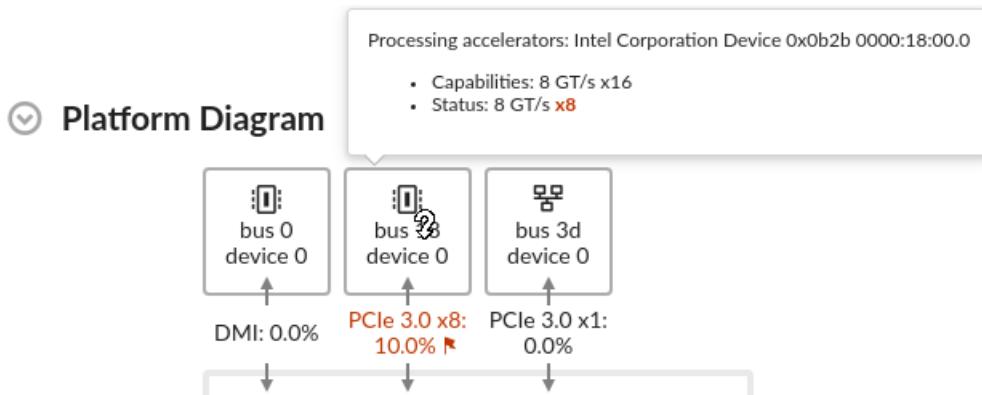


NOTE

You can observe the **Platform Diagram** in server platforms based on Intel® microarchitectures code named Skylake (with up to four sockets) and Sapphire Rapids.

I/O devices are shown with short names that indicate the PCIe bus and device numbers. Full device name, link capabilities, and status are shown in the device tooltip. Hover over the device image to see detailed device information.

The **Platform Diagram** highlights device status issues that may be a reason of limited throughput. A common issue is that the configured link speed/width does not match the maximum speed/width of the device.



When device capabilities are known and the maximum physical bandwidth can be calculated, the device link is attributed with the **Effective Link Utilization** metric that represents the ratio of bandwidth consumed on data transfers to the available physical bandwidth. This metric does not account for protocol overhead (TLP headers, DLLPs, physical encoding) and reflects link utilization in terms of payloads. Thus, it cannot reach 100%. However, this metric can give a clue on how far from saturation the link is. Maximum theoretical bandwidth is calculated for device link capabilities as shown in the device tooltip.

The Platform Diagram shows the **Average DRAM Utilization** when the **Evaluate max DRAM bandwidth** checkbox is selected in the analysis configuration. Otherwise, it shows the average DRAM bandwidth.

If the system is equipped with persistent memory, the Platform Diagram shows the **Average Persistent Memory Bandwidth**.

The **Average UPI Utilization** metric reveals UPI utilization in terms of transmit. The Platform Diagram shows a single cross-socket connection, regardless of how many UPI links connect a pair of packages. If there is more than one link, the maximum value is shown.

The **Average Physical Core Utilization** metric, displayed on top of each socket, indicates the utilization of physical cores by computations of the application being analyzed.

Once you examine topology and utilization, drill down into the details to investigate platform performance.

High Bandwidth Memory Data in Platform Diagram

For server platforms based on Intel® microarchitecture code named Sapphire Rapids, the **Platform Diagram** also includes information about High Bandwidth Memory (HBM). Use this information to distinguish from DRAM-specific utilization in the diagram.

For example, this diagram shows information about HBM mode utilization, where the system has no DRAM.

Platform Diagram



Here is an example of the **Platform Diagram** data in a system that has both HBM and DRAM.

Platform Diagram



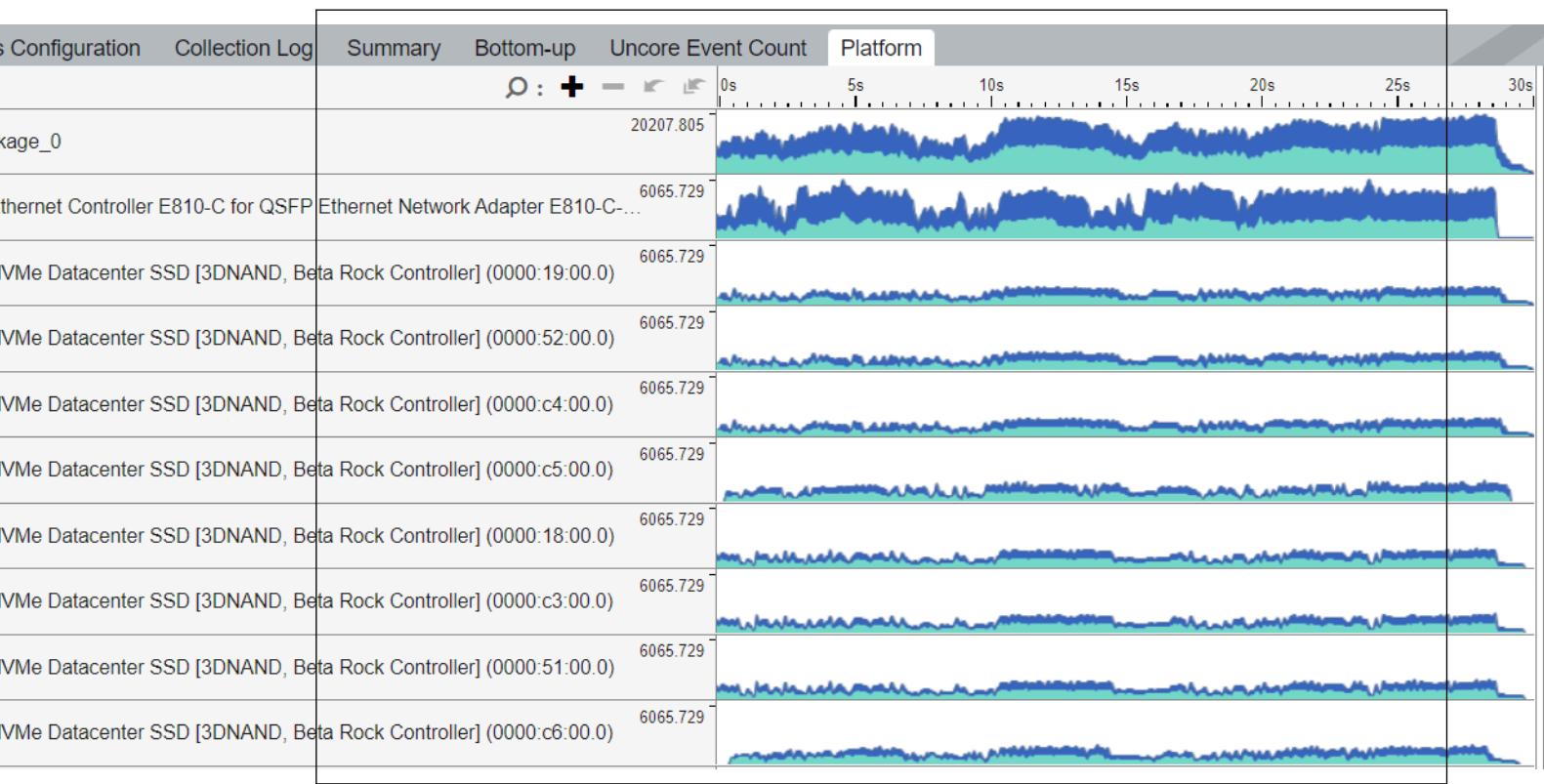
Analyze Platform I/O Bandwidth

To explore I/O traffic processing on the platform, start your investigation with the **PCIe Traffic Summary** section of the **Summary** window. These top-level metrics reflect the total Inbound and Outbound I/O traffic:

- **Inbound PCIe Bandwidth** is induced by I/O devices—whether external PCIe devices and/or integrated accelerators—that write to and read from the system memory. These reads and writes are processed by the platform through the Intel® Data Direct I/O (Intel® DDIO) feature.
 - **Inbound PCIe Read** — the I/O device reads from the platform memory.
 - **Inbound PCIe Write** — the I/O device writes to the platform memory.
- **Outbound PCIe Bandwidth** is induced by core transactions targeting the memory or registers of the I/O device. Typically, the core accesses the device memory through the Memory-Mapped I/O (MMIO) address space.
 - **Outbound PCIe Read** — the core reads from the registers of the device.
 - **Outbound PCIe Write** — the core writes to the registers of the device.

The granularity of **Inbound and Outbound PCIe Bandwidth** metrics depends on CPU model, collector used, and user privileges. For details, see the [Platform-Level Metrics table](#).

You can analyze the **Inbound and Outbound PCIe Bandwidth** over time on a per-device basis using the timeline in the **Bottom-up** or the **Platform** tabs:



Analyze Efficiency of Intel® Data Direct I/O Utilization

To understand whether your application utilizes Intel® DDIO efficiently, explore the second level metrics in the **PCIe Traffic Summary** section.

PCIe Traffic Summary

Inbound PCIe Read, MB/sec	7,035.360
L3 Hit, %	40.133
L3 Miss, %	59.867 ↘
Average Latency, ns	1,165.003 ↘
Inbound PCIe Write, MB/sec	7,567.562
L3 Hit, %	1.526
L3 Miss, %	98.474 ↘
CPU/IO Conflicts, %	0.000
Average Latency, ns	137.503
Outbound PCIe Read, MB/sec	0.002 ↘
Outbound PCIe Write, MB/sec	25.200

The **L3 Hit/Miss Ratios** for **Inbound** I/O requests reflect the proportions of requests made by I/O devices to the system memory that hit/miss the L3 cache. For a detailed explanation of Intel® DDIO utilization efficiency, see the [Effective Utilization of Intel® Data Direct I/O Technology](#) Cookbook recipe.

NOTE

L3 Hit/Miss metrics are available for Intel® Xeon® processors code named Haswell and newer.

The **Average Latency** metric of the **Inbound PCIe read/write groups** shows an average amount of time the platform spends on processing inbound read/write requests for a single cache line.

The **CPU/IO conflicts** ratio shows a portion of Inbound I/O write requests that experienced contention for a cache line between the IO controller and some other agent on the CPU, which can be a core or another IO controller. These conflicts are caused by the simultaneous access to the same cache line. Under certain conditions, such access may cause the IO controller to lose ownership of this cache line. This forces the IO controller to reacquire the ownership of this cache line. Such issues can occur in applications that use the polling communication model, resulting in suboptimal throughput and latency. To resolve this, consider tuning the **Snoop Response Hold Off** option of the Integrated IO configuration of UEFI/BIOS (option name may vary depending on platform manufacturer).

NOTE

Average Latency for inbound I/O reads/writes and **CPU/IO Conflicts** metrics are available on Intel® Xeon® processors code named Skylake and newer.

The granularity of **DDIO efficiency** metrics—second-level metrics for Inbound I/O bandwidth—depends on CPU model, collector used, and user privileges. For details, see the [Platform-Level Metrics table](#).

You can get a per-device breakdown for **Inbound and Outbound Traffic**, **Inbound request L3 hits and misses**, **Average latencies**, and **CPU/IO Conflicts** using the **Bottom-up** pane with the **Package / M2PCIe** or **Package / IO Unit** grouping:

g	Summary	Bottom-up	Uncore Event Count	Platform	
Unit	Inbound PCIe Read, MB/sec	Inbound PCIe Write, MB/sec			
		L3 Hit, %	L3 Miss, %	CPU/IO Co...	Average Lat...
Beta Rock Controller] (0000:c3:00.0)	7035.360	1.526	98.474	0.000	1.000
HP Ethernet Network Adapter E812	2761.143	2.680	97.320	0.000	1.000
Beta Rock Controller] (0000:18:00.0)	1519.143	0.136	99.864	0.000	2.000
Beta Rock Controller] (0000:51:00.0)	1378.313	0.075	99.925	0.000	2.000
MA) (0000:00:01.[0-7]), Platform C	1376.570	3.750	96.250	0.000	2.000
	0.191	74.372	25.628	0.000	2.000

[Analyze Utilization of Intel® Virtualization Technology for Directed I/O](#)

To understand how your workload utilizes the Intel® Virtualization Technology for Directed I/O (Intel VT-d), explore **Intel® VT-d** section of the result **Summary** tab. Intel VT-d enables addresses remapping for **Inbound I/O requests**.

NOTE

Intel VT-d metrics are available starting with server platforms based on Intel® microarchitecture code named Ice Lake.

Intel® VT-d

<input checked="" type="checkbox"/> Address Translation Rate, MT/s ⓘ:	49.437
IOTLB Hit, % ⓘ:	67.610
<input checked="" type="checkbox"/> IOTLB Miss, % ⓘ:	32.390 ↘
Average IOTLB Miss Penalty, ns ⓘ:	297.931
Memory Accesses Per IOTLB Miss ⓘ:	1.000

The top-level metric shows the average total **Address Translation Rate**.

The IOTLB (I/O Translation Lookaside Buffer) is an address translation cache in the remapping hardware unit that caches effective translations from virtual addresses, used by devices, to host physical addresses. IOTLB lookups happen on address translation requests. The **IOTLB Hit** and **IOTLB Miss** metrics reflect the ratios of address translation requests hitting and missing the IOTLB.

The next-level metrics for IOTLB misses are:

- **Average IOTLB Miss Penalty, ns** — average amount of time spent on handling an IOTLB miss. Includes looking up the context cache, intermediate page table caches and page table reads (page walks) on a miss, which turn into memory read requests.
- **Memory Accesses Per IOTLB Miss** — average number of memory read requests (page walks) per IOTLB miss.

The granularity of **Intel VT-d** metrics depends on CPU model, collector used, and user privileges. For details, see the [Platform-Level Metrics table](#). When prerequisites are met, **Intel VT-d** metrics can be viewed per sets of I/O devices—PCIe devices and/or integrated accelerators. Each set includes all devices handled by the single I/O controller, which commonly serves 16 PCIe lanes. Switch to the **Bottom-up** window and use **Package / IO Unit** grouping:

Analysis Configuration	Collection Log	Summary	Bottom-up	Uncore Event Count	Platform
Analysis Configuration: Package / IO Unit					
				Address Translation Rate, MT/s ▼	
			IOTLB Hit, %	IOTLB Miss, %	
				Average IOTLB Miss Penalty, ns	Memory Accesses Per IOTLB Miss
Package_1			63.872	314.100	
Ethernet Controller XL710 for 40GbE QSFP+ Ethernet Controller			63.872	314.098	
Ethernet Controller 10G X550T Ethernet Converged Network Adapter			57.118	1130.768	
VMWare Datacenter SSD [3DNAND, Beta Rock Controller] N			0.000	0.000	
Package_0			77.046	233.675	
Intel® QuickData Technology (CBDMA) (0000:00:01.[0-7])			77.046	233.675	
VMWare Datacenter SSD [3DNAND, Beta Rock Controller] N			0.000	0.000	

Analyze MMIO Access

Outbound I/O traffic visible in the **PCIe Traffic Summary** section of the **Summary** tab is caused by cores writing to and reading from memory/registers of I/O devices.

Typically, cores access I/O device memory through the Memory-Mapped I/O (MMIO) address space. Each load or store operation targeting the MMIO address space that an I/O device is mapped to causes outbound I/O read or write transactions respectively. When performed through the usual load and store instructions, such memory accesses are quite expensive, since they are affected by the I/O device access latency. Therefore, such accesses should be minimized to achieve high performance. The latest Intel architectures incorporate [direct store instructions \(MOVDIR*\)](#) which may enable high rate for MMIO writes, usually used for job submission or "doorbell rings".

Enable the **Locate MMIO accesses** option during analysis configuration to detect the sources of outbound traffic. Use the **MMIO Access** section to locate functions performing **MMIO Reads** and **MMIO Writes** that target specific PCIe devices.

Traffic Summary

Outbound PCIe Read, MB/sec^⑦: 2,135.987
 Outbound PCIe Write, MB/sec^⑧: 1,845.772
 Inbound PCIe Read, MB/sec^⑨: 0.005 ↗
 Inbound PCIe Write, MB/sec^⑩: 4.958

MMIO Access

This section lists functions accessing PCIe devices through Memory-Mapped I/O (MMIO) address space during collection run. Reads/writes from/to where PCIe device is mapped lead to Outbound PCIe Read/Write transactions respectively. MMIO reads are long-latency loads that are usually configuration. MMIO writes are typically used for doorbells, i.e. updates of tail/head pointers of ring buffers used for core/device communication. Throughput explore and limit MMIO accesses on the hot path by avoiding MMIO reads and minimizing MMIO writes.

Memory-Mapped PCIe Device / Source Function	Source File	MMIO Reads	MMIO Writes
Ethernet Controller XL710 for 40GbE QSFP+ Ethernet Converged Network Adapter XL710-Q2 0000:af:00.1	rte_io.h	11,099 ↗	8
rte_read32_relaxed	rte_io.h	0	8
rte_read32_relaxed	rte_io.h	11,099 ↗	8

Use the **Bottom-up** pane to locate sources of memory-mapped PCIe device accesses. Explore the call stacks and drill down to source and assembly view:

Function / Memory-Mapped PCIe Device / Call Stack	MMIO Reads	MMIO Writes	Module	Source File
rte_read32_relaxed	2,018	0	testpmd	rte_io.h
Ethernet Controller XL710 for 40GbE QSFP+ Ethernet Converged Network Adapter XL710-Q2 0000:af:00.1	2,018	0	testpmd	rte_io.h
rte_read32 ← i40e_read_addr ← i40e_stat_update_48 ← i40e_update_vsi_stats	2,018	0	testpmd	rte_io.h

Double click on the function name to drive into source code or assembly view to locate the code responsible for MMIO reads and writes at source line level:

Source	Assembly	MMIO Reads: Total	MMIO Reads: Self	MMIO Writes: Total	MMIO Writes: Self
S... ▲					
248 static __rte_always_inline uint32_t					
249 rte_read32_relaxed(const volatile void *addr)					
250 {					
251 > return *(const volatile uint32_t *)addr;		22.2%	4,036	0.0%	0
252 }					

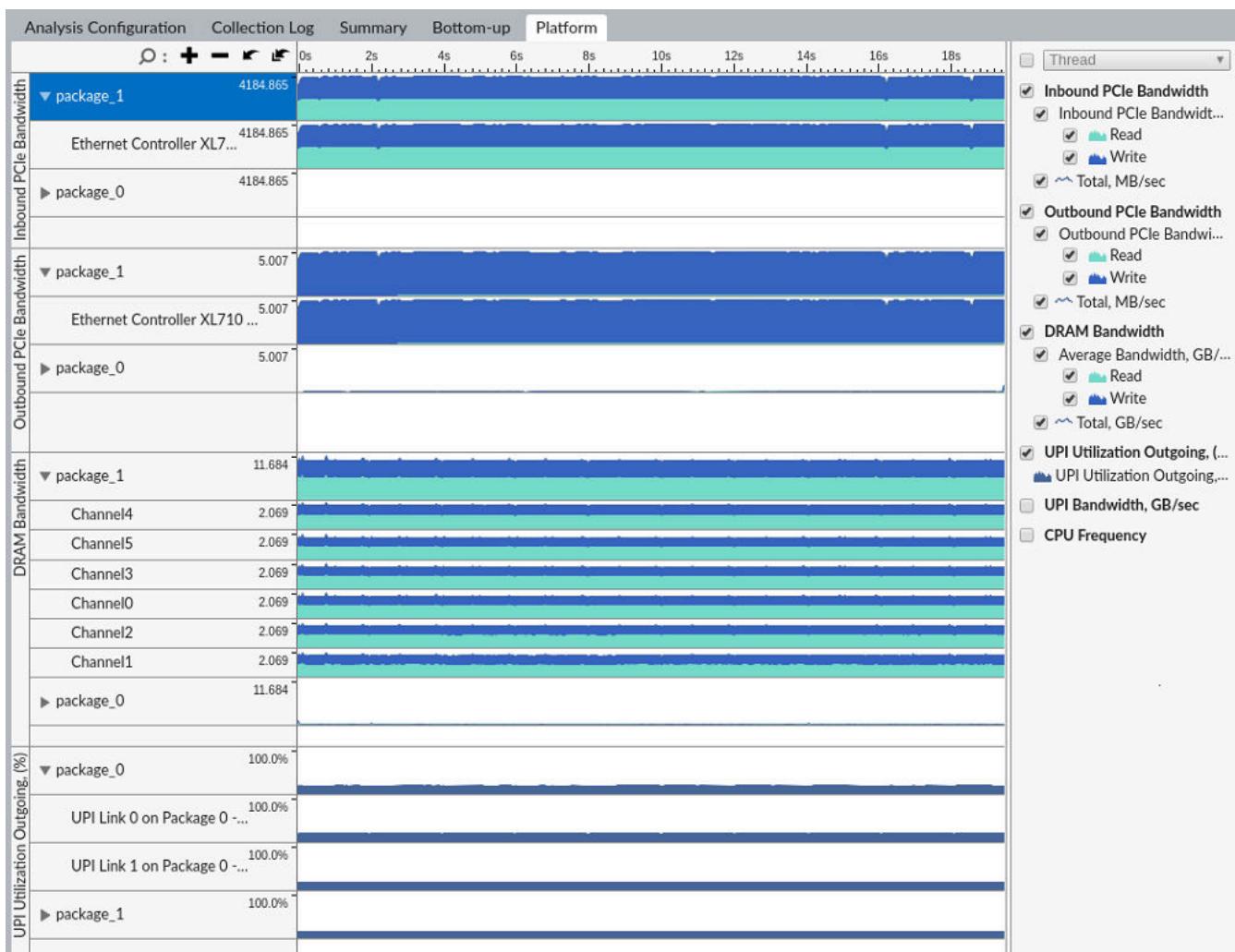
NOTE

MMIO access data is collected when the **Locate MMIO accesses** check box is selected. However, there are some limitations:

- This feature is only available starting with server platforms based on the Intel® microarchitecture code name Skylake.
- Only **Attach to Process** and **Launch Application** collection modes are supported. When running in the **Profile System** mode, this option only reveals functions performing reads from uncacheable memory.

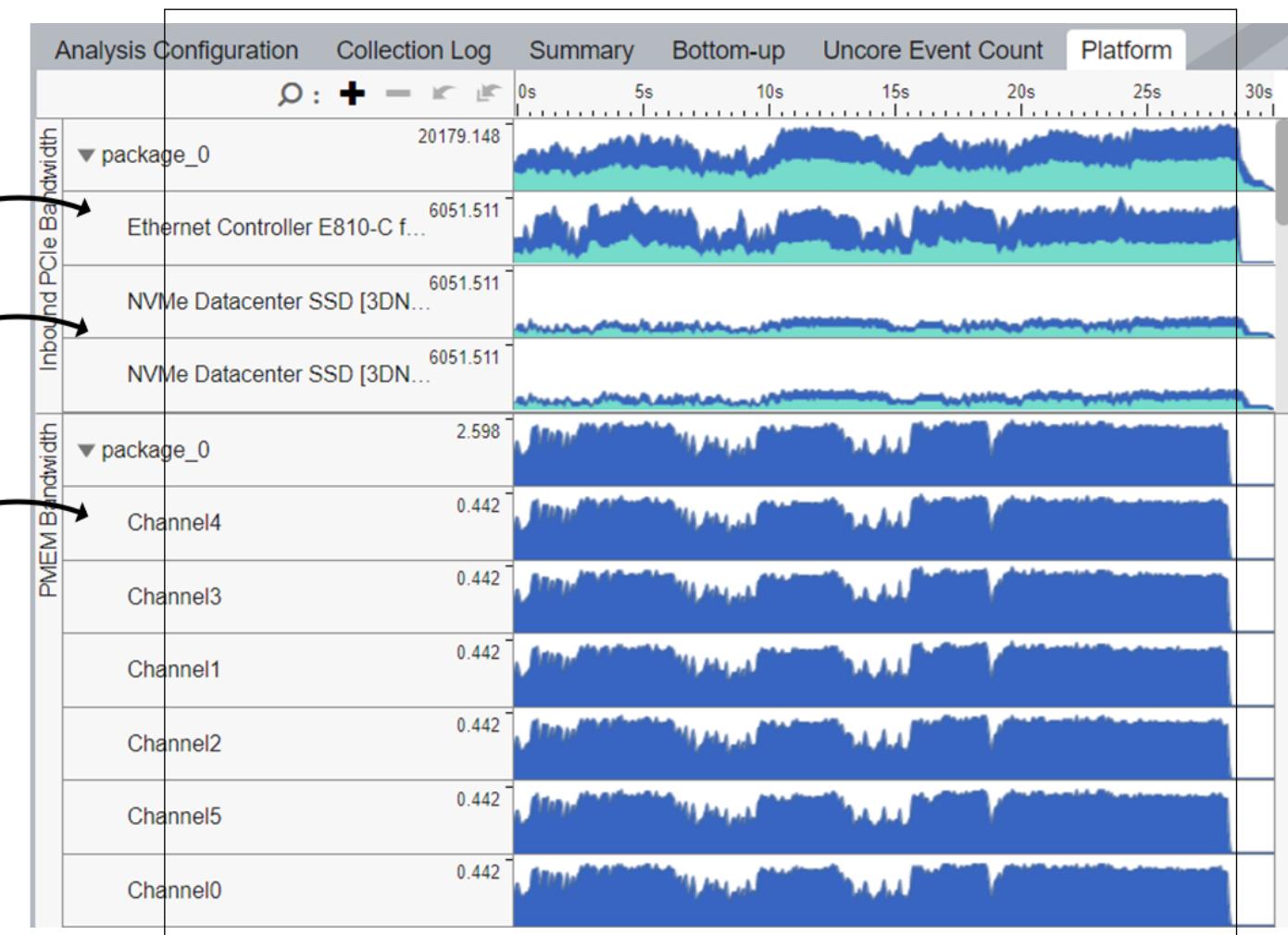
Analyze Memory, Persistent Memory, and Cross-Socket Bandwidth

Use the **Platform** tab to correlate I/O traffic with DRAM, PMEM (persistent memory) and cross-socket interconnect bandwidth consumption:



VTune Profiler provides per-channel breakdown for DRAM and PMEM bandwidth:

network traffic
storage traffic
persistent memory traffic



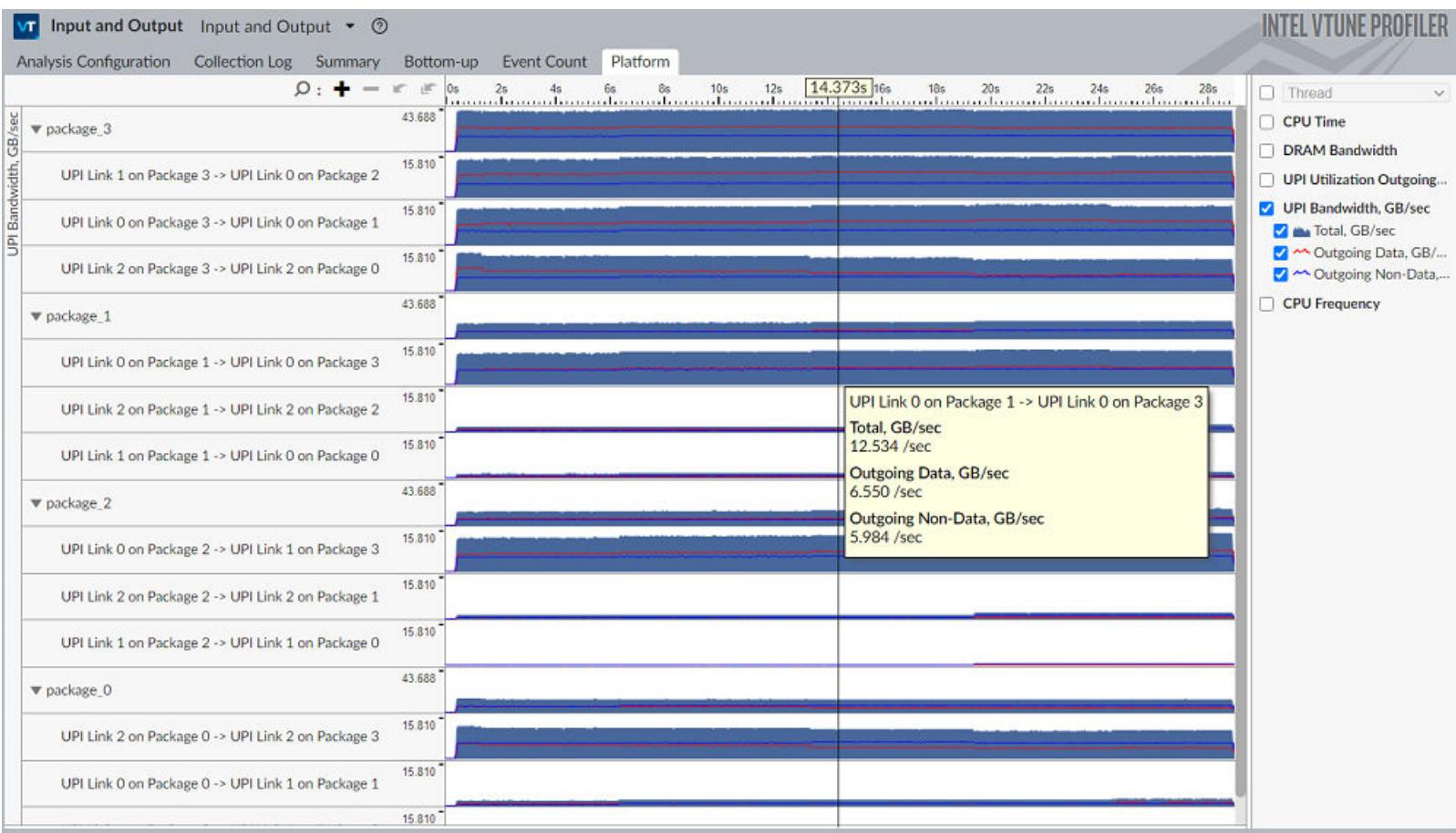
Two metrics are available for Intel® UPI traffic:

- **UPI Utilization Outgoing** – ratio metric that shows UPI utilization in terms of transmit.
- **UPI Bandwidth** – shows detailed bandwidth information with breakdown by data/non-data.

You can get a breakdown of UPI metrics by UPI links. See the specifications of your processor to determine the number of UPI links that are enabled on each socket of your processor.

UPI link names reveal the topology of your system by showing which sockets and UPI controllers they are connected to.

Below is an example of a result collected on a four-socket server powered by Intel® processors with microarchitecture code named Skylake. The data reveals significant UPI traffic imbalance with bandwidth being much higher on links connected to socket 3:



[Cookbook: PCIe Traffic in DPDK Apps](#)

[Cookbook: Effective Utilization of Intel® Data Direct I/O Technology](#)

Analyze DPDK Applications

Use the *Input and Output* analysis of Intel® VTune™ Profiler to profile DPDK applications and collect batching statistics for polling threads performing Rx and event dequeue operations.

NOTE

To profile a DPDK application using VTune Profiler, make sure DPDK is built with VTune Profiler options enabled. See the [DPDK guide](#) for more information.

When profiling DPDK as [FD.io VPP](#) plugin, modify `DPDK_MESON_ARGS` variable in `build/external/packages/dpdk.mk` with the same flags as described in [Profiling with VTune](#) section.

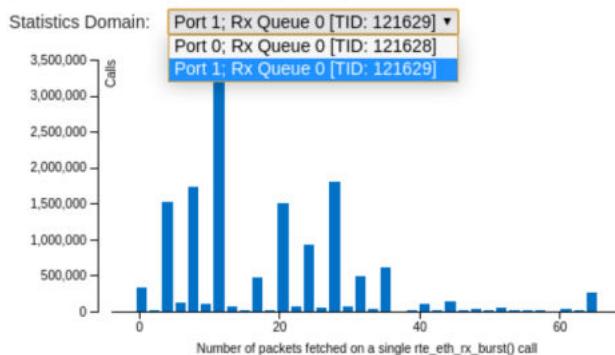
DPDK statistics collection is not supported for FreeBSD* targets and is not available in Profile System mode.

Analyze Rx Batch Statistics

Start with the **Summary** tab and explore the **DPDK Rx Batch Statistics** histogram to get summary statistics for packet batches retrieving and to get a full characterization of core utilization on Rx. The histogram is available for each polling thread associated with a specific Rx queue:

DPDK Rx Batch Statistics

In data plane applications, where fast packet processing is required, the DPDK polls a certain port for incoming packets in an infinite loop. To understand efficiency of the polling thread utilization, explore the batch statistics of fetching packets with the `rte_eth_rx_burst()` batch operation.



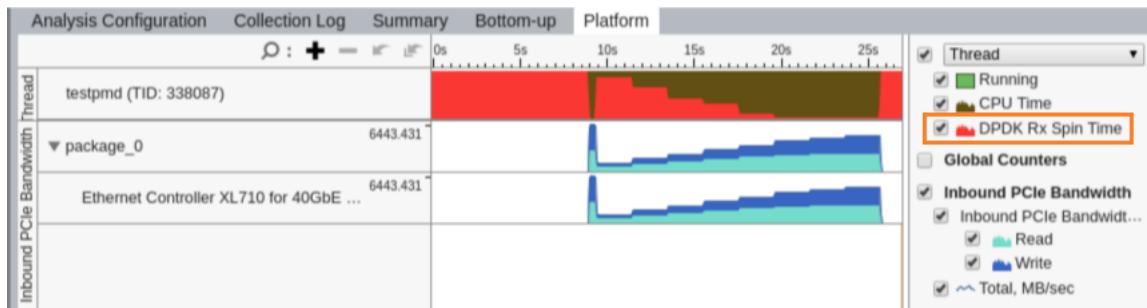
Analyze Rx Spin Time

While the polling loop is running on a core, the **CPU Time** metric for this core is always close to 100%, regardless of how many loop cycles DPDK spends in an idle state. Therefore, the **CPU Time** metric cannot be used to reliably identify how the core is utilized on packet retrieval. For this polling model, a better utilization indicator might be the **Rx Spin Time** value, which is the ratio of wasted polling loop cycles. Wasted cycles are loop iterations during which DPDK does not receive any packets.

The **DPDK Rx Spin Time** metric shows the ratio of polling cycles fetching no packets, or the number `rte_eth_rx_burst()` calls that returned zero packets, to the total number of polling loop cycles:

$$\text{DPDK Rx Spin Time} = \frac{\text{Num of calls that return 0 packets}}{\text{Total num of calls}}$$

Use the **Platform** tab to explore the **DPDK Rx Spin Time** metric on the timeline at per-thread basis:



To learn more about core utilization in DPDK applications, see the corresponding [cookbook recipe](#).

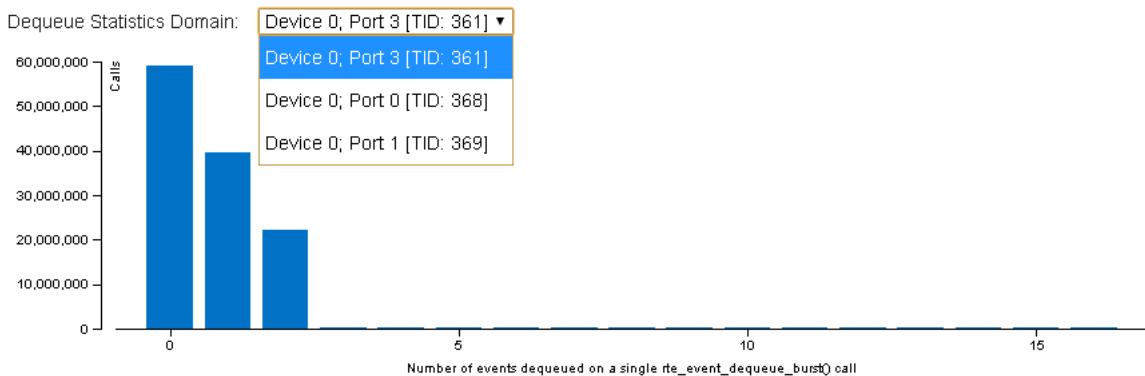
Analyze DPDK Event Dequeue Statistics

Use the Input and Output analysis to collect DPDK eventdev dequeue batch statistics and analyze eventdev pipeline configuration efficiency.

Start your investigation with the **DPDK Events Dequeue Statistics** section of the **Summary** tab:

DPDK Events Dequeue Statistics

Explore the statistics of the events dequeued by rte_event_dequeue_burst() function to understand the efficiency of the eventdev pipeline.



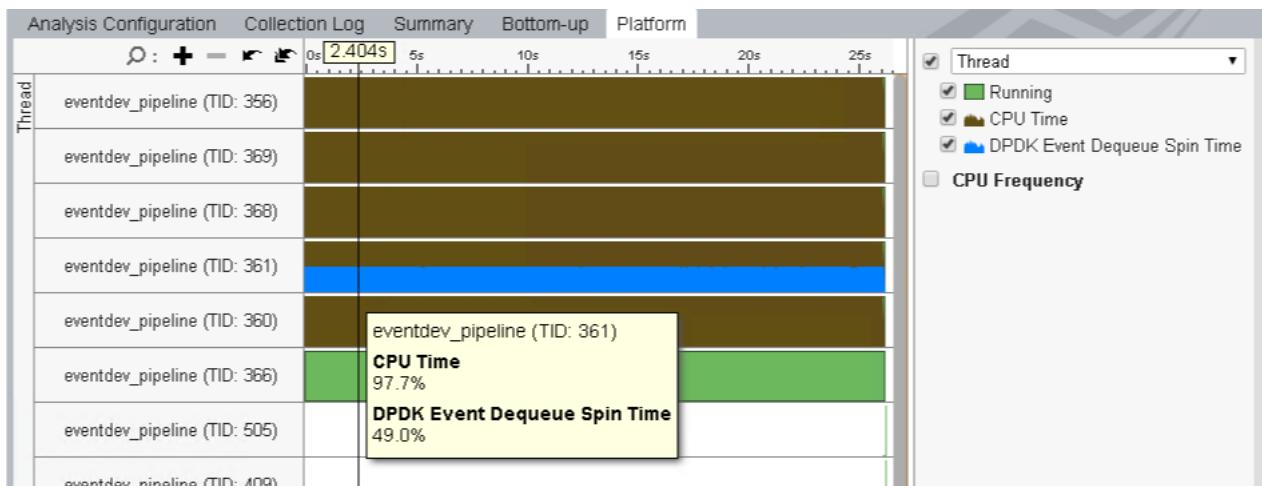
This histogram shows batching statistics for packet (event) dequeue operation from the DPDK eventdev library. It provides statistics for each eventdev port, representing each worker thread that polls the event device. Explore the histogram to identify inhomogenous load distribution, oversubscribed, or underutilized worker threads.

Analyze DPDK Event Dequeue Spin Time

The **DPDK Event Dequeue Spin Time** metric represents the ratio of empty dequeue cycles, or the number of rte_event_dequeue_burst() calls that have returned zero events, with respect to the total number of dequeue calls:

$$\text{DPDK Event Dequeue Spin Time} = \frac{\text{Num of calls that return 0 packets}}{\text{Total num of dequeue calls}}$$

Navigate to the **Platform** tab to explore the **DPDK Event Dequeue Spin Time** metric on the timeline. Per-worker dequeue statistics reveal details about load balancing, which enables you to analyze pipeline configuration efficiency and to identify underlying pipeline bottlenecks.



To learn more about the DPDK eventdev pipeline, see the [DPDK Event Device Profiling](#) Cookbook recipe.

[Cookbook: PCIe Traffic in DPDK Apps](#)

[Cookbook: Core Utilization in DPDK Apps](#)

[Cookbook: DPDK Event Device Profiling](#)

Analyze SPDK Applications

Use the Input and Output analysis of Intel® VTune™ Profiler to profile SPDK applications and estimate SPDK Effective Time and SPDK Latency, and identify under-utilized throughput of an SPDK device.

NOTE

To enable VTune Profiler capabilities, make sure SPDK is built using the --with-vtune-install-dir> advanced build option.

When profiling in **Attach to Process** mode, make sure to [set up the environment variables](#) before launching the application.

Not available in **Profile System** mode.

SPDK Effective Time

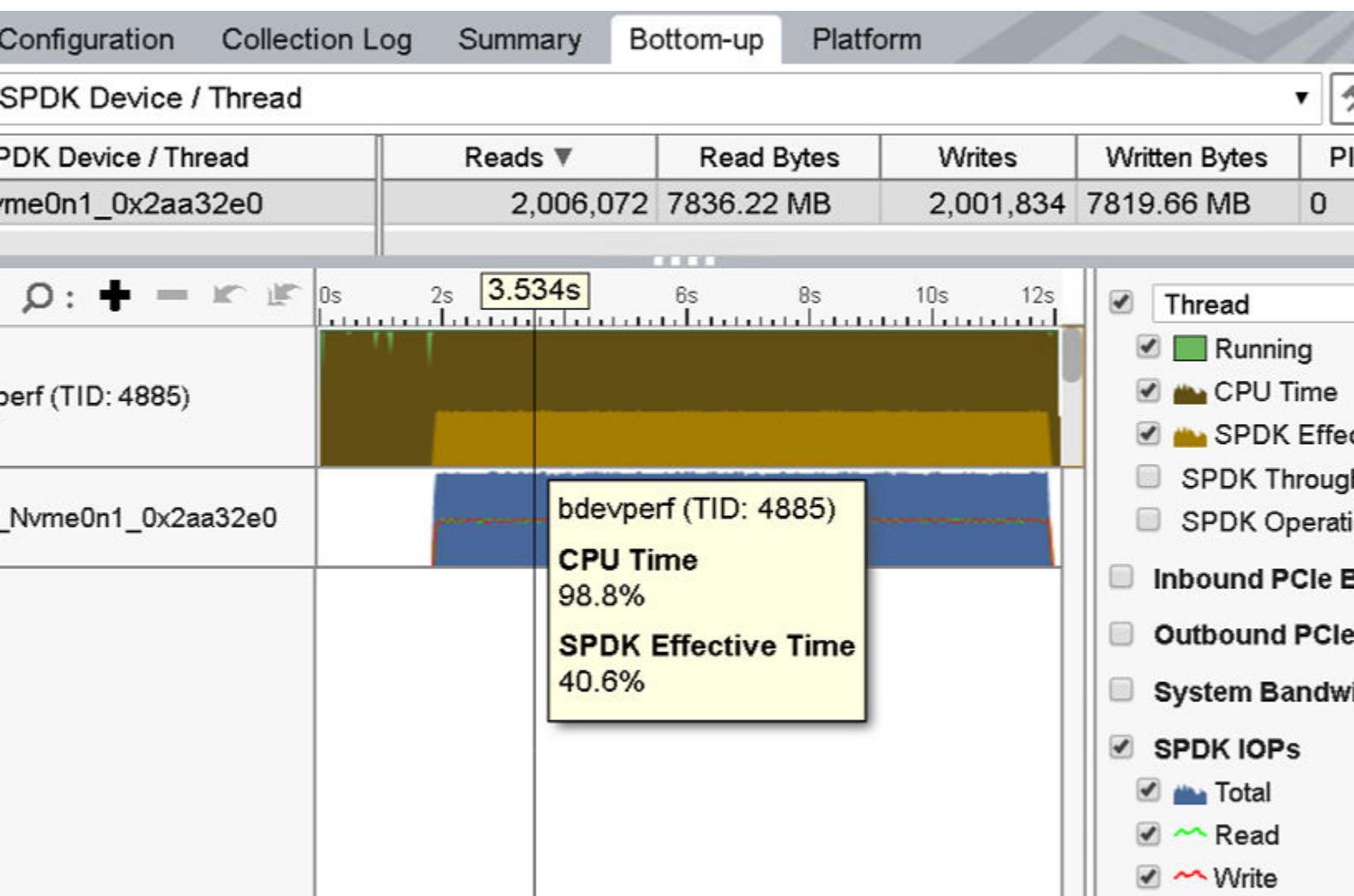
Start your investigation with the **Summary** window that displays overall SPDK performance statistics, grouped by executed operation types.

SPDK Info	
④ Reads:	2,006,072
bdev_Nvme0n1_0x2aa32e0:	2,006,072
④ Read Bytes:	7836.22 MB
④ Writes:	2,001,834
④ Written Bytes:	7819.66 MB
④ SPDK Effective Time^①:	4.016s

The **SPDK Effective Time** metric shows the amount of time the application spent performing any activity, excluding polling for I/O operation completion:

SPDK Effective Time = CPU Elapsed Time – IO Wait Time

To analyze this metric on a per-thread basis, use the **Bottom-up** or **Platform** tabs:



Analyze SPDK Throughput

Use the **SPDK Throughput Utilization** histogram of the **Summary** tab to understand utilization of specific storage devices managed by SPDK:

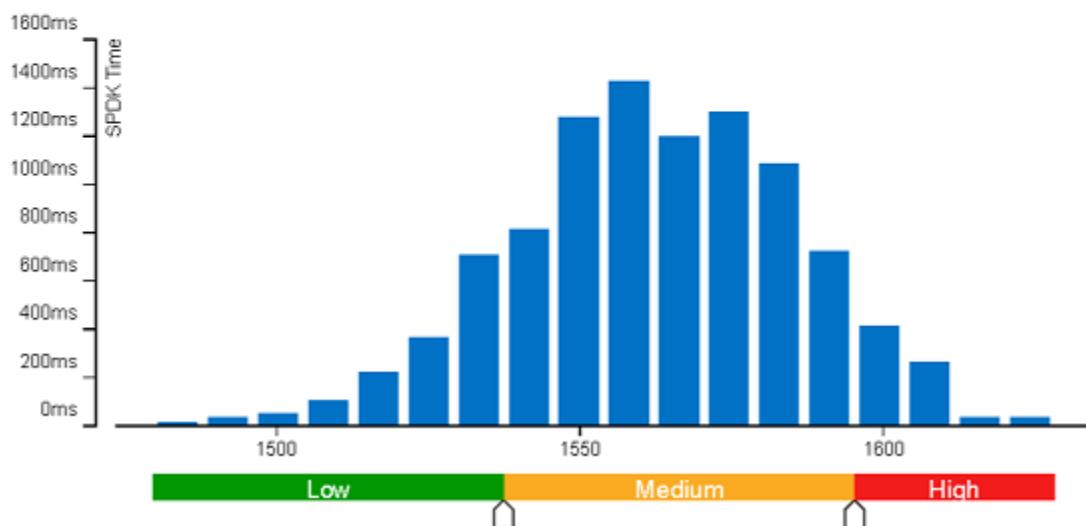
SPDK Throughput

Analyze information on the SPDK throughput utilization per device.

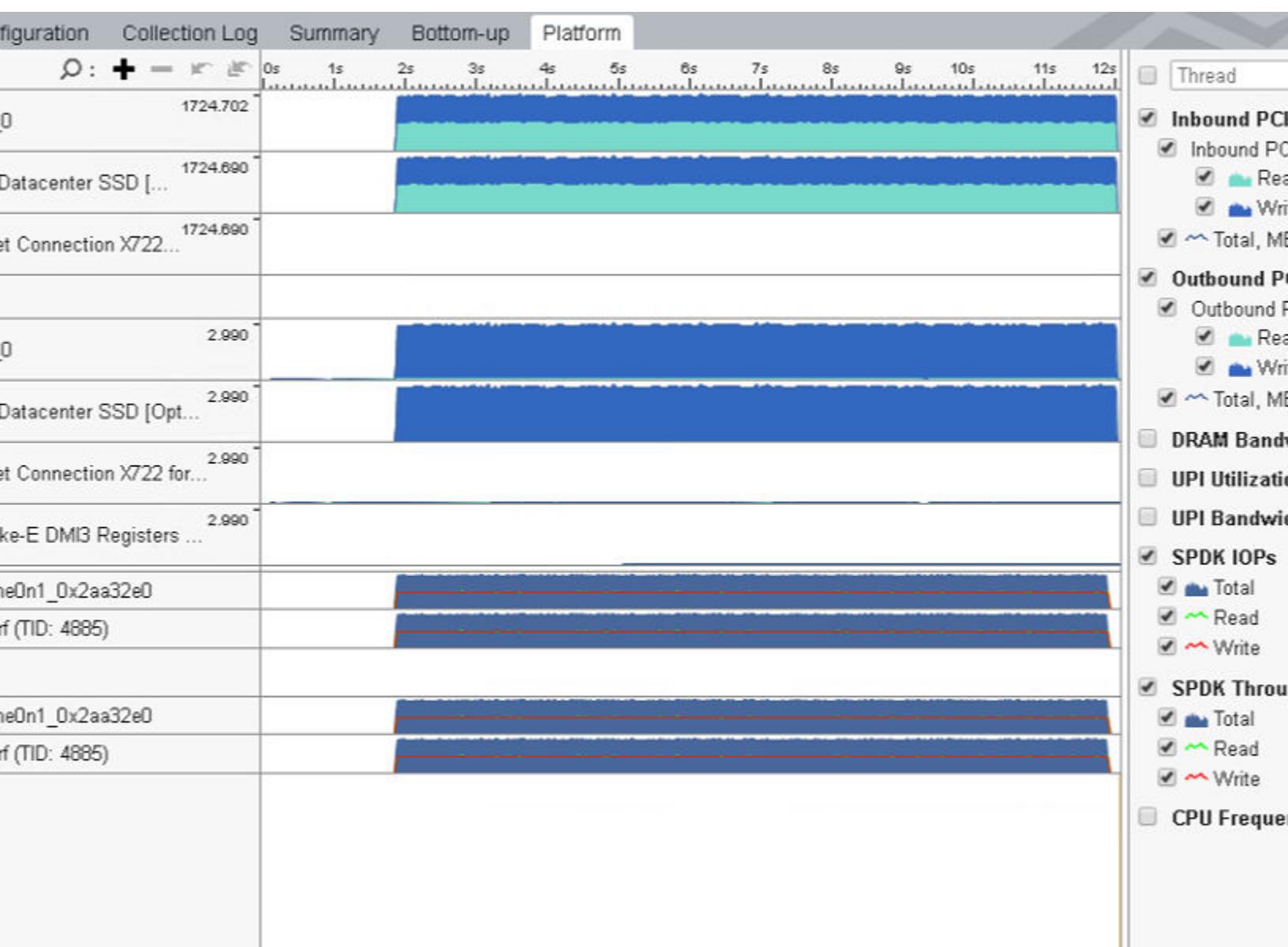
SPDK Device: bdev_Nvme0n1_0x2aa32e0 ▾

SPDK Throughput Histogram

Explore an over-time distribution of the throughput utilization by IO operations for the selected SPDK device.



You can use the timeline in the **Platform** tab to correlate areas of SPDK throughput utilization with SPDK I/O operations and to get a breakdown of PCIe traffic per physical device:



Analyze SPDK Latency

Explore the **SPDK Latency** histogram of the **Summary** tab to understand how much time the SPDK application spends experiencing certain I/O operation latency on a per-device basis.

$$\text{Latency} = \frac{\text{Sample Duration}}{\text{Total Number of IOPs in Sample}}$$

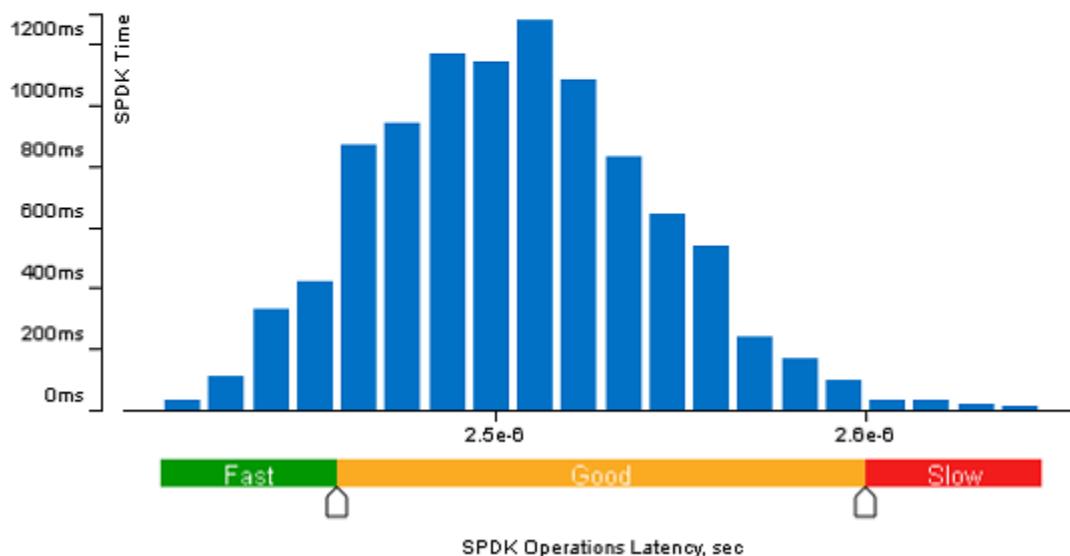
SPDK Latency

Analyze information on the SPDK operations latency per device.

SPDK Device: bdev_Nvme0n1_0x2aa32e0 ▾

SPDK Latency Histogram

Explore the distribution of the IO operations latency over time for the selected SPDK device.



- **Active** state — the CPU core is executing a thread
- **I/O Wait** — the CPU core is idle, but there is a thread that could potentially be executed on this core that is blocked by disk access.

All I/O metrics collected by VTune Profiler, such as **I/O Wait Time**, **I/O Waits**, and **I/O Queue Depth**, are collected in a system-wide mode and are not target-specific.

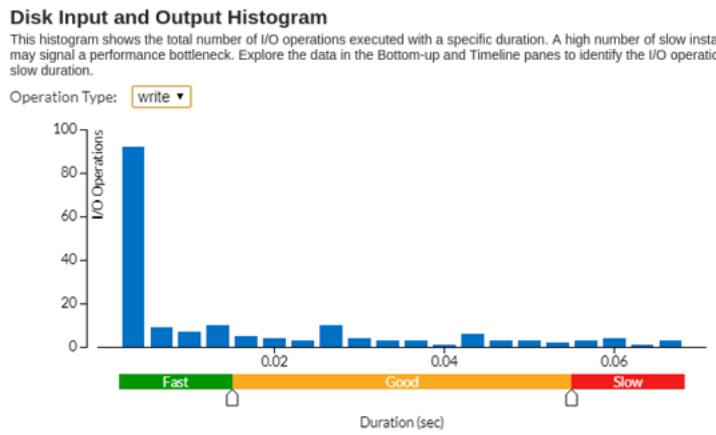
Analyze I/O Wait Time

To analyze **I/O Wait Time**, start with the **Summary** window. This window provides a quick overview of the target system performance and introduces the **I/O Wait Time** metric that helps you identify whether your application is I/O-bound:

Elapsed Time	: 7.222s
I/O Wait Time	: 0.183s
CPU Time	: 3.958s
Instructions Retired	: 8,459,636,000
CPI Rate	: 0.788
Total Thread Count	: 245
Paused Time	: 0s

The **I/O Wait Time** metric represents a portion of time during which the threads are in I/O wait state while the system has cores in idle state. In this case, the number of threads is not greater than the number of idling cores. This aggregated **I/O Wait Time** metric is an integral function of the **I/O Wait** metric that is available in the **Timeline** pane of the **Bottom-up** window.

To estimate how quickly storage requests are served by the kernel sub-system, see the **Disk Input and Output Histogram**. Use the **Operation Type** drop-down menu to select the type of I/O operation you are interested in. For example, for I/O writes, 2-4 storage requests executed within 0.06 seconds or more are classified as slow by VTune Profiler:

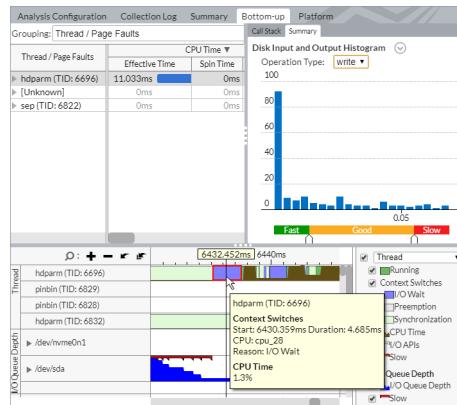


To explore this type of I/O request in greater detail, switch to the **Bottom-up** window.

Analyze Slow I/O Requests

In the **Bottom-up** window, select an area of interest on the timeline, then use the **Zoom In and Filter by Selection** context menu option. The **Summary** histogram is updated to show the data for the selected time range.

For example, in this case, there were 2-4 slow write requests executed during the 6th second of application execution:



By zooming in on an area of interest, you can get a closer look at different metrics and understand the reason behind high I/O wait time.

VTune Profiler collects the **I/O Wait** type of context switches caused by I/O accesses from the thread, and provides a system-wide **I/O Wait** metric in the **CPU Activity** area. Use this data to identify imbalance between I/O and compute operations.

System-wide **I/O Wait** shows the time during which the system cores were idle, but there were threads in a context switch due to I/O access. Use this metric to estimate the dependency of performance on the storage medium.

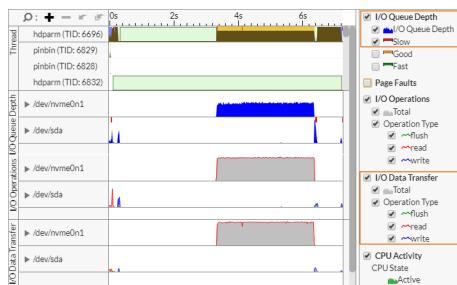
For example, an **I/O Wait** value of 100% means that all cores of the system are idle, but there are threads blocked by I/O requests. To solve this issue, change the logic of the application to run compute threads in parallel with I/O tasks. Alternatively, consider using faster storage.

An **I/O Wait** value of 0% could mean one of the following:

- Regardless of the number of threads blocked on storage access, all CPU cores are actively executing application code.
- No threads are blocked on storage access.

Explore the **I/O Queue Depth** area to see the number of storage requests submitted to the storage device. Spikes correspond to the maximum number of requests. Zero-value gaps on the **I/O Queue Depth** chart correspond to points in application run when storage was not utilized at all.

To identify the exact points in time when slow I/O packets were scheduled for execution, enable the **Slow** markers for the **I/O Queue Depth** metric:

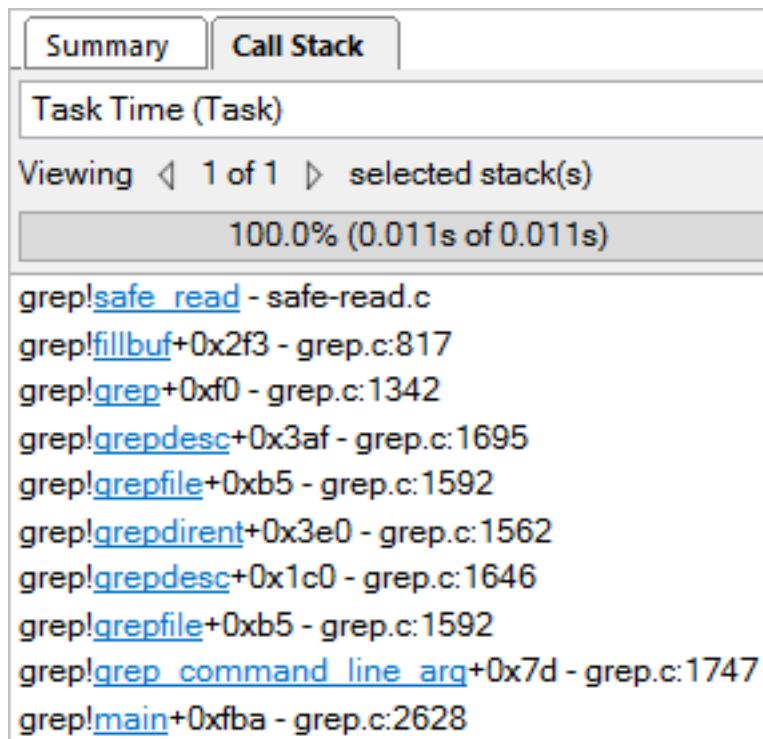


To identify points of high bandwidth, analyze the **I/O Data Transfer** area that shows the number of bytes read from or written to the storage device.

Analyze Call Stack for I/O Functions

VTune Profiler instruments all user-space I/O functions. This enables you to correlate slow I/O requests with instrumented user-space activities. You can do that by examining the full call stack that points to the exact API invocation.

To view a **Task Time** call stack for a particular I/O call, select the required **I/O API** marker on the timeline and explore the stack in the **Call Stack** pane:



Accelerators Analysis Group

The **Accelerators** group introduces analysis types that monitor CPU, GPU, FPGA, and NPU usage.

- Use the [GPU Offload](#) analysis to profile applications that use a Graphics Processing Unit (GPU) for rendering, video processing, and computations. This analysis type helps you identify whether your application is CPU or GPU bound.
- For GPU-bound applications, use the [GPU Compute/Media Hotspots](#) (preview) analysis type to see the GPU kernel execution per code line. Identify performance issues caused by memory latency or inefficient kernel algorithms.
- Use the [CPU/FPGA Interaction](#) analysis to explore FPGA utilization for each FPGA accelerator and identify the most time-consuming FPGA computing tasks.
- Use the [NPU Exploration analysis](#) (preview) to profile and optimize artificial intelligence(AI) workloads running on Intel architectures.

NOTE

A **PREVIEW FEATURE** may or may not appear in a future production release. While a preview feature is available for your use, feedback about its usefulness will determine its availability in future releases. Data collected with a preview feature is not guaranteed to be compatible with future releases.

Prerequisites:

- [Install the sampling driver](#) for hardware event-based sampling collection types. For Linux* and Android* targets, if the sampling driver is not installed, VTune Profiler can work on Perf* ([driverless collection](#)).

- To enable system-wide and uncore event collection, use root or sudo to set `/proc/sys/kernel/perf_event_paranoid` to 0.

```
$ echo 0>/proc/sys/kernel/perf_event_paranoid
```

- To enable the collection of Ftrace events on a system where the Linux Ftrace subsystem is only accessible for the root user, change system permissions using the `prepare-debugfs-and-gpu-environment.sh` script with root privileges.

See Also

[Optimize applications for Intel® GPUs with Intel® VTune Profiler](#)

[GPU Architecture Terminology for Intel® Xe Graphics](#)

[Optimize Your GPU Application with Intel oneAPI Base Toolkit](#)

[Offload Modeling Perspective](#) in Intel® Advisor to estimate GPU offload overhead
in Intel® Advisor to estimate GPU offload overhead

GPU Offload Analysis

Explore code execution on various CPU and GPU cores on your platform, correlate CPU and GPU activity, and identify whether your application is GPU or CPU bound.

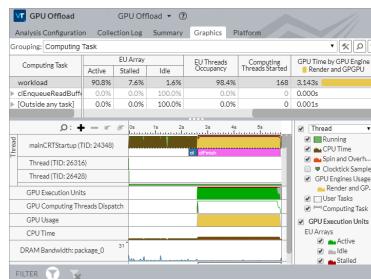
Run the GPU Offload analysis for applications that use a Graphics Processing Unit (GPU) for rendering, video processing, and computations with explicit support of SYCL*, Intel® Media SDK and OpenCL™ software technology.

The tool infrastructure automatically aligns clocks across all cores in the entire system so that you can analyze some CPU-based workloads together with GPU-based workloads within a unified time domain.

This analysis enables you to:

- Identify how effectively your application uses SYCL or OpenCL kernels and explore them further with [GPU Compute/Media Hotspots](#) analysis
- Analyze execution of Intel Media SDK tasks over time (for Linux targets only)
- Explore GPU usage and analyze a software queue for GPU engines at each moment of time

For the GPU Offload analysis, Intel® VTune™ Profiler instruments your code executing both on CPU and GPU. Depending on your configuration settings, VTune Profiler provides performance metrics that give you an insight into the efficiency of GPU hardware use. You can also identify next steps in your analysis.



Aspects of the GPU Offload Analysis

By default, the GPU Offload analysis enables the **GPU Utilization** option to explore GPU busyness over time and understand whether your application is CPU or GPU bound. Consequently, if you explore the [Timeline view](#) in the **Graphics** window, you may observe:

- The GPU is busy most of the time
- There are small idle gaps between busy intervals
- The GPU software queue is rarely decreased to zero

If these behaviors exist, you can conclude that your application is GPU bound.

If the gaps between busy intervals are big and the CPU is busy during these gaps, your application is CPU bound.

But such obvious situations are rare and you need a detailed analysis to understand all dependencies. For example, an application may be mistakenly considered GPU bound when the usage of GPU engines is serialized (for example, when GPU engines responsible for video processing and for rendering are loaded in turns). In this case, an ineffective scheduling on the GPU results from the application code running on the CPU.

Configure the Analysis

On Windows systems, to monitor general GPU usage over time, run VTune Profiler as an Administrator.

- [Set up your system for GPU analysis.](#)
- For SYCL applications: make sure to compile your code with the `-gline-tables-only` and `-fdebug-info-for-profiling` Intel oneAPI DPC++ Compiler options.
- [Create a project](#) and specify an analysis system and target.

Run the Analysis

1. Open the **Configure Analysis** window. Click the



button on the welcome screen (standalone version) or the



Configure Analysis(Visual Studio IDE) toolbar button.

2. Open the Analysis Tree from the **HOW** pane and select **GPU Offload** analysis from the **Accelerators** group.

The GPU Offload analysis is pre-configured to collect GPU usage data and collect Processor Graphics hardware events (Global Memory Accesses preset).

NOTE

If you have multiple Intel GPUs connected to your system, run the analysis on the GPU of your choice or on all connected devices. For more information, see [Analyze Multiple GPUs](#).

3. Configure these GPU analysis options:

- Use the **Trace GPU programming APIs** option to analyze SYCL, Level-Zero, OpenCL™, and Intel Media SDK programs running on Intel Processor Graphics. This option may affect the performance of your application on the CPU side.
- Use the **Collect host stacks** option to analyze call stacks executed on the CPU and identify critical paths. You can also examine the CPU-side stacks for GPU computing tasks to investigate the efficiency of your GPU offload. When results display, sort through SYCL*, Level-Zero, or OpenCL™ runtime call stacks by selecting a **Call Stack** mode in the filter bar.
- Use the **Analyze CPU-GPU bandwidth** option to display data transfers based on hardware events on the timeline. This type of analysis requires [Intel sampling drivers](#) to be installed.
- For GPUs with Xe Link connections, use the **Analyze Xe Link Usage** option to examine the traffic between GPU interconnects (Xe Link). This information can help you assess data flow between GPUs and the usage of the Xe Link.
- Use the **Show GPU performance insights** to get metrics (based on the analysis of Processor Graphics events) that help you estimate the efficiency of hardware usage and learn next steps. The following Insights metrics are collected:
 - The **EU Array** metric shows the breakdown of GPU core array cycles, where:

- **Active:** The normalized sum of all cycles on all cores spent actively executing instructions.
Formula:

$$\frac{\sum_{\text{across all EUs}} \text{cycles when EU executes instructions}}{\sum_{\text{across all EUs}} \text{all cycles}}$$

- **Stalled:** The normalized sum of all cycles on all cores spent stalled. At least one thread is loaded, but the core is stalled for some reason. Formula:

$$\frac{\sum_{\text{across all EUs}} \text{cycles when EU does not execute instructions and at least one thread is scheduled on EU}}{\sum_{\text{across all EUs}} \text{all cycles}}$$

- **Idle:** The normalized sum of all cycles on all cores when no threads were scheduled on a core.
Formula:

$$\frac{\sum_{\text{across all EUs}} \text{cycles when no threads scheduled on EU}}{\sum_{\text{across all EUs}} \text{all cycles}}$$

- The **EU Threads Occupancy** metric shows the normalized sum of all cycles on all cores and thread slots when a slot has a thread scheduled.
- The **Computing Threads Started** metric shows the number of threads started across all EUs for compute work.

HOW

GPU Offload ▾

Explore code execution on various CPU and GPU cores on your platform, estimate how your code benefits from offloading to the GPU, and identify whether your application is CPU or GPU bound. [Learn more](#)

Trace GPU programming APIs
 Collect host stacks
 Analyze CPU host-GPU bandwidth
 Show GPU performance insights

Details

4. Click **Start** to run the analysis.

NOTE Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see [GPU Architecture Terminology for Intel® Xe Graphics](#).

Run from Command Line

Type this command:

```
$ vtune -collect gpu-offload [-knob <knob_name=knob_option>] -- <target>  
[target_options]
```

NOTE

To [generate the command line](#) for any analysis configuration, use the **Command Line** button at the bottom of the interface.

Once the GPU Offload Analysis completes data collection, the **Summary** window displays metrics that describe:

- GPU usage
- GPU idle time
- Xe Link Usage
- The most active computing tasks that ran on the CPU host
- The most active computing tasks that ran on the CPU when the GPU was idle
- The most active computing tasks that ran on the GPU, along with occupancy information

Analysis Configuration Collection Log Summary Graphics Platform

Elapsed Time [?]: 8.684s

GPU Time, % of Elapsed time [?]: 1.2% 

Use this section to understand whether the GPU was utilized properly and which of the engines were utilized, at least one piece of work scheduled to them.

GPU Time, % of Elapsed time

GPU Utilization breakdown by GPU engines.

GPU Adapter / GPU Stack / GPU Engine	GPU Time	GPU Time, % of Elapsed time [?]
GPU 3	0.134s	1.2% 
GPU Stack 0	0.029s	0.3% 
Render and GPGPU	0.029s	0.3% 
GPU Stack 1	0.105s	1.2% 
Render and GPGPU	0.105s	1.2% 
GPU 5	0.133s	1.2% 
GPU Stack 0	0.028s	0.3% 
Render and GPGPU	0.028s	0.3% 
GPU Stack 1	0.105s	1.2% 
Render and GPGPU	0.105s	1.2% 
GPU 1	0.134s	1.2% 
GPU Stack 0	0.028s	0.3% 
Render and GPGPU	0.028s	0.3% 
GPU Stack 1	0.106s	1.2% 
Render and GPGPU	0.106s	1.2% 
GPU 0	0.057s	0.3% 
GPU Stack 0	0.029s	0.3% 
Render and GPGPU	0.029s	0.3% 
GPU Stack 1	0.028s	0.3% 
Render and GPGPU	0.028s	0.3% 
GPU 2	0.133s	1.2% 
GPU Stack 0	0.028s	0.3% 
Render and GPGPU	0.028s	0.3% 
GPU Stack 1	0.105s	1.2% 
Render and GPGPU	0.105s	1.2% 
GPU 4	0.133s	1.2% 
GPU Stack 0	0.028s	0.3% 
Render and GPGPU	0.028s	0.3% 
GPU Stack 1	0.105s	1.2% 
Render and GPGPU	0.105s	1.2% 

*N/A is applied to non-summable metrics.

Top Hotspots when GPU was idle

You also see **Recommendations** and guidance for next steps.

Analyze Multiple GPUs

If you connect multiple Intel GPUs to your system, VTune Profiler identifies all of these adapters in the **Target GPU** pull down menu. Follow these guidelines:

- Use the **Target GPU** pulldown menu to specify the device you want to profile.

- The **Target GPU** pulldown menu displays only when VTune Profiler detects multiple GPUs running on the system. The menu then displays the name of each GPU with the bus/device/function (BDF) of its adapter. You can also find this information on your Windows (see Task Manager) or Linux (run `lspci`) system.
- If you do not select a GPU, VTune Profiler selects the most recent device family in the list by default.
- Select **All devices** to run the analysis on all of the GPUs connected to your system.
- Full compute set in **Characterization** mode is not available for multi-adapter/tile analysis.

Once the analysis completes, VTune Profiler displays summary results per GPU including tile information in the **Summary** window.

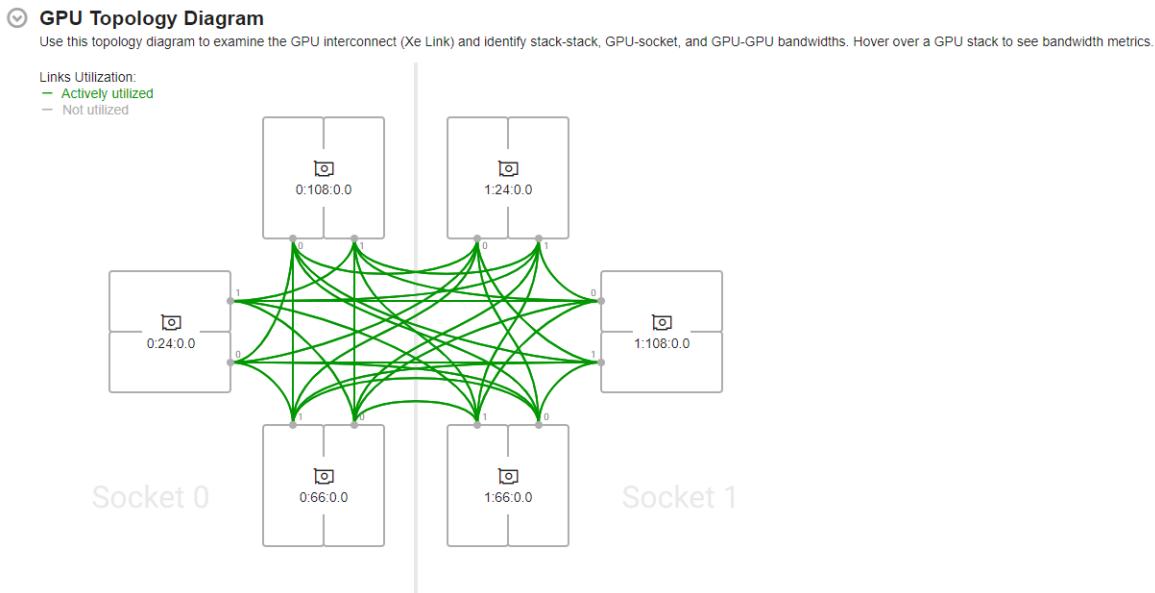
Naming Convention for GPU Adapters

The results of GPU profiling analyses use aliases to refer to GPU adapters. .

- Aliases identify GPU adapters in the **Summary**, **Grid**, and **Timeline** sections of profiling results. The full names of GPU adapters display in the **Collection** and **Platform Information** sections, along with BDF details.
- A single alias identifies a GPU adapter for all results collected on the same machine.
- Aliases follow the naming convention GPU 0, GPU 1, and so on.
- The assignment of aliases happens in this order:
 1. Intel GPU adapters, starting with the lowest PCI address
 2. Non-Intel GPU adapters
 3. Other software devices like drivers

The GPU Topology Diagram

When you run a GPU analysis across multiple Intel GPUs (or multi-stack GPUs) connected to your system, the Summary window displays interconnections between these GPUs in the **GPU Topology diagram**. This diagram contains cross-GPU information for a maximum of 2 sockets and 6 GPUs connected to the system.

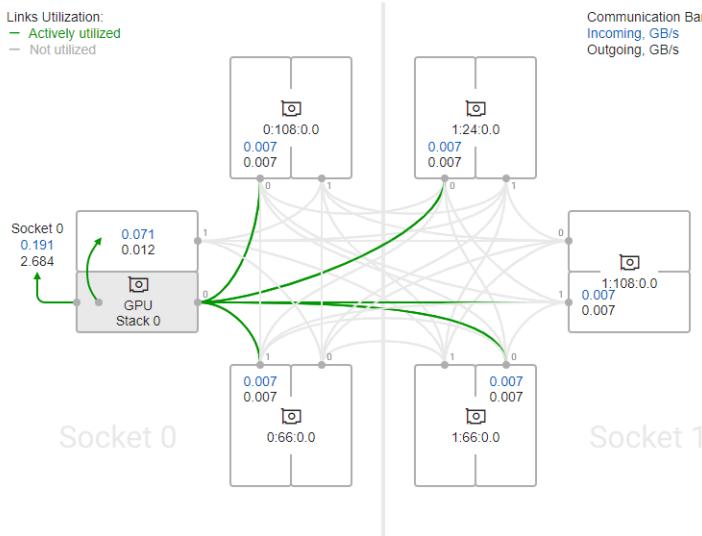


The GPU Topology diagram displays topological information about the sockets (available for GPU connection) as well as interconnect (Xe Link) connections between GPUs. You can identify GPUs in the GPU Topology diagram by their Bus Device Function (BDF) numbers.

Hover over a GPU stack to see actively utilized links (highlighted in green) and corresponding bandwidth metrics.

GPU Topology Diagram

Use this topology diagram to examine the GPU interconnect (Xe Link) and identify stack-stack, GPU-socket, and GPU-GPU bandwidths. Hover over a GPU stack to see bandwidth metrics.



Use the information presented here to see average data transferred:

- Through Xe Links
- Between GPU stacks
- Between GPUs and sockets

Analyze Xe Link Usage

For GPUs with Xe Link connections, when you check the option (before running the analysis) to analyze interconnect (Xe Link) usage, the **Summary** window includes a section that displays the aggregated bandwidth and traffic data through GPU interconnects (Xe Link). Use this information with the GPU Topology diagram to detect any imbalances in the distribution of traffic between GPUs. See if some links are used more frequently than others, and understand why this is happening.

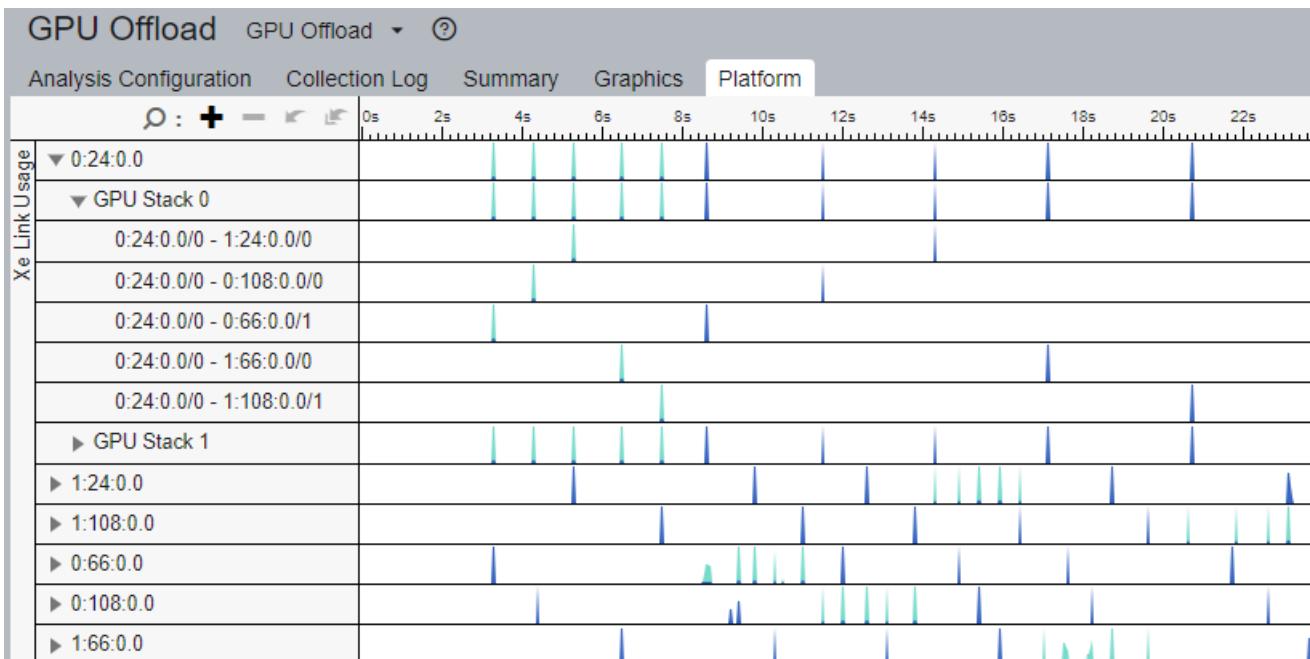
Xe Link Usage

This section lists the aggregated bandwidth and traffic data through GPU interconnect (Xe Link).

From GPU/Stack - To GPU/Stack	Outgoing, MB	BW Outgoing, MB/sec	Incoming, MB	BW Incoming, MB/sec
0:24:0.0/0 - 1:24:0.0/0	168.002	7.065	168.002	7.065
0:24:0.0/0 - 0:108:0.0/0	168.002	7.065	168.002	7.065
0:24:0.0/0 - 0:66:0.0/1	168.002	7.065	168.002	7.065
0:24:0.0/0 - 1:66:0.0/0	168.002	7.065	168.002	7.065
0:24:0.0/0 - 1:108:0.0/1	168.002	7.065	168.002	7.065
1:108:0.0/0 - 1:24:0.0/1	168.002	7.065	168.002	7.065
1:108:0.0/0 - 0:24:0.0/1	168.002	7.065	168.002	7.065
1:108:0.0/0 - 0:66:0.0/0	168.002	7.065	168.002	7.065
1:108:0.0/0 - 1:66:0.0/1	168.002	7.065	168.002	7.065
1:108:0.0/0 - 0:108:0.0/1	168.002	7.065	168.002	7.065
0:66:0.0/0 - 0:108:0.0/1	168.002	7.065	168.002	7.065
0:66:0.0/0 - 0:24:0.0/1	168.002	7.065	168.002	7.065
0:66:0.0/0 - 1:108:0.0/0	168.002	7.065	168.002	7.065
0:66:0.0/0 - 1:66:0.0/1	168.002	7.065	168.002	7.065
0:66:0.0/0 - 1:24:0.0/1	168.002	7.065	168.002	7.065

Analyze Overtime Data

Along with the Xe Link usage information in the **Summary** window, the **Platform** window displays bandwidth data over time.



Use this information to:

- Match traffic data with kernels or code execution.
- See the bandwidth during any time of the execution of the application.
- Understand how the use of Xe Links improves the performance of your application.
- Verify if the Xe Links reached the bandwidth expected during application execution.

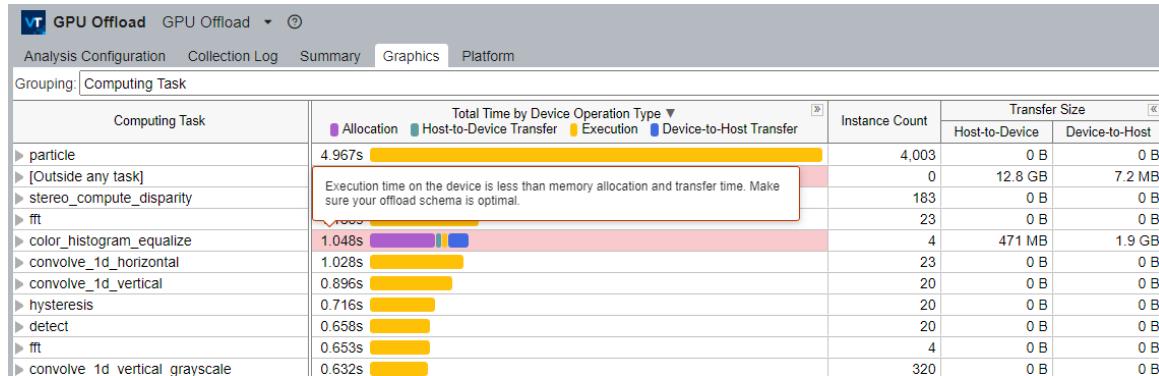
Analyze Data Transfer Between Host and Device

To understand the efficiency of data transfer between the CPU host and GPU device, see metrics in the **Summary** and **Graphics** windows.

The Summary window displays the total time spent on computing tasks as well as the execution time per task. The difference indicates the amount of time spent on data transfers between host and device. If the execution time is lower than the data transfer time, this indicates that your offload schema could benefit from optimization.

In the Summary window, look for offload cost metrics including **Host-to-Device Transfer** and **Device-to-Host Transfer**. These metrics can help you locate unnecessary memory transfers that reduce performance.

In the Graphics window, see the **Total Time by Device Operation Type** column, which displays the total time for each computation task.



The total time is broken down into:

- Allocation time
- Time for data transfer from host to device
- Execution time
- Time for data transfer from device to host

This breakdown can help you understand better the balance between data transfer and GPU execution time. The Graphics window also displays in the **Transfer Size** section, the size of the data transfer between host and device per computation task.

Computation tasks with sub-optimal offload schemas are highlighted in the table with details to help you improve those schemes.

Computing Task	Total Time by Device Operation Type	Instance Count	Transfer Size	
			Host-to-Device	Device-to-Host
particle	4.967s	4,003	0 B	0 B
[Outside any task]		0	12.8 GB	7.2 MB
stereo_compute_disparity		183	0 B	0 B
fft		23	0 B	0 B
color_histogram_equalize	1.048s	4	471 MB	1.9 GB
convolve_1d_horizontal	1.028s	23	0 B	0 B
convolve_1d_vertical	0.896s	20	0 B	0 B
hysteresis	0.716s	20	0 B	0 B
detect	0.658s	20	0 B	0 B
fft	0.653s	4	0 B	0 B
convolve_1d_vertical_grayscale	0.632s	320	0 B	0 B

Examine Energy Consumption by your GPU

In Linux environments, when you run the GPU Offload analysis on an Intel® Iris® X e MAX graphics discrete GPU, you can see energy consumption information for the GPU device. To collect this information, make sure you check the **Analyze power usage** option when you configure the analysis.

GPU Offload ▾
Explore code execution on various CPU and GPU cores on your platform, estimate how your code benefits from offloading to the GPU, and identify whether your application is CPU or GPU bound. [Learn more](#)

Trace GPU programming APIs

Collect host stacks

Analyze CPU host-GPU bandwidth

Show GPU performance insights

Analyze power usage

NOTE Energy consumption metrics do not display in GPU profiling analyses that scan Intel® Iris® X e MAX graphics on Windows machines.

Once the analysis completes, see energy consumption data in these sections of your results.

In the **Graphics** window, observe the **Energy Consumption** column in the grid when grouped by **Computing Task**. Sort this column to identify the GPU kernels that consumed the most energy. You can also see this information mapped in the timeline.

Tune for Power Usage

When you locate individual GPU kernels that consume the most energy, for optimum power efficiency, start by tuning the top energy hotspot.

Tune for Processing Time

If your goal is to optimize GPU processing time, keep a check on energy consumption metrics per kernel to monitor the tradeoff between performance time and power use.

Move the **Energy Consumption** column next to **Total Time** to make this comparison easier.

Computing Task	Energy Consumption (mJ) ▼	Computing Task					Work Size	
		Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	Global	Local
► clEnqueueReadImage	419829.895	48.966s	0.816s	60				
► clEnqueueWriteBuffer	47357.239	4.841s	0.004s	1,131				
► clEnqueueWriteImage	38314.819	3.731s	0.041s	90				
► particle	36702.637	1.807s	0.000s	4,004	32		262144	64
► clEnqueueReadBuffer	22355.591	2.125s	0.003s	812				
► stereo_compute_disparity	14262.390	0.646s	0.004s	183	32		464 x 384	16 x 16
► convolve_1d_vertical_grayscale	8891.663	0.531s	0.002s	320	32		656 x 856	16 x 8
► convolve_1d_horizontal_grayscale	8879.944	0.533s	0.002s	320	32		704 x 852	64 x 4
► detect	6203.857	0.407s	0.020s	20	16		167136	
► convolve_1d_vertical	4898.315	0.230s	0.011s	20	32		5960 x 4096	4 x 64
► knn_match	4764.221	0.302s	0.015s	20	16		4096	64
► compute_keypoints	4567.261	0.300s	0.001s	320	32		592 x 800	8 x 16
► convolve_1d_horizontal	4056.946	0.229s	0.011s	20	32		5960 x 4096	4 x 64
► hough_get_scores	3087.097	0.207s	0.010s	20	32		2624 x 3584	64 x 2
► blur	2429.260	0.119s	0.006s	20	16		5984 x 4096	32 x 2
► fast_corners	2420.593	0.135s	0.000s	320	16		592 x 792	16 x 8
► non_maximum_suppression	2346.069	0.107s	0.005s	20	32		5968 x 4096	16 x 16
► hysteresis	2315.308	0.145s	0.007s	20	32		23550464	64
► calculate_descriptors	1897.156	0.124s	0.000s	323	8		4032	64
► fft	1557.983	0.103s	0.026s	4	16		4194304	32

You may notice that the correlation between power use and processing time is not direct. The kernels that compute the fastest may not be the same kernels that consume the least amounts of energy. Check to see if larger values of power usage correspond to longer stalls/wait periods.

Support for SYCL® Applications using oneAPI Level Zero API

This section describes support in the GPU Offload analysis for SYCL applications that run OpenCL or [oneAPI Level Zero API](#) in the back end. VTune Profiler supports version 1.0.4 of the [oneAPI Level Zero API](#).

Support Aspect	SYCL application with OpenCL as back end	SYCL application with Level Zero as back end
Operating System	Linux OS Windows OS	Linux OS Windows OS
Data collection	VTune Profiler collects and shows GPU computing tasks and the GPU computing queue.	VTune Profiler collects and shows GPU computing tasks and the GPU computing queue.
Data display	VTune Profiler maps the collected GPU HW metrics to specific kernels and displays them on a diagram.	VTune Profiler maps the collected GPU HW metrics to specific kernels and displays them on a diagram.
Display Host side API calls	Yes	Yes

Support Aspect	SYCL application with OpenCL as back end	SYCL application with Level Zero as back end
Source Assembler for computing tasks	Yes	Yes

Support for DirectX Applications

This section describes support available in the GPU analysis to trace Microsoft® DirectX* applications running on the CPU host. This support is available in the **Launch Application** mode only.

Support Aspect	DirectX Application
Operating system	Windows OS
API version	DXGI, Direct3D 11, Direct3D 12, Direct3D 11 on 12
Display host side API calls	Yes
Direct Machine Learning (DirectML) API	Yes
Device side computing tasks	No
Source Assembler for computing tasks	No

See Also

[Optimize applications for Intel® GPUs with Intel® VTune Profiler](#)
[Offload and Optimize OpenMP* Applications with Intel Tools](#)
[GPU Architecture Terminology for Intel® Xe Graphics](#)
[Debug the DPC++ and OpenMP* Offload Process](#)
[OneTrace Tracing and Profiling Tool for Data Parallel C++ \(DPC++\)](#)

GPU Compute/Media Hotspots Analysis (Preview)

Analyze the most time-consuming GPU kernels, characterize GPU usage based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types.

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

Use the GPU Compute/Media Hotspots analysis to:

- Explore GPU kernels with high GPU utilization, estimate the effectiveness of this utilization, identify possible reasons for stalls or low occupancy and options.
- Explore the performance of your application per selected GPU metrics over time.
- Analyze the hottest SYCL* standards or OpenCL™ kernels for inefficient kernel code algorithms or incorrect work item configuration.

The GPU Compute/Media Hotspots analysis is a good next step if you have already run the [GPU Offload](#) analysis and identified:

- a performance-critical kernel for further analysis and optimization;

- a performance-critical kernel that it is tightly connected with other kernels in the program and may slow down their performance.

HOW

GPU Compute/Media Hotspots (preview)

Analyze the most time-consuming GPU kernels, characterize GPU utilization based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types. [Learn more](#)

Characterization [?](#)

Source Analysis [?](#)

Basic Blocks Latency

Computing task of interest	Instance step
Enter computing task of interest	Def

Overhead

Details >

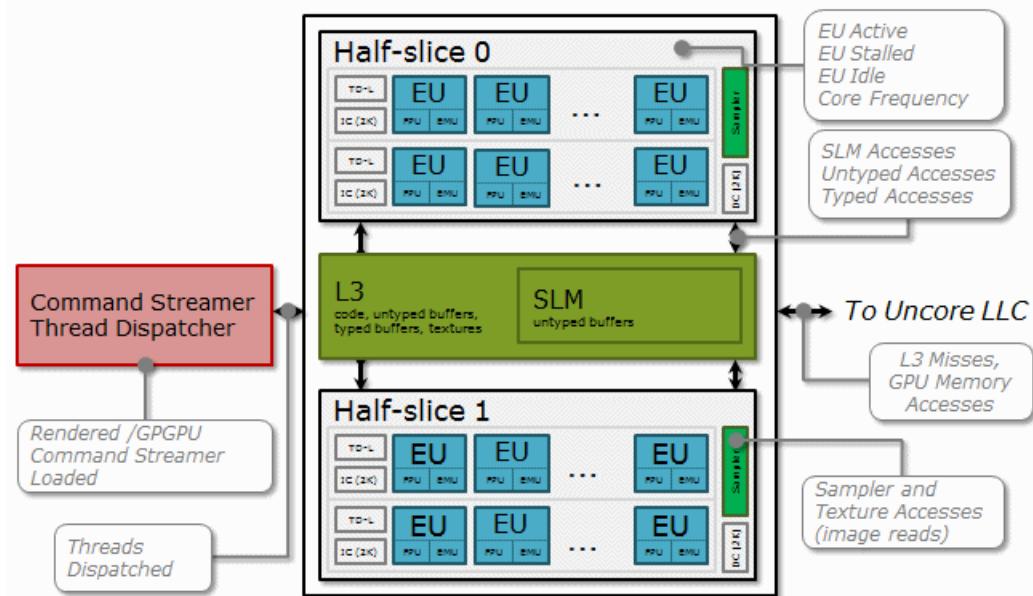
How It Works: Intel Graphics Render Engine and Hardware Metrics

A GPU is a highly parallel machine where an array of small cores, or execution units (EUs), do graphical or computational work. Each EU simultaneously runs several lightweight threads. When one of these threads is picked up for an execution, it can hide stalls in the other threads if the other threads are stalled waiting for data from memory or other units.

To use a full potential of the GPU, applications should enable the scheduling of as many threads as possible and minimize idle cycles. Minimizing stalls is also very important for graphics and general purpose computing GPU applications.

VTune Profiler can monitor Intel Graphics hardware events and display metrics about integral GPU resource usage over a sampled period, for example, ratio of cycles when EUs were idle, stalled, or active as well as statistics on memory accesses and other functional units. If the VTune Profiler traces GPU kernel execution, it annotates each kernel with GPU metrics.

The scheme below displays metrics collected by the VTune Profiler across different parts of the Intel® Processor Graphics Gen9:



GPU metrics help identify how efficiently GPU hardware resources are used and whether any performance improvements are possible. Many metrics are represented as a ratio of cycles when the GPU functional unit(s) is in a specific state over all the cycles available for a sampling period.

Configure the Analysis

- Make sure you [set up the system and enable required permissions](#) for GPU analysis.
- For SYCL applications: make sure to compile your code with the `-gline-tables-only` and `-fdebug-info-for-profiling` Intel oneAPI DPC++ Compiler options.
- [Create a project](#) and specify an analysis system and target.

Run the Analysis

1. Click the 

(standalone GUI)/ 

(Visual Studio IDE) **Configure Analysis** toolbar button to open the **Configure Analysis** window .

2. Click anywhere in the title bar of the **HOW** pane. Open the Analysis Tree and select **GPU Compute/ Media Hotspots (Preview)** analysis from the **Accelerators** group. This analysis is pre-configured to collect GPU usage data, analyze GPU task scheduling and identify whether your application is CPU or GPU bound.

NOTE

If you have multiple Intel GPUs connected to your system, run the analysis on the GPU of your choice or on all connected devices. For more information, see [Analyze Multiple GPUs](#).

3. Choose and configure one of these analysis modes:
 - [Characterization](#)
 - [Source analysis](#)
4. Optionally, narrow down the analysis to specific kernels you identified as performance-critical (stalled or time-consuming) in the GPU Offload analysis, and specify them as **Computing tasks of interest** to profile. If required, modify the **Instance step** for each kernel, which is a sampling interval (in the number of kernels). This option helps reduce profiling overhead.

5. **(Optional)** To collect data on energy consumption, check the **Analyze power usage** option. This feature is available when you profile applications in a Linux environment and use an Intel® Iris® Xe MAX graphics discrete GPU.
6. For GPUs with interconnect (Xe Link) connections, use the **Analyze Xe Link Usage** option to examine the traffic between GPU interconnects (or Xe links). This information can help you assess data flow between GPUs and the usage of their interconnects.
7. Click **Start** to run the analysis.

Run from Command Line

To run the GPU Compute/Media Hotspots analysis from the command line, type:

```
vtune -collect gpu-hotspots [-knob <knob_name=knob_option>] -- <target>  
[target_options]
```

NOTE

To generate the command line for this configuration, use the **Command Line...** button at the bottom.

Analyze Multiple GPUs

If you connect multiple Intel GPUs to your system, VTune Profiler identifies all of these adapters in the **Target GPU** pulldown menu. Follow these guidelines:

- Use the **Target GPU** pulldown menu to specify the device you want to profile.
- The **Target GPU** pulldown menu displays only when VTune Profiler detects multiple GPUs running on the system. The menu then displays the name of each GPU with the bus/device/function (BDF) of its adapter. You can also find this information on your Windows (see Task Manager) or Linux (run `lspci`) system.
- If you do not select a GPU, VTune Profiler selects the most recent device family in the list by default.
- Select **All devices** to run the analysis on all of the GPUs connected to your system.
- Full compute set in **Characterization** mode is not available for multi-adapter/tile analysis.

Once the analysis completes, VTune Profiler displays summary results per GPU including tile information in the **Summary** window.

Analysis Results

Once the GPU Compute/Media Hotspots Analysis completes data collection, the **Summary** window displays metrics that describe:

- GPU time
- Occupancy
- Peak occupancy you can expect with the existing computing task configuration
- The most active computing tasks that ran on the GPU

() ▾ ③ ⓘ

roup

is group.

s in this group.

s with maximum contribution to the high bandwidth utilization.

the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn more about bandwidth utilization types, see [Bandwidth Utilization Types](#).

rm data.

E=20.04 DISTRIB_CODENAME=focal DISTRIB_DESCRIPTION="Ubuntu 20.04.1 LTS"

NOTE Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see [GPU Architecture Terminology for Intel® Xe Graphics](#).

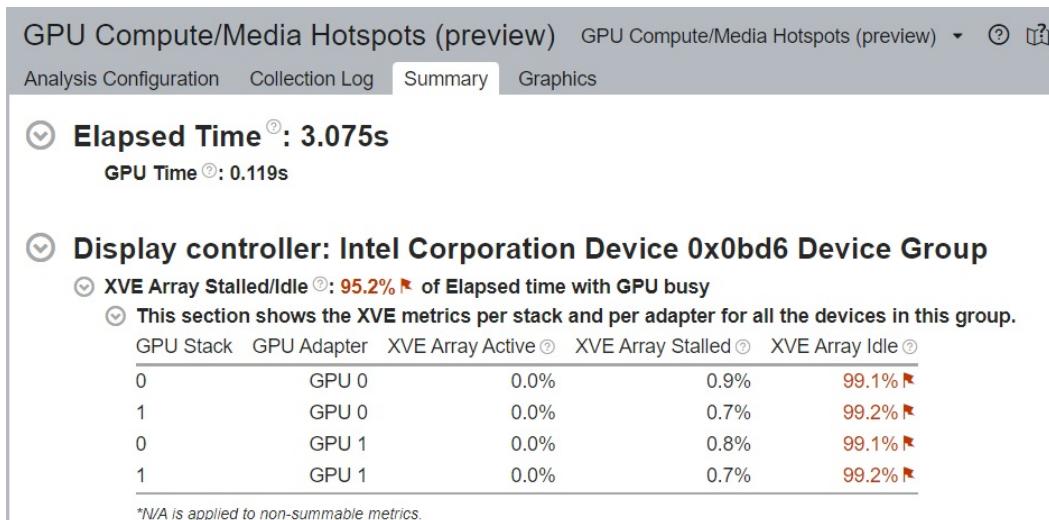
Analysis Results for Multiple GPUs

When you profile an application running on multiple Intel® GPUs, the **Summary** window of the GPU Compute/Media Hotspots Analysis displays results by grouping GPUs of the same Intel microarchitecture. Each architecture group then contains metric information for that group.

For each architecture group, you can see the values for these metrics:

- Occupancy
- GPU L3 Bandwidth Bound

For every adapter in an architecture group, you can also see the values for the XVE Array Active/Stalled/Idle metrics.



Naming Convention for GPU Adapters

The results of GPU profiling analyses use aliases to refer to GPU adapters. .

- Aliases identify GPU adapters in the **Summary**, **Grid**, and **Timeline** sections of profiling results. The full names of GPU adapters display in the **Collection** and **Platform Information** sections, along with BDF details.
- A single alias identifies a GPU adapter for all results collected on the same machine.
- Aliases follow the naming convention GPU 0, GPU 1, and so on.
- The assignment of aliases happens in this order:
 1. Intel GPU adapters, starting with the lowest PCI address
 2. Non-Intel GPU adapters
 3. Other software devices like drivers

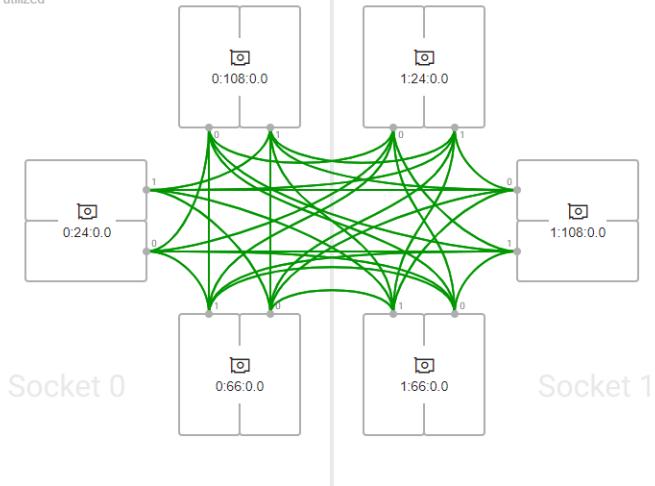
The GPU Topology Diagram

When you run a GPU analysis across multiple Intel GPUs (or multi-stack GPUs) connected to your system, the Summary window displays interconnections between these GPUs in the **GPU Topology diagram**. This diagram contains cross-GPU information for a maximum of 2 sockets and 6 GPUs connected to the system.

GPU Topology Diagram

Use this topology diagram to examine the GPU interconnect (Xe Link) and identify stack-stack, GPU-socket, and GPU-GPU bandwidths. Hover over a GPU stack to see bandwidth metrics.

Links Utilization:
— Actively utilized
— Not utilized



The GPU Topology diagram displays topological information about the sockets (available for GPU connection) as well as interconnect (Xe Link) connections between GPUs. You can identify GPUs in the GPU Topology diagram by their Bus Device Function (BDF) numbers.

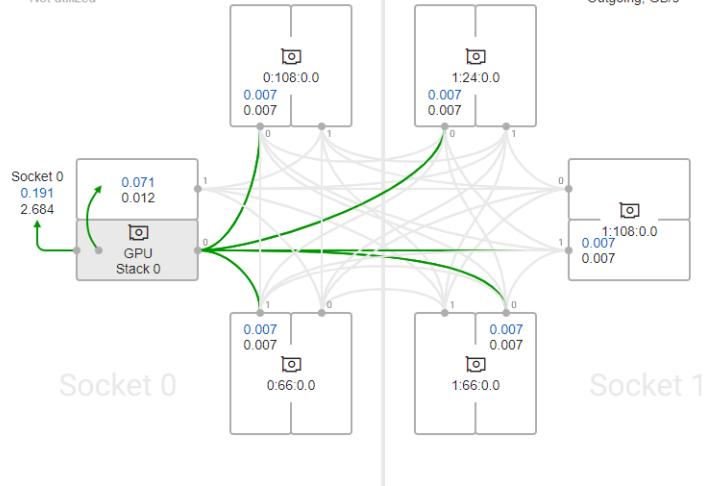
Hover over a GPU stack to see actively utilized links (highlighted in green) and corresponding bandwidth metrics.

GPU Topology Diagram

Use this topology diagram to examine the GPU interconnect (Xe Link) and identify stack-stack, GPU-socket, and GPU-GPU bandwidths. Hover over a GPU stack to see bandwidth metrics.

Links Utilization:
— Actively utilized
— Not utilized

Communication Bandwidth:
Incoming, GB/s
Outgoing, GB/s



Use the information presented here to see average data transferred:

- Through Xe Links
- Between GPU stacks
- Between GPUs and sockets

Analyze Xe Link Usage

For GPUs with Xe Link connections, when you check the option (before running the analysis) to analyze interconnect (Xe Link) usage, the **Summary** window includes a section that displays the aggregated bandwidth and traffic data through GPU interconnects (Xe Link). Use this information with the GPU Topology diagram to detect any imbalances in the distribution of traffic between GPUs. See if some links are used more frequently than others, and understand why this is happening.

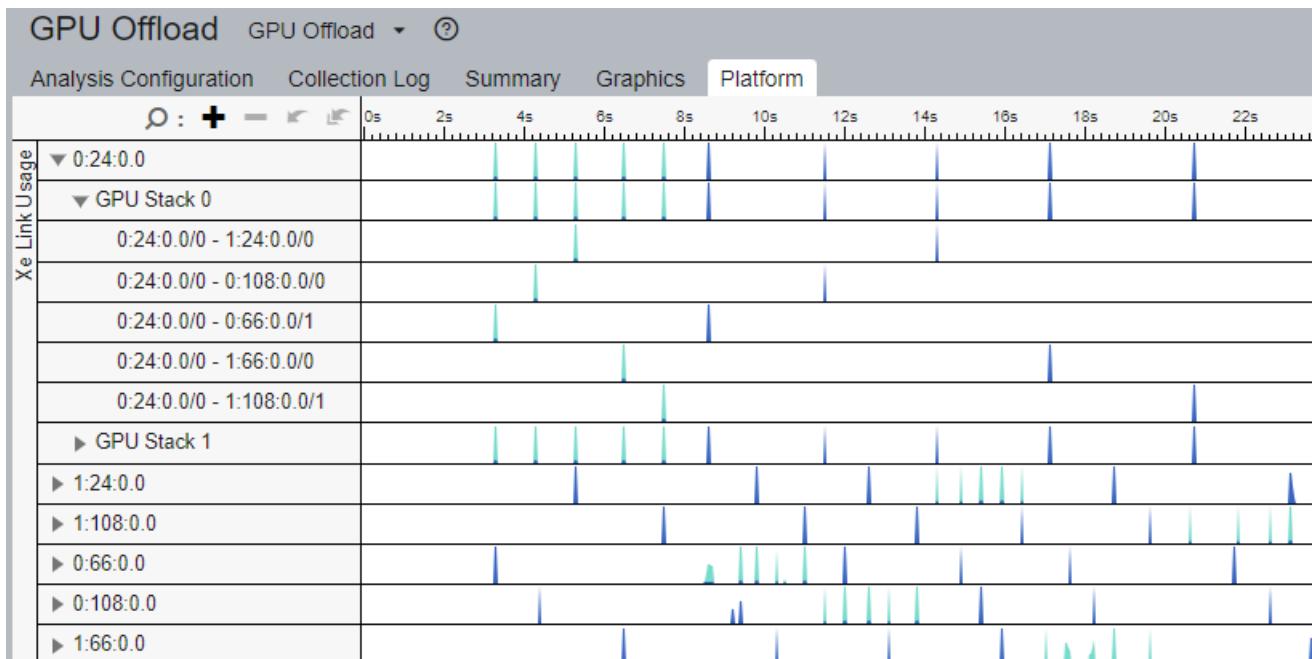
Xe Link Usage

This section lists the aggregated bandwidth and traffic data through GPU interconnect (Xe Link).

From GPU/Stack - To GPU/Stack	Outgoing, MB	BW Outgoing, MB/sec	Incoming, MB	BW Incoming, MB/sec
0:24:0:0/0 - 1:24:0:0/0	168.002	7.065	168.002	7.065
0:24:0:0/0 - 0:108:0:0/0	168.002	7.065	168.002	7.065
0:24:0:0/0 - 0:66:0:0/1	168.002	7.065	168.002	7.065
0:24:0:0/0 - 1:66:0:0/0	168.002	7.065	168.002	7.065
0:24:0:0/0 - 1:108:0:0/1	168.002	7.065	168.002	7.065
1:108:0:0/0 - 1:24:0:0/1	168.002	7.065	168.002	7.065
1:108:0:0/0 - 0:24:0:0/1	168.002	7.065	168.002	7.065
1:108:0:0/0 - 0:66:0:0/0	168.002	7.065	168.002	7.065
1:108:0:0/0 - 1:66:0:0/1	168.002	7.065	168.002	7.065
1:108:0:0/0 - 0:108:0:0/1	168.002	7.065	168.002	7.065
0:66:0:0/0 - 0:108:0:0/1	168.002	7.065	168.002	7.065
0:66:0:0/0 - 0:24:0:0/1	168.002	7.065	168.002	7.065
0:66:0:0/0 - 1:108:0:0/0	168.002	7.065	168.002	7.065
0:66:0:0/0 - 1:66:0:0/1	168.002	7.065	168.002	7.065
0:66:0:0/0 - 1:24:0:0/1	168.002	7.065	168.002	7.065

Analyze Bandwidth Data Over Time

Along with the Xe Link usage information in the **Summary** window, the **Platform** window displays bandwidth data over time.



Use this information to:

- Match traffic data with kernels or code execution.
- See the bandwidth during any time of the execution of the application.
- Understand how the use of Xe Links improves the performance of your application.
- Verify if the Xe Links reached the bandwidth expected during application execution.

Configure Characterization Analysis

Use the **Characterization** configuration option to:

- Monitor the Render and GPGPU engine usage (Intel Graphics only)
- Identify the loaded parts of the engine
- Correlate GPU and CPU data

When you select the **Characterization** radio button, you can select platform-specific presets of GPU metrics. With the exception of the Dynamic Instruction Count preset, all other presets collect the following data about the activity of Execution Units (EU):

- EU Array Active
- EU Array Stalled
- EU Array Idle
- Computing Threads Started
- Thread Occupancy
- Core Frequency

Each preset introduces additional metrics:

- The **Overview** metric set includes additional metrics that track general GPU memory accesses such as Memory Read/Write Bandwidth and XVE pipelines utilization. These metrics can be useful for both graphics and compute-intensive applications.
- The **Global Memory Accesses** metric group includes additional metrics that show the bandwidth between the GPU and system memory as well as bandwidth between GPU stacks. The farther a memory level is located from an XVE, the greater the impact on its performance by unnecessary access operations to the memory level.
- The **LSE/SLM Accesses** metric group includes metrics which cover the XVE to L1 cache traffic. This metric group requires two application runs to collect information.
- The **HDC Accesses** metric group includes metrics which measure the traffic between XVE and L3, that is passing by the L1 cache.
- The **Full Compute** metric group is a combination of all of the other event sets. Therefore, it requires multiple application runs.
- The **Dynamic Instruction Count** metric group counts the execution frequency of specific classes of instructions. With this metric group, you also get an insight into the efficiency of SIMD utilization by each kernel.

NOTE

You can run the GPU Compute/Media Hotspots analysis in Characterization mode for Windows* and Linux* targets. However, for all presets (with the exception of the **Dynamic Instruction Count** preset), you must have root/administrative privileges to run the GPU Compute/Media Hotspots analysis in Characterization mode.

Alternatively, on Linux* systems, you can configure the system to allow further collections for non-privileged users. To do this, in the bin64 folder of your installation directory, run the [prepare-debugfs-and-gpu-environment.sh](#) script with root privileges.

Additional Data for Characterization Analysis

Use the **Trace GPU programming APIs** option to analyze SYCL, OpenCL™, or Intel Media SDK programs running on Intel Processor Graphics. This option may affect the performance of your application on the CPU side.

For SYCL or OpenCL applications, identify the hottest kernels and the GPU architecture block that contains a performance issue for a particular kernel.

For Intel Media SDK programs, examine the execution of Intel Media SDK tasks on the timeline. Correlate this data with GPU usage at each instant.

Usage Considerations:

- For OpenCL kernels, you can run the characterization analysis for Windows and Linux targets running on Intel Graphics.
- The Intel Media SDK program analysis is available for Windows and Linux targets running on Intel Graphics.
- The characterization analysis supports the **Launch Application** and **Attach to Process** target types only.
- The Level Zero runtime does not support the **Attach to Process** target type.
- When profiling OpenCL kernels in the **Attach to Process** mode, if you attached to a process when the computing queue is already created, VTune Profiler does not display data for the OpenCL kernels in this queue.
- To collect the data required to compute memory bandwidth, use the **Analyze memory bandwidth** option . For this analysis, install [Intel sampling drivers](#) first.
- Use the **GPU sampling internal, ms** field to specify an interval (in milliseconds) between GPU samples for GPU hardware metrics collection. By default, VTune Profiler uses an interval of 1ms.

Configure Source Analysis

In the Source Analysis, VTune Profiler helps you identify performance-critical basic blocks, issues caused by memory accesses in the GPU kernels.

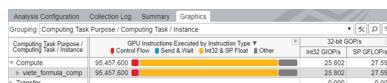
When you select the **Source Analysis** radio button, the configuration pane expands a drop-down menu where you can select a profiling mode to specify a type of issues you want to analyze:

- **Basic Block Latency** option helps you identify issues caused by algorithm inefficiencies. In this mode, VTune Profiler measures the execution time of all basic blocks. Basic block is a straight-line code sequence that has a single entry point at the beginning of the sequence and a single exit point at the end of this sequence. During post-processing, VTune Profiler calculates the execution time for each instruction in the basic block. So, this mode helps understand which operations are more expensive.
- **Memory Latency** option helps identify latency issues caused by memory accesses. In this mode, VTune Profiler profiles memory read/synchronization instructions to estimate their impact on the kernel execution time. Consider using this option, if you ran the GPU Compute/Media Hotspots analysis in the Characterization mode, identified that the GPU kernel is throughput or memory-bound, and want to explore which memory read/synchronization instructions from the same basic block take more time.

In the **Basic Block Latency** or **Memory Latency** profiling modes, the GPU Compute/Media Hotspots analysis uses these metrics:

- **Estimated GPU Cycles**: The average number of cycles spent by the GPU executing the profiled instructions.
- **Average Latency**: The average latency of the memory read and synchronization instructions, in cycles.
- **GPU Instructions Executed per Instance**: The average number of GPU instructions executed per one kernel instance.
- **GPU Instructions Executed per Thread**: The average number of GPU instructions executed by one thread per one kernel instance.

If you enable the **Instruction count** profiling mode, VTune Profiler shows a breakdown of instructions executed by the kernel in the following groups:



Control Flow group	if, else, endif, while, break, cont, call, calla, ret, goto, jmpi, brd, brc, join, halt and mov, add instructions that explicitly change the ip register.
Send & Wait group	send, sends, sendc, sendsc, wait
Int16 & HP Float Int32 & SP Float Int64 & DP Float groups	<p>Bit operations (only for integer types): and, or, xor, and others.</p> <p>Arithmetic operations: mul, sub, and others; avg, frc, mac, mach, mad, madm.</p> <p>Vector arithmetic operations: line, dp2, dp4, and others.</p> <p>Extended math operations.</p>
Other group	Contains all other operations including nop.

In the **Instruction count** mode, the VTune Profiler also provides **Operations per second** metrics calculated as a weighted sum of the following executed instructions:

- Bit operations (only for integer types):
 - and, not, or, xor, asr, shr, shl, bfre, bfe, bfi1, bfi2, ror, rol - weight 1
- Arithmetic operations:
 - add, addc, cmp, cmpn, mul, rndu, rndd, rnde, rndz, sub - weight 1
 - avg, frc, mac, mach, mad, madm - weight 2
- Vector arithmetic operations:
 - line - weight 2
 - dp2, sad2 - weight 3
 - lrp, pln, sada2 - weight 4
 - dp3 - weight 5
 - dph - weight 6
 - dp4 - weight 7
 - dp4a - weight 8
- Extended math operations:
 - math.inv, math.log, math.exp, math.sqrt, math.rsq, math.sin, math.cos (weight 4)
 - math.fdiv, math.pow (weight 8)

NOTE

The type of an operation is determined by the type of a destination operand.

View Data

VTune Profiler runs the analysis and opens the data in the **GPU Compute/Media Hotspots** viewpoint providing various platform data in the following windows:

- **Summary** window displays overall and per-engine GPU usage, percentage of time the EUs were stalled or idle with potential reasons for this, and the hottest GPU computing tasks.
- **Graphics** window displays CPU and GPU usage data per thread and provides an extended list of GPU hardware metrics that help analyze accesses to different types of GPU memory. For GPU metrics description, hover over the column name in the grid or right-click and select the **What's This Column?** context menu option.

Support for SYCL* Applications using oneAPI Level Zero API

This section describes support in the GPU Compute/Media Hotspots analysis for SYCL applications that run OpenCL or [oneAPI Level Zero API](#) in the back end. VTune Profiler supports version 0.91.10 of the oneAPI Level Zero API.

Support Aspect	SYCL application with OpenCL as back end	SYCL application with Level Zero as back end
Operating System	Linux OS Windows OS	Linux OS Windows OS
Data collection	VTune Profiler collects and shows GPU computing tasks and the GPU computing queue.	VTune Profiler collects and shows GPU computing tasks and the GPU computing queue.
Data display	VTune Profiler maps the collected GPU HW metrics to specific kernels and displays them on a diagram.	VTune Profiler maps the collected GPU HW metrics to specific kernels and displays them on a diagram.
Display Host side API calls	Yes	Yes
Source Assembler for computing tasks	Yes	Yes
Instrumentation for GPU code (Source Analysis option or Dynamic Instruction Count characterization option)	Yes	Yes

NOTE

For a use case on profiling a SYCL application running on an Intel GPU, see [Profiling a SYCL App Running on a GPU](#) in the Intel® VTune Profiler Performance Analysis Cookbook .

Support for DirectX Applications

This section describes support available in the GPU analysis to trace Microsoft® DirectX* applications running on the CPU host. This support is available in the **Launch Application** mode only.

Support Aspect	DirectX Application
Operating system	Windows OS
API version	DXGI, Direct3D 11, Direct3D 12, Direct3D 11 on 12
Display host side API calls	Yes
Direct Machine Learning (DirectML) API	Yes
Device side computing tasks	No
Source Assembler for computing tasks	No

See Also

[Optimize applications for Intel GPUs with Intel VTune Profiler](#)
[Offload and Optimize OpenMP* Applications with Intel Tools](#)
[Run Roofline Insights Perspective with Intel Advisor](#)
[GPU Architecture Terminology for Intel Xe Graphics](#)
[Optimize Your GPU Application with Intel oneAPI Base Toolkit](#)
[GPU Compute/Media Hotspots View](#)

[EU Array Stalled/Idle](#)

[Set Up System for GPU Analysis](#)

[Rebuild and Install the Kernel for GPU Analysis](#)

[Intel® Media SDK Program Analysis](#)

[GPU OpenCL™ Application Analysis](#)

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

[Problem: No GPU Utilization Data Is Collected](#)

GPU Compute/Media Hotspots View

Use the GPU Compute/Media Hotspots viewpoint in Intel® VTune™ Profiler to analyze how your GPU-bound code is utilizing GPU and CPU resources.

Depending on the profiling mode selected for the GPU Compute/Media Hotspots analysis, you can explore your GPU-side code performance from different perspectives:

- Run the analysis in **Characterization mode** to see performance issues for the code offloaded to the GPU:
 - [Analyze Memory Accesses and XVE Pipeline Utilization](#) using GPU hardware events.
 - [Analyze GPU Instruction Execution](#)
- Run the analysis in [Source Analysis](#) mode to find the most expensive operations and explore instruction execution:
 - Example: Basic Block Latency Profiling
 - Example: Memory Latency Profiling
- [Analyze Xe Vector Engine \(XVE\) Stalls](#)
- [Examine Energy Consumption by your GPU](#)

Analyze Memory Accesses and XVE Pipeline Utilization

Use the **Characterization** mode to start analyzing GPU-bound applications. This mode is enabled by default in the configuration of the GPU Compute/Media Hotspots analysis.

In the **Summary** window, the **Hottest GPU Computing Task** section displays the most time-consuming GPU tasks.

GPU Compute/Media Hotspots (preview) ⓘ ⓘ

Analysis Configuration Collection Log Summary Graphics

Collapsed Elapsed Time ⓘ: 24.995s

If your application target was run more than once during the collection, this value includes elapsed time for all the runs.

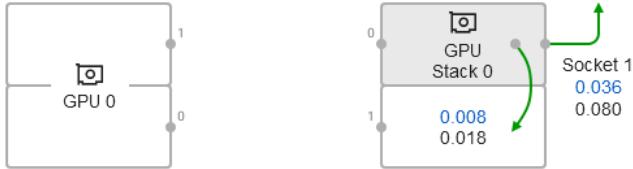
GPU Time ⓘ: 4.156s

Collapsed GPU Topology Diagram

Use this topology diagram to examine the GPU interconnect (Xe Link) and identify stack-stack, GPU-socket, and GPU-GPU bandwidths. Hover over a GPU stack to see bandwidth metrics.

Links Utilization:
— Actively utilized
— Not utilized

Communication Bandwidth:
 Incoming, GB/s
 Outgoing, GB/s



Collapsed Display controller: Intel Corporation Device 0x0bd6 Device Group

Collapsed XVE Array Stalled/Idle ⓘ: 12.1% ⓘ of Elapsed time with GPU busy

Collapsed This section shows the XVE metrics per stack and per adapter for all the devices in this group.

GPU Stack	GPU Adapter	XVE Array Active ⓘ	XVE Array Stalled ⓘ	XVE Array Idle ⓘ
0	GPU 0	8.7%	1.1%	90.2% ⓘ
1	GPU 0	8.7%	1.1%	90.2% ⓘ
0	GPU 1	8.7%	1.1%	90.2% ⓘ
1	GPU 1	8.7%	1.1%	90.3% ⓘ

*N/A is applied to non-summable metrics.

GPU L3 Bandwidth Bound ⓘ: 0.0% of peak value

Collapsed Occupancy ⓘ: 98.7% of peak value

Collapsed This section shows the computing tasks with low occupancy metric for all the devices in this group.

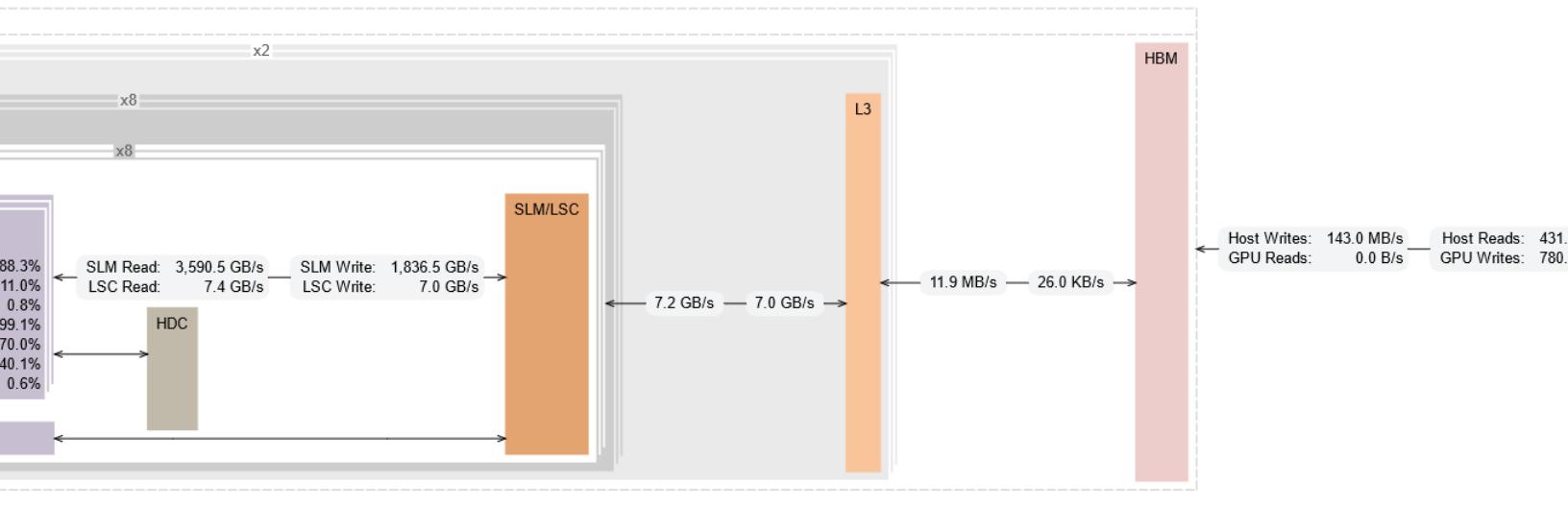
Computing Task	Total Time ⓘ	Occupancy ⓘ	SIMD Utilization ⓘ
slm_latency	2.087s	49.6% ⓘ	0.0%
slm_latency	2.062s	49.6% ⓘ	0.0%

*N/A is applied to non-summable metrics.

Click on one of these tasks and see detailed information in the **Graphics** tab. Learn more about GPU hardware metrics that were collected for the hotspot. By default, this is the **Overview** set of metrics.

The following figure is an example of a Full Compute analysis when you run GPU analysis on Intel® Data Center GPU Max Series (codenamed Ponte Vecchio). Your own results may vary, depending on the configuration of your analysis and choice of hardware.

iew) ®
Graphics



	XVE Threads Occupancy	Computing Threads Started	XVE Instructions								L3 Read Bandwidth, GB/sec	L3 Write Bandwidth, GB/sec
			ALU0 and ALU1 active	ALU0 and ALU2 active	Send active	Branch active	ALU0 active	ALU1 active	ALU2 active			
9.8%	9.8%	7,168,000	2.4%	0.0%	1.5%	0.0%	6.8%	3.9%	0.1%		0.748	0.000
9.8%	9.8%	3,584,000	2.4%	0.0%	1.5%	0.0%	6.8%	3.9%	0.1%		0.719	0.000
99.1%	99.1%	3,584,000	25.0%	0.3%	15.8%	0.0%	70.0%	40.1%	0.6%		7.220	0.000
0.0%	0.0%	0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.000	0.000
0.0%	0.0%	0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		18.212	0.000
0.0%	0.0%	0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.003	0.000
9.8%	9.8%	3,584,000	2.4%	0.0%	1.5%	0.0%	6.8%	3.9%	0.1%		0.776	0.000
9.8%	9.8%	7,168,000	2.4%	0.0%	1.5%	0.0%	6.8%	3.9%	0.1%		0.751	0.000

The **Memory Hierarchy Diagram** shows the GPU hardware metrics that are mapped for a task you select in the table below it. The diagram updates dynamically to reflect the metrics of the current selection in the table.

Display Options for Memory Hierarchy Diagram

Right click on the **Memory Hierarchy Diagram** and open **Show Data As** to change the display :

- The default view is **Bandwidth**. This shows the memory bandwidth.
- The **Total Size** view is useful when you know the amount of data transfer that was supposed to happen through the compute task. Significantly large numbers indicate inefficiency.
- In the **Percentage of Bandwidth Maximum Value** view, see if the kernel was limited by maximum bandwidth on any of the links.

Platform Tab

The **Platform** tab displays a view over time of:

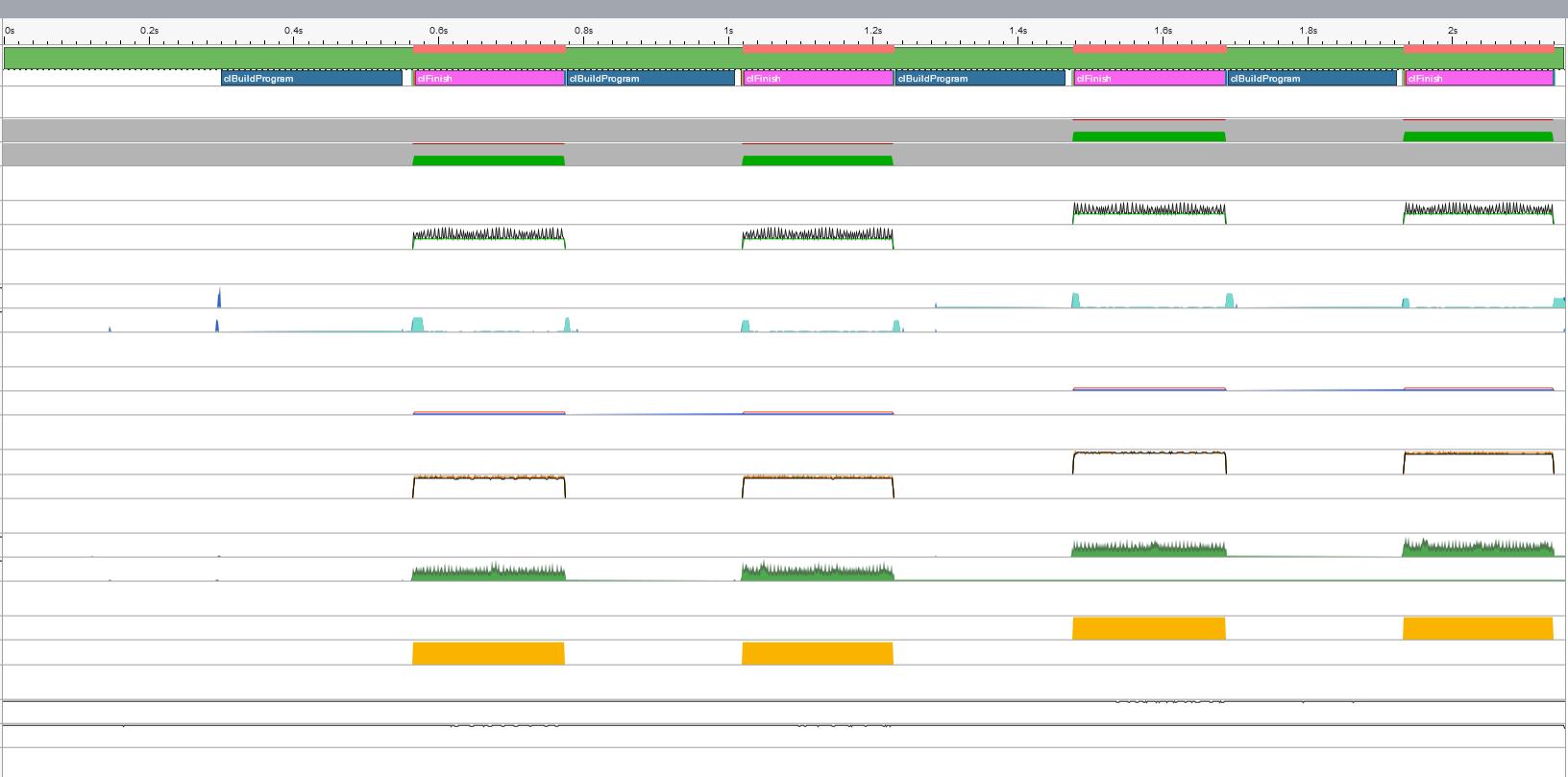
- CPU threads
- Compute runtime API called by the application

- GPU compute tasks
- GPU hardware metrics

Use this visual display to identify irregularities.

Hotspots (preview) ⓘ

Ion Log Summary Graphics

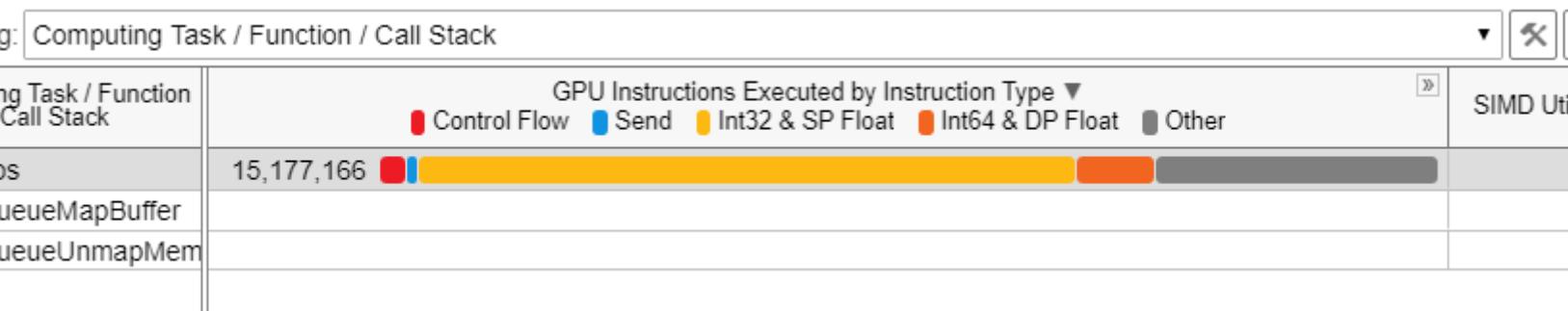


Stack / Computing Task

Work Size	Computing Task					Data Transferred			XVE Array			XVE Threads Occupancy	Computing Threads Started		
	Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	Size	Total, GB/sec	Active	Stalled	Idle				ALU0 and ALU1 active	ALU0 and ALU1 idle
76 1024	417.630ms	2.047ms	204			58.7 MB	0.014	8.4%	1.0%	90.5%	9.5%	1,433,600	2.4%		
	209.739ms	2.056ms	102			29.4 MB	0.014	8.4%	1.0%	90.5%	9.5%			716,800	2.4%
	208.576ms	2.086ms	100	32		0 B	0.000	88.3%	10.9%	0.8%	99.1%			716,800	25.0%
	0.491ms	0.491ms	1			14.7 MB	29.886	0.0%	0.0%	100.0%	0.0%			0	0.0%
	0.671ms	0.671ms	1			14.7 MB	21.866	0.0%	0.0%	100.0%	0.0%			0	0.0%
	0ms	0ms	0			0 B	0.000	0.0%	0.0%	100.0%	0.0%			0	0.0%
	207.891ms	2.038ms	102			29.4 MB	0.014	8.4%	1.0%	90.5%	9.4%			716,800	2.4%
	417.816ms	2.048ms	204			58.7 MB	0.014	8.4%	1.1%	90.5%	9.5%			1,433,600	2.4%

Analyze GPU Instruction Execution

If you enabled the **Dynamic Instruction Count** preset as part of the Characterization analysis configuration, the **Graphics** tab shows a breakdown of instructions executed by the kernel in the following groups:



GPU Instructions Executed by Instruction Type ▾	
Control Flow	Send
Int32 & SP Float	Int64 & DP Float
Other	
15,177,166	

Control Flow group if, else, endif, while, break, cont, call, calla, ret, goto, jmp, brd, brc, join, halt and mov, add instructions that explicitly change the ip register.

Send group send, sends, sendc, sendsc

Synchronization group wait

Int16 & HP Float | Int32 & SP Float | Int64 & DP Float groups

- Bit operations (only for integer types): and, or, xor, and others.
- Arithmetic operations: mul, sub, and others; avg, frc, mac, mach, mad, madm.
- Vector arithmetic operations: line, dp2, dp4, and others.
- Extended math operations: math.sin, math.cos, math.sqrt, and others.

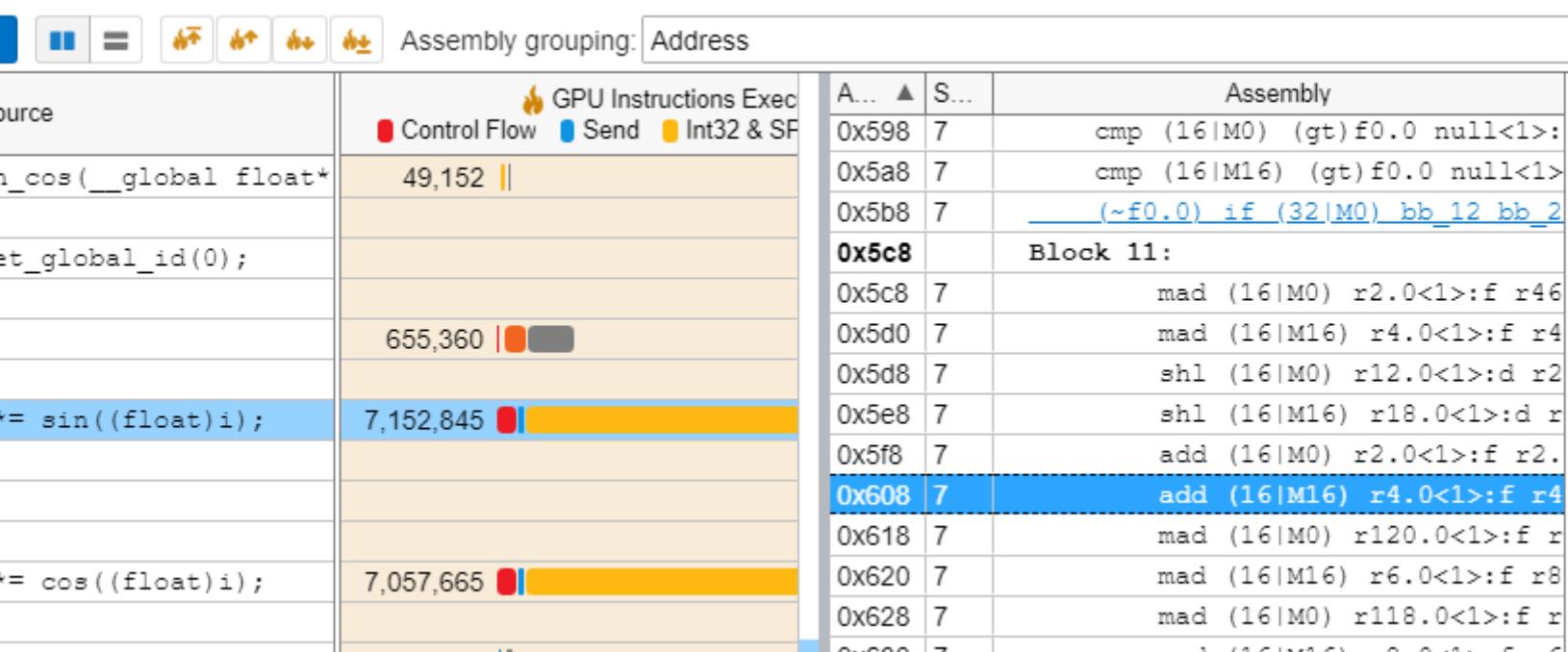
Other group Contains all other operations including nop.

NOTE

The type of an operation is determined by the type of a destination operand.

In the **Graphics** tab, VTune Profiler also provides the SIMD Utilization metric. This metric helps identify kernels that underutilize the GPU by producing instructions that cause thread divergence. A common cause of low SIMD utilization is conditional branching within the kernel, since the threads execute all of the execution paths sequentially, with each thread executing one path while the other threads are stalled.

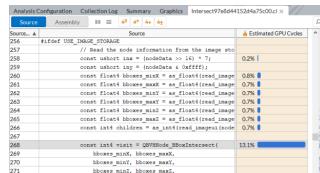
To get additional information, double-click the hottest function to open the source view. Enable both the **Source** and **Assembly** panes to get a side-by-side view of the source code and the resulting assembly code. You can then locate the assembly instructions with low SIMD Utilization values and map them to specific lines of code by clicking on the instruction. This allows you to determine and optimize the kernels that do not meet your desired SIMD Utilization criteria.



NOTE For information on the Instruction Set Architecture (ISA) of Intel® Iris® Xe MAX Graphics, see the [Intel® Iris® Xe MAX Graphics Open Source Programmer's Reference Manual](#).

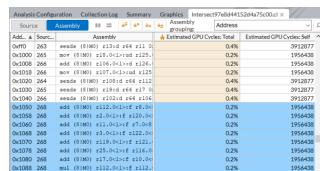
Analyze Source

In the **Source Analysis** mode for the GPU Compute/Media Hotspots analysis, you can analyze a kernel of interest for basic block latency or memory latency issues. To do this, in the **Graphics** tab, expand the kernel node and double-click the function name. VTune Profiler redirects you to the hottest source line for the selected function:



The GPU Compute/Media Hotspots analysis provides a full-scale analysis of the kernel source per code line. The hottest kernel code line is highlighted by default.

To view the performance statistics on GPU instructions executed per kernel instance, switch to the **Assembly view**:



NOTE

If your OpenCL kernel uses inline functions, make sure to enable the **Inline Mode** on the filter toolbar to have a correct attribution of the GPU Cycles per function. [See examples.](#)

Example: Basic Block Latency Profiling

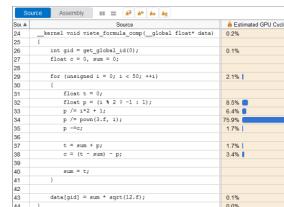
You have an OpenCL kernel that performs compute operations:

```
__kernel void viete_formula_comp(__global float* data)
{
    int gid = get_global_id(0);
    float c = 0, sum = 0;

    for (unsigned i = 0; i < 50; ++i)
    {
        float t = 0;
        float p = (i % 2 ? -1 : 1);
        p /= i*2 + 1;
        p /= pown(3.f, i);
        p -=c;

        t = sum + p;
        c = (t - sum) - p;
        sum = t;
    }
    data[gid] = sum * sqrt(12.f);
}
```

To compare these operations, run the GPU In-kernel profiling in the **Basic block latency** mode and double-click the kernel in the grid to open the Source view:



The Source view analysis highlights the `pown()` call as the most expensive operation in this kernel.

Example: Memory Latency Profiling

You have an OpenCL kernel that performs several memory reads (lines 14, 15 and 20):

```
__kernel void viete_formula_mem(__global float* data)
{
    int gid = get_global_id(0);
    float c = 0;

    for (unsigned i = 0; i < 50; ++i)
    {
        float t = 0;
        float p = (i % 2 ? -1 : 1);
        p /= i*2 + 1;
        p /= pown(3.f, i);
        p -=c;

        t = data[gid] + p;
    }
}
```

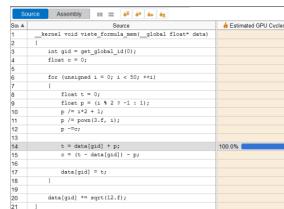
```

    c = (t - data[gid]) - p;

    data[gid] = t;
}
data[gid] *= sqrt(12.f);
}

```

To identify which read instruction takes the longest time, run the GPU In-kernel Profiling in the **Memory latency** mode:



The Source view analysis shows that the compiler understands that each thread works only with its own element from the input buffer and generates the code that performs the read only once. The value from the input buffer is stored in the registry and reused in other operations, so the compiler does not generate additional reads.

Analyze XVE Stalls

Whenever your GPU is not fully employed, use available guidance for Xe Vector Engines (XVEs) to understand reasons for stalled or idle behavior in some of your GPU stacks.

NOTE The inclusion of XVE stall reasons in the results of the GPU Compute/Media Hotspots analysis happens only for Intel® Data Center GPU Max Series devices (code named Ponte Vecchio).

First, run the GPU Compute/Media Hotspots analysis in **Characterization** mode. Select the **Overview** option in this mode.

Once the analysis completes, check the **Summary** tab of the viewpoint. The **XVE Array Stalled/Idle** table displays a list of those Xe Vector Engines (XVEs) that remained in the stalled (received computing tasks but did not execute them) or idle (never received computing tasks) states when the profiling happened. Both of these cases present areas for improvement and better use of available hardware.

For a GPU stack with high values of the XVE Array Stalled metric, your next step is to run the GPU Compute/Media Hotspots analysis in the **Source Analysis** mode.

In the **Source Analysis** mode, select the **Stall Reasons** option before you repeat the analysis.

When the analysis completes, see the **Hottest GPU Computing Tasks** table. Here, you can find a list of the busiest functions sorted in order by Total Time.

Welcome x Configure Analysis x r008gh x r006gh x

GPU Compute/Media Hotspots (preview)

Analysis Configuration Collection Log Summary Graphics

Elapsed Time: 4.072s

Hottest GPU Computing Tasks

This section lists the most active computing tasks running on the GPU, sorted by the Total Time.

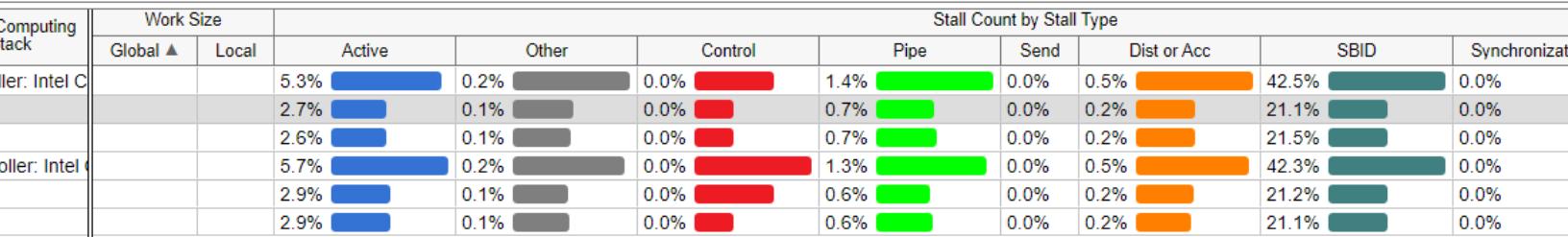
Computing Task	Total Time	Average Time	Instance Count	Control	Dist or Acc	Instruction Fetch	Pipe	SBID	Send	Synchronization	Other
Matrix1<float>	0.015s	0.004s	4	0.0%	0.2%	0.0%	0.5%	2.7%	0.0%	0.0%	0.1%
Matrix1<float>	0.014s	0.004s	4	0.0%	0.4%	0.0%	1.0%	5.6%	0.0%	0.0%	0.2%

*N/A is applied to non-summable metrics.

For computing tasks that caused an XVE stall, you can learn about the various reasons that contributed to the stall.

Click on a function and switch to the **Graphics** view to see a distribution of stall counts. Make sure to select the Computing Task/Function grouping in the **Graphics** view to locate the stalls for each computing task.

GPU Stack / Computing Task / Function / Call Stack



Reasons for XVE Stalls

There may be several possible reasons for a stall in your Xe Vector Engine (XVE). The following table describes these reasons:

Reason for XVE Stall	Explanation
Active	At least one instruction is dispatching into a pipeline.
Control	The percentage of stalls when the instruction was waiting for a Branch unit to become available.
Dist or Acc	The percentage of stalls when the instruction was waiting for a Distance or Architecture Register File (ARF) dependency to resolve.
Instruction Fetch	The percentage of stalls when the XVE was waiting for an instruction to be returned from the instruction cache.
Pipe	The percentage of stalls when the instruction won arbitration but could not be dispatched into a Floating-Point or Extended Math unit. This can occur due to a bank conflict with the General Registry File (GRF).
SBID	The percentage of stalls when the instruction was waiting for a Software Scoreboard dependency to resolve.
Send	The percentage of stalls when the instruction was waiting for a Send unit to become available.

Reason for XVE Stall	Explanation
Synchronization	The percentage of stalls when the instruction was waiting for a thread synchronization dependency to resolve.
Other	The percentage of stalls when other factors stalled the execution of the instruction.

Examine Energy Consumption by your GPU

In Linux environments, when you run the GPU Compute/Media Hotspots analysis on an Intel® Iris® Xe MAX graphics discrete GPU, you can see energy consumption information for the GPU device. To collect this information, make sure you check the **Analyze power usage** option when you configure the analysis.

The screenshot shows the 'GPU Compute/Media Hotspots (preview)' interface. At the top, there's a 'Characterization' section with a radio button and an 'Overview' dropdown. Below it is a 'GPU sampling interval, ms' input field set to '1'. There are two checkboxes: 'Analyze memory bandwidth' (unchecked) and 'Trace GPU programming APIs' (checked). A bar chart titled 'Overhead' is visible on the right. Below these, there's a 'Source Analysis' section with a radio button and a dropdown menu showing 'Instance step' and 'Default'. Under 'Source Analysis', there's a note about enabling energy consumption characterization and a checked checkbox for 'Analyze power usage', which is highlighted with a yellow box. At the bottom, there's a 'Details' button.

Once the analysis completes, see energy consumption data in these sections of your results.

In the **Graphics** window, observe the **Energy Consumption** column in the grid when grouped by **Computing Task**. Sort this column to identify the GPU kernels that consumed the most energy. You can also see this information mapped in the timeline.

Tune for Power Usage

When you locate individual GPU kernels that consume the most energy, for optimum power efficiency, start by tuning the top energy hotspot.

Tune for Processing Time

If your goal is to optimize GPU processing time, keep a check on energy consumption metrics per kernel to monitor the tradeoff between performance time and power use.

Move the **Energy Consumption** column next to **Total Time** to make this comparison easier.

Computing Task	Energy Consumption (mJ) ▼	Computing Task					Work Size	
		Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	Global	Local
► clEnqueueReadImage	419829.895	48.966s	0.816s	60				
► clEnqueueWriteBuffer	47357.239	4.841s	0.004s	1,131				
► clEnqueueWriteImage	38314.819	3.731s	0.041s	90				
► particle	36702.637	1.807s	0.000s	4,004	32		262144	64
► clEnqueueReadBuffer	22355.591	2.125s	0.003s	812				
► stereo_compute_disparity	14262.390	0.646s	0.004s	183	32		464 x 384	16 x 16
► convolve_1d_vertical_grayscale	8891.663	0.531s	0.002s	320	32		656 x 856	16 x 8
► convolve_1d_horizontal_grayscale	8879.944	0.533s	0.002s	320	32		704 x 852	64 x 4
► detect	6203.857	0.407s	0.020s	20	16		167136	
► convolve_1d_vertical	4898.315	0.230s	0.011s	20	32		5960 x 4096	4 x 64
► knn_match	4764.221	0.302s	0.015s	20	16		4096	64
► compute_keypoints	4567.261	0.300s	0.001s	320	32		592 x 800	8 x 16
► convolve_1d_horizontal	4056.946	0.229s	0.011s	20	32		5960 x 4096	4 x 64
► hough_get_scores	3087.097	0.207s	0.010s	20	32		2624 x 3584	64 x 2
► blur	2429.260	0.119s	0.006s	20	16		5984 x 4096	32 x 2
► fast_corners	2420.593	0.135s	0.000s	320	16		592 x 792	16 x 8
► non_maximum_suppression	2346.069	0.107s	0.005s	20	32		5968 x 4096	16 x 16
► hysteresis	2315.308	0.145s	0.007s	20	32		23550464	64
► calculate_descriptors	1897.156	0.124s	0.000s	323	8		4032	64
► fft	1557.983	0.103s	0.026s	4	16		4194304	32

You may notice that the correlation between power use and processing time is not direct. The kernels that compute the fastest may not be the same kernels that consume the least amounts of energy. Check to see if larger values of power usage correspond to longer stalls/wait periods.

NOTE Energy consumption metrics do not display in GPU profiling analyses that scan Intel® Iris® X e MAX graphics on Windows machines.

See Also

[Hotspots Report](#)
from command line

[View Data on Inline Functions](#)

[Offload and Optimize OpenMP* Applications with Intel Tools](#)

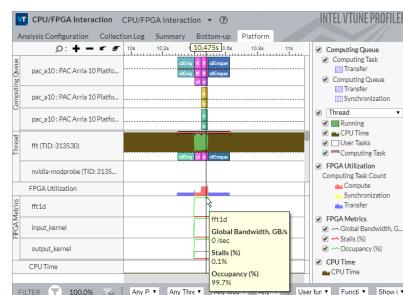
[Optimize applications for Intel® GPUs with Intel® VTune Profiler](#)

[Optimize Your GPU Application with Intel oneAPI Base Toolkit](#)

CPU/FPGA Interaction Analysis

Use the CPU/FPGA Interaction analysis to assess the balance between CPU and FPGA in systems with FPGA hardware that run SYCL or OpenCL™ applications.

Use the CPU/FPGA Interaction analysis to assess FPGA performance of executed kernels, overall time for memory transfers between the CPU and FPGA, and wait time impact on CPU and FPGA workloads.



Intel® VTune™ Profiler collects these FPGA device metrics:

- Global Bandwidth
- Stalls
- Occupancy
- Activity
- Idle

Configure and Run Analysis

Follow this procedure to configure options for the CPU/FPGA Interaction analysis:

Prerequisites:

- To obtain device side information from the FPGA when profiling, make sure you specify the profile flag for the compile operation:

To compile	Use	Specify
OpenCL Applications	Intel® FPGA SDK for OpenCL™ Offline Compiler	-profile option
SYCL Applications	Intel® oneAPI DPC++/C++ Compiler	-Xsprofile option

For other compiler options (exclusive to OpenCL profiling), see the [FPGA Programming Guide](#).

- [Create a VTune Profiler project](#).

1. Click the



(standalone GUI)/

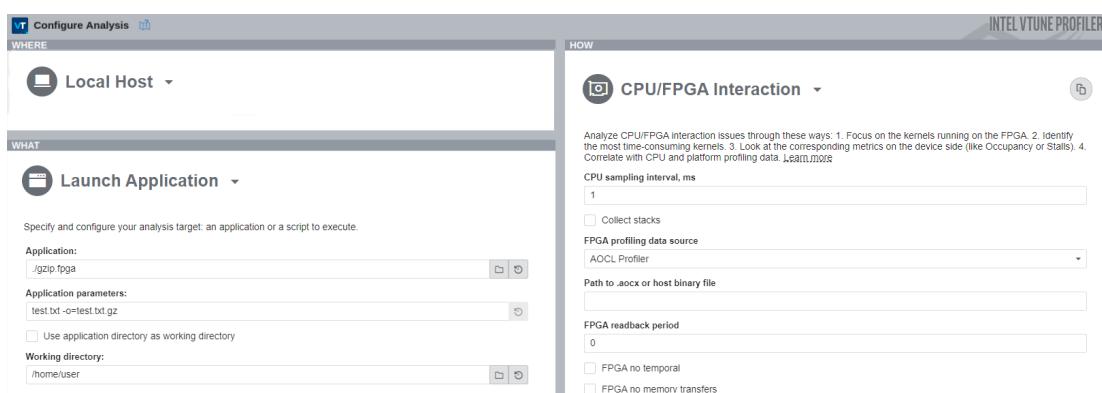


(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. In the **WHAT** pane,

- Specify the host executable in the **Application** bar.
- If applicable, specify arguments for the host application as **Application parameters**.



3. In the **HOW** pane, click the



Browse button.

- Select **CPU/FPGA Interaction** analysis type from the **Accelerators** group.
- Enter the CPU sampling interval in milliseconds.
- Specify if the collection should include CPU call stacks.

- Specify a source for the FPGA profiling data:
 - **OpenCL Profiling API** - This source profiles only the host application.
 - **AOCL Profiler** - This source profiles the host application as well as the design on your FPGA.

NOTE

To [generate the command line](#) for this configuration, use the



Command Line button.

4. Click the



Start button to [run the analysis](#).

[Import FPGA Data collected with Profiler Runtime Wrapper](#)

If you collected FPGA profiling data with the Profiler Runtime Wrapper in the format of a `profile.json` file, you can also [import](#) it to the VTune Profiler project.

To speed up the loading of the collected data, copy the `profile.json` to an empty folder and import that folder instead of the entire compilation directory.

See the [FPGA Optimization Guide](#) for information on generating the profiling data with the Profiler Runtime Wrapper (oneAPI applications only).

[View Data](#)

The CPU/FPGA Interaction analysis results appear in the CPU/FPGA Interaction viewpoint. The viewpoint contains these windows:

- The [Summary window](#) displays statistics on the overall application execution, identifying CPU time and processor utilization, and execution time for SYCL or OpenCL kernels. Double click a kernel in the Bottom-up view to see detailed performance data through the Source view.
- The [Bottom-up window](#) displays functions in the Bottom-up tree, CPU time and CPU utilization per function. Click the functions or kernels in this view to see the Source view.
- The [Platform window](#) displays over-time metric and performance data for SYCL or OpenCL kernels, memory transfers, CPU context switches, FPU utilization, and CPU threads with SYCL or OpenCL kernels.

[What's Next](#)

Use the CPU/FPGA Interaction viewpoint to review the following:

- **FPGA Utilization:** Look at the **FPGA Top Compute Tasks** on the **Summary** window for a list of kernels running on the FPGA. The **Bottom-up** window shows the Total and Average execution time for every kernel.
- **Memory Transfers:** Look at the **Data Transferred** column on the **Bottom-up** window or the **Computing Queue** rows on the **Platform** window to view SYCL or OpenCL kernels and memory transfers.
- **Workload Impact:** The **Context Switch Time** metric on the **Summary** window shows how much time was spent in CPU context switches. Context switches can also be seen on the **Platform** tab as they occurred during application execution.

[See Also](#)

[fpga-interaction Command Line Analysis](#)

[CPU/FPGA Interaction View](#)

Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide

CPU/FPGA Interaction View

Use the CPU/FPGA Interaction viewpoint to assess FPGA performance of executed kernels, overall time for memory transfers between the CPU and FPGA, and how well a workload is balanced between the CPU and FPGA.

To interpret the performance data provided in the CPU/FPGA Interaction viewpoint, you may follow the steps below:

1. Define a Performance Baseline
2. Assess FPGA Utilization
3. Review Memory Transfers
4. Determine Workload Impact
5. Review FPGA device metrics
6. Analyze channel depth
7. Analyze loops
8. Analyze Source of the host application part
9. Analyze Source of the kernel running on FPGA device

Define a Performance Baseline

Start with exploring the **Summary** window that provides general information on your application execution. Key areas for optimization include application execution time, tasks with high CPU or FPGA time, and kernel execution time.

Use the Elapsed Time value as a baseline for comparison of versions before and after optimization.

Assess FPGA Utilization

Look at the **FPGA Top Compute Tasks** list on the **Summary** window for a list of kernels running on the FPGA.

FPGA Top Compute Tasks

This section lists the most active FPGA compute tasks in your application.

Computing Task (FPGA)	Computing Task Time	Computing Task Count [®]
multi_compute	2000.000s	2
mem_writestream	0.029s	5
nop	0.021s	1
autorun_k1	0.000s	2
output_kernel	0.000s	2
[Others]	0.000s	4

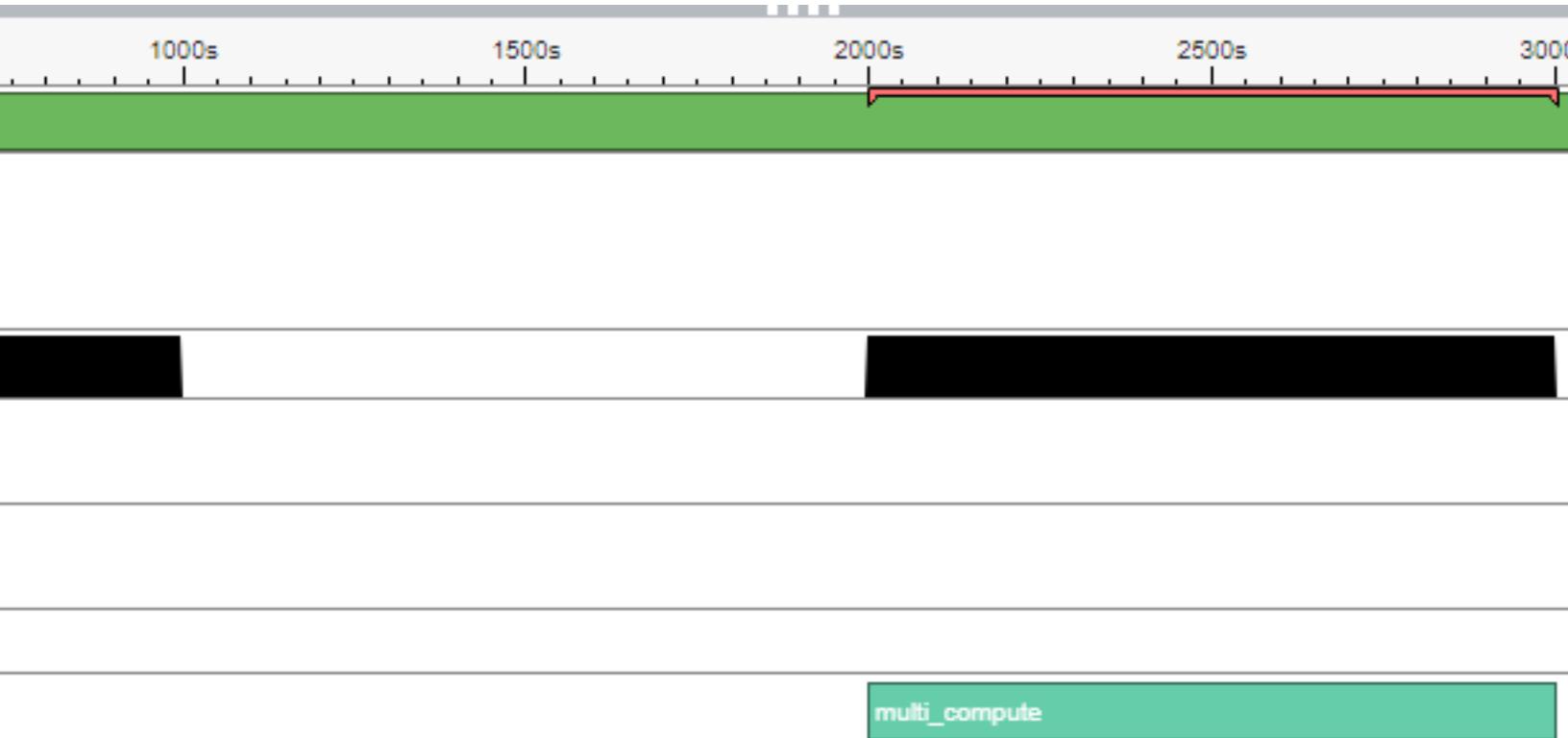
*N/A is applied to non-summable metrics.

Switch to the **Bottom-up** window and use the **Computing Task Purpose / Source Computing Task (FPGA)** grouping to view the hotspots for kernels.

Tip

You can click a task from the **FPGA Top Compute Tasks** list to be taken to that task on the **Bottom-up** window.

Review the **FPGA Utilization** timeline, which shows how many kernels and transfers are executing at the same time on the FPGA.



Review Memory Transfers

Look at the **Data Transferred** column on the **Bottom-up** window or the **Computing Queue** rows on the **Platform** window to view the FPGA kernels and memory transfers.

Determine Workload Impact

The **Context Switch Time** metric on the **Summary** window shows the amount of time the CPU spent in context switches. Switch to the **Platform** window and hover over the timeline to view the reason for the context switch. In some cases, CPU context switches may represent CPU waits for the FPGA. Look at the **FPGA Utilization** line to identify times when the CPU may have been waiting on the FPGA and vice versa. For instance, when there is no FPGA activity, but CPU activity is high, it is likely that the FPGA is waiting for the CPU to complete a preparation step.

Review FPGA Device Metrics

Switch to the **Bottom-up** window to analyze **Stalls**, **Global Bandwidth** and **Occupancy** metrics and see how efficiently your kernels run on the FPGA device.

Analyze the **Idle %** metrics values to understand the percentage of cycles when there were no valid work-items executing or stalling the memory or channel instruction. The **Activity %** metric shows the percentage of cycles a predicated channel or memory instruction is enabled.

Analyze Channel Depth

In the Bottom-up window, locate the **Average** and **Maximum Channel Depth** information for selected instances. If required, adjust the channel depth for your needs.

Task		Device Metrics					C	
Index	Instance Count	Stalls (%)	Occupancy (%)	Idle (%)	Data Transferred, Global			Cycles
					Size	Average Bandwidth, G...		
0s	2	26.2%	23.8%	0.0%	0 B	0.000	1	
0s		52.3%	23.8%	0.0%	0 B	0.000	1	
0s		0.0%	23.8%	0.0%	0 B	0.000	1	
0s	2	42.6%	28.7%	0.0%	0 B	0.000	1	
0s		42.6%	28.7%	0.0%	0 B	0.000	1	
0s		42.6%	28.7%	0.0%	0 B	0.000	1	
0s	3	0.0%	50.0%	0.0%	0 B	0.000	1	

If the channel is full all the time, the write side of the channel is working faster than the read side, and the channel will be stalling in the write kernel. If the channel is mostly empty, the read side is likely to be stalling, and if the channel is bigger than 32 bits deep, you can reduce it in size without a performance hit.

Analyze Loops

Analyze the occupancy for profiled loops:

Summary **Bottom-up** Platform

Compute Unit

Computing Task			Device Metrics				
	Average Time	Instance Count	Stalls (%)	Occupancy (%)	Idle (%)	Activity (%)	Size
52s	1.152s	1	0.0%	40.0%	20.0%	0.0%	6
	0s		0.0%	0.0%	100.0%	0.0%	6
	0s		0.0%	0.0%	0.0%	0.0%	
	0s		0.0%	75.0%	0.0%	0.0%	
	0s		0.0%	25.2%	0.0%	0.0%	
	0s		0.0%	100.0%	0.0%	0.0%	

Analyze Source of the Host Application Part

Double-click the function you want to optimize to view its related source code file in the Source/Assembly window. You can open the code editor directly from the Intel® VTune™ Profiler and edit your code (for example, minimizing the number of calls to the hotspot function).

Analyze Source of the Kernel Running on an FPGA Device

Double-click the kernel to see FPGA device metrics per the kernel source lines. Use the Source view to see what channels and memories cause most stalls and how much data they transfer.

See Also

[Analyze Performance](#)

[Source Code Analysis](#)

[Reference](#)

[Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide](#)

NPU Exploration Analysis (Preview)

Use the NPU Exploration analysis to profile and optimize artificial intelligence (AI) workloads running on Intel architectures.

A Neural Processing Unit(NPU) can accelerate the performance of AI workloads that have been explicitly offloaded onto it by an operating system. NPUs are uniquely designed to improve the performance of AI and machine-learning(ML) workloads. Use the [Intel® Distribution of OpenVINO™ toolkit](#) to offload popular ML

models (like speech or image recognition tasks) to Intel NPUs. Then use the NPU Exploration analysis to profile AI and ML workloads. Collect performance data and optimize the performance of these AI/ML applications.

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

NPU Data Collection Modes

When you run the NPU Exploration Analysis, Intel® VTune™ Profiler can collect hardware metrics about NPU performance in one of two ways:

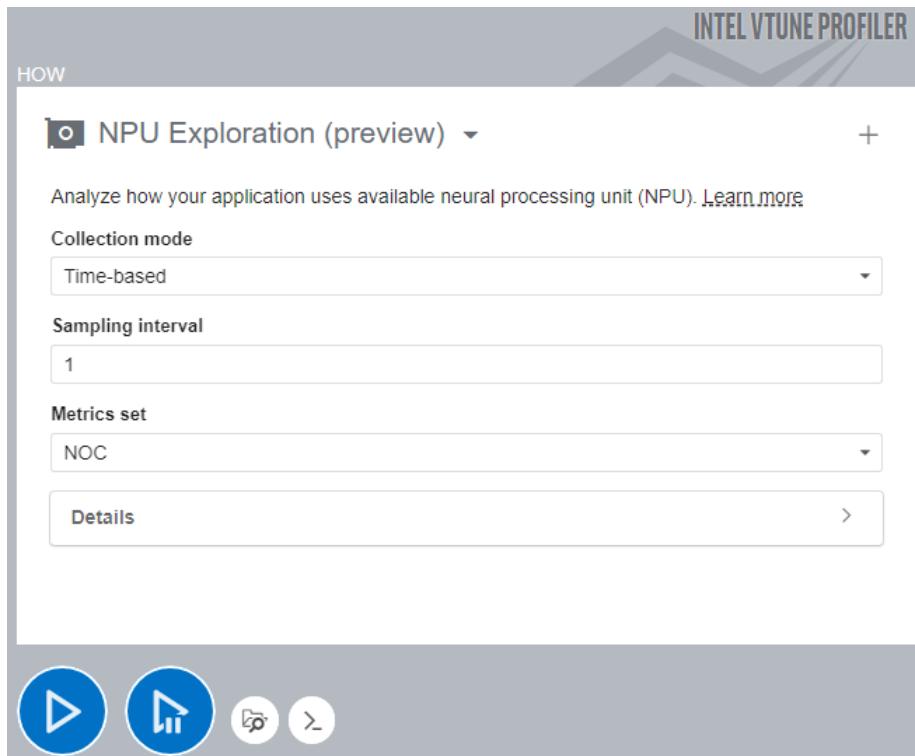
- Time-based mode
- Query-based mode

This table captures the differences between the two collection modes so you can make the best choice for your workload.

	Time-based mode	Query-based mode
How it works	Intel® VTune™ Profiler collects metrics system-wide, similar to CPU uncore metrics.	Intel® VTune™ Profiler collects metrics for each instance of a Level Zero inference. Metrics are collected system-wide but data collection is closely tied to the inference. The collection starts immediately before each inference and stops immediately after.
Size of typical workload	Large	Small
Execution time of instance	>5 ms	<5 ms
Sampling interval	Specify value between 0.1 ms and 1000 ms	N/A
Benefits	Use this mode for larger workloads. Optimize applications with reasonable efficiency and reduced overhead.	Use this mode for smaller workloads. Optimize application more efficiently, even if runtime is longer. Examine effectively if your workload is DDR memory bound.
Usage considerations	Less overhead for application. This mode requires Level Zero backend to be installed, with normal NPU drivers. However, the mode does not require the application to use Level Zero to collect metrics, except for computing tasks.	More overhead for application. This mode requires the application to use Level Zero for the backend.

Configure and Run Analysis

1. In the **Accelerators** group of the Analysis Tree in the VTune Profiler user interface, select **NPU Exploration (preview)**.
2. In the **WHAT** pane, specify the path to the AI/ML application in the **Application** bar.
3. If necessary, specify relevant **Application parameters** as well.
4. In the **HOW** pane, select a **Collection mode**.
5. Specify a sampling interval.



6. Click the



Start button to [run the analysis](#).

Run from Command Line

To run the NPU Exploration analysis from the command line, type:

```
$ vtune -collect npu [-knob <knob_name=knob_option>] -- <target> [target_options]
```

NOTE

To [generate the command line](#) for any analysis configuration, use the **Command Line** button at the bottom of the user interface.

Once VTune Profiler completes data collection, the results of the NPU Exploration analysis appear in the NPU Exploration viewpoint.

See Also

[NPU Exploration View](#)

[npu](#)

NPU Exploration View

Use the NPU Exploration viewpoint to assess and optimize the performance of AI or ML workloads on Intel Neural Processing Units (NPU).

When the NPU Exploration analysis executes, Intel® VTune™ Profiler collects NOC metric set data about the DDR bandwidth between the NPU and DDR memory. Once data collection completes, Intel® VTune™ Profiler prepares the results and displays them in the **Summary** window.

NPU Exploration Summary

The **Summary** window displays NPU performance data starting with these sections:

- **NPU Device Load** - This section indicates the amount of data transferred between the NPU and DDR memory.
- **NPU Top Compute Tasks** - This section captures the total amount of time when tasks got executed on the NPU.

The screenshot shows the Intel VTune Profiler interface with the 'NPU Exploration (preview)' viewpoint selected. The main content area displays the following sections:

- Elapsed Time**: 6.214s
 - Total Thread Count: 44
 - Paused Time: 0s
- NPU Device Load**

Device	NPU DDR Data Transferred
Intel(R) NPU	20.3 GB

*N/A is applied to non-summable metrics.
- NPU Top Compute Tasks**

This section lists the most active NPU compute tasks in your application.

Computing Task (NPU)	Computing Task Time	Computing Task Count
zeAppendGraphExecute	1.817s	999
zeAppendGraphInitialize	0.000s	1

*N/A is applied to non-summable metrics.
- Top Tasks**

This section lists the most active tasks in your application.

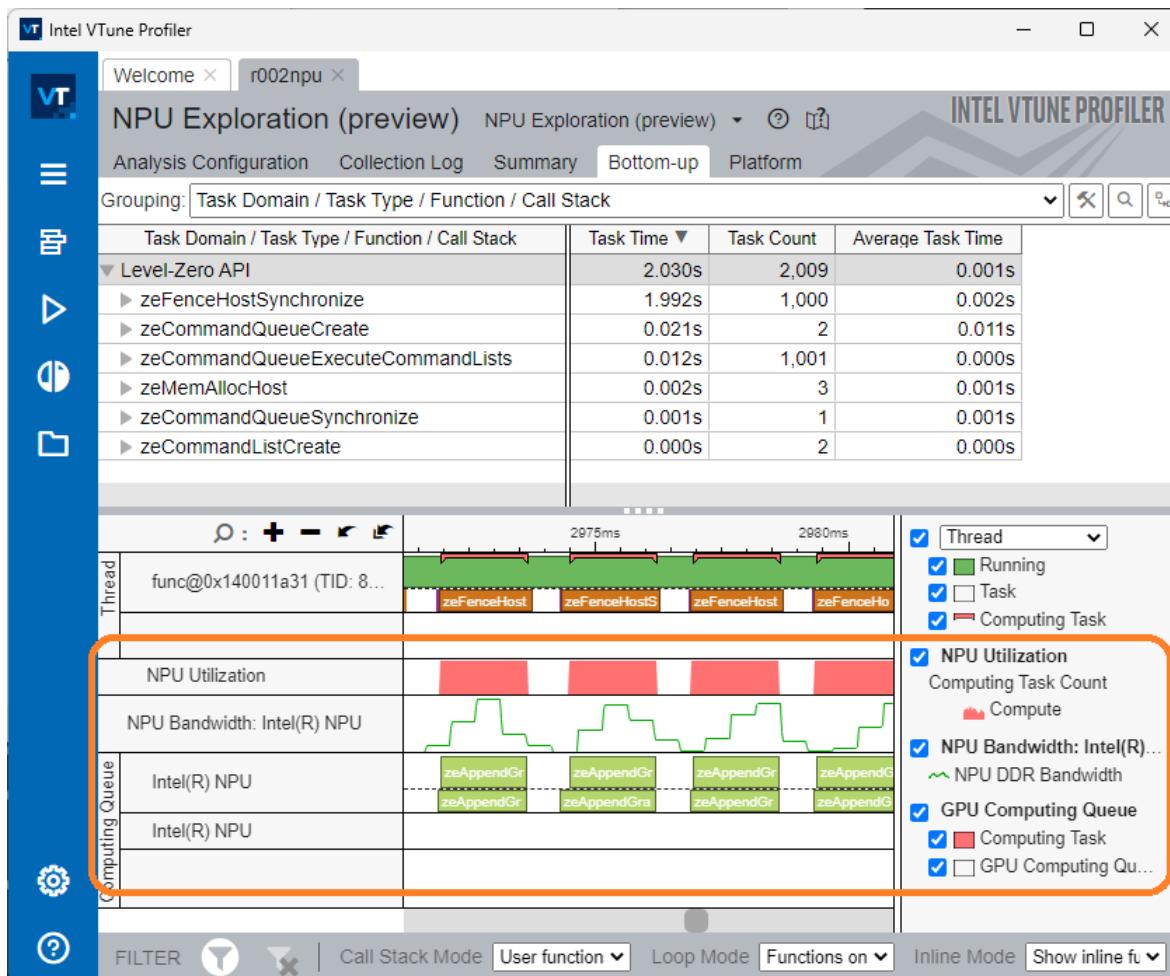
Task Type	Task Time	Task Count	Average Task Time
zeFenceHostSynchronize	1.988s	1,000	0.002s
zeCommandQueueExecuteCommandLists	0.020s	1,001	0.000s
zeCommandQueueCreate	0.009s	2	0.005s
zeMemAllocHost	0.002s	3	0.001s
zeCommandQueueSynchronize	0.001s	1	0.001s
[Others]	0.000s	2	0.000s

*N/A is applied to non-summable metrics.
- Effective CPU Utilization Histogram**
- Collection and Platform Info**

Next, see the list of **Top Tasks** to review the various host tasks which offloaded work onto the NPU.

NPU Exploration Bottom-up Window

Continue your examination of host tasks by switching to the **Bottom-up** window. In the **Grouping** pull down menu, select the **Task Domain / Task Type / Function / Call Stack** grouping.



See the execution of device tasks from the instant they started. This is the instant when the task was appended to the **Computing Queue**.

In the **Computing Queue** section, the portion of the graph above the dotted line indicates duration when the task was executed on the NPU.

The portion of the graph below the dotted line indicates the duration for which the task was waiting in the queue for execution on the NPU. Tasks are removed from the **Computing Queue** when they finish executing on the NPU.

See Also

[Analyze Performance](#)

[Source Code Analysis](#)

[Reference](#)

[Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide](#)

Platform Analysis Group

The **Platform Analysis** group contains the System Overview analysis to monitor system behavior and power usage.

[System Overview](#) analysis is a driverless event-based sampling analysis that monitors the general behavior of your target system. Use this analysis to identify platform-level factors that limit performance, including power usage and throttling.

NOTE Starting with the 2024.0 release of Intel® VTune™ Profiler, the **Platform Profiler** application is available as a separate download. Access this application from the [Intel Registration Center](#).

The **Platform Profiler** application will be discontinued in a future release. As a workaround, consider using the EMON data collector. To learn more, see this [transition article](#).

- To collect platform behavior data using the 2023.2 or newer versions of VTune Profiler, use the standalone Platform Profiler collector. You can then visualize collected data with the Platform Profiler server. See [Platform Analysis](#) for more information.
 - The Platform Profiler analysis type is not available in versions of VTune Profiler newer than 2023.2. To use the Platform Profiler analysis type (from the GUI or command line), switch to a version of VTune Profiler older than 2023.2. You can then follow procedures described in [Platform Profiler Analysis](#).
-

Prerequisites:

- For best results, [install the sampling driver](#) for hardware event-based sampling collection types. For Linux* and Android* targets, if the sampling driver is not installed, VTune Profiler can work on Perf* ([driverless collection](#)).
- To enable system-wide and uncore event collection, use root or sudo to set `/proc/sys/kernel/perf_event_paranoid` to 0.

```
$ echo 0>/proc/sys/kernel/perf_event_paranoid
```

System Overview Analysis

Use a platform-wide System Overview analysis to monitor a general behavior of your target system and identify platform-level factors that limit performance.

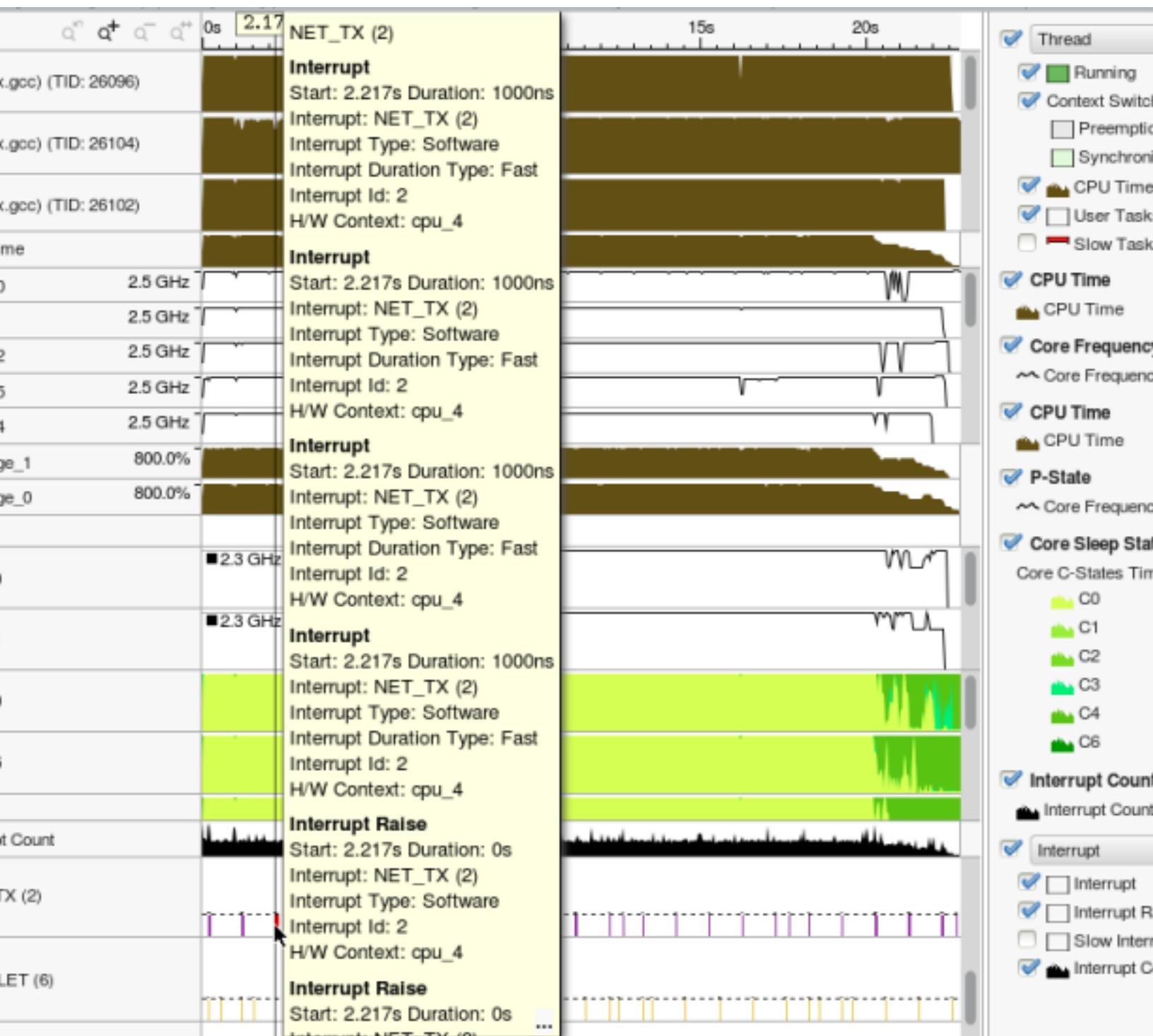
The System Overview analysis supports the following profiling modes:

- [Hardware Event-Based Sampling](#) serves as an entry-point analysis to identify how effectively your code utilizes CPU, GPU, DRAM, I/O, and PCIe.
- [Hardware Tracing](#) (Linux* and Android* targets) analyzes your code at the microsecond level and helps identify a cause of latency issues.

You can also use the System Overview analysis to get power usage data for your system, with a breakdown of power usage by socket and DRAM module.

Hardware Event-Based Sampling Mode

In this mode, you can capture overall CPU, GPU, and I/O resources utilization and see recommendations for next steps. Use this mode as an entry-level analysis to triage system performance issues.



For Linux targets, the System Overview analysis collects the following Ftrace* events: sched, freq, idle, workq, irq, softirq.

For Android targets, the System Overview analysis collects the following events:

- Atrace* events: input, view, webview, audio, video, camera, hal, res, dalvik
- Ftrace events: sched, freq, idle, workq, filesystem, irq, softirq, sync, disk

Hardware Tracing Mode (Linux and Android Targets)

Use this mode to capture CPU core activities at the microsecond level and detect unusual behavior.

Prerequisites:

- To enable system-level analysis for this mode, consider setting the `/proc/sys/kernel/perf_event_paranoid` value to 0 or less.
- To see the kernel module and its symbols, set `/proc/sys/kernel/kptr_restrict` to 0.
- Make sure there is a disk space on both target and host systems. Depending on the number of CPU cores, the amount of collected data may reach 1GB per second.
- Make sure your kernel version is 4.3 or higher.
- This mode is available for platforms based on Intel® microarchitectures code named Skylake and newer.

In the hardware tracing mode, you can do the following:

- Analyze user/kernel mode transitions and interrupts
- Explore execution of unexpected processes or system services
- Measure particular stages of workload execution without static instrumentation
- Analyze CPU core activities at the microsecond level
- Analyze a kernel/driver or application module by measuring exact CPU time with a nanosecond precision
- Triage latency issues resulted from:
 - changes in the execution code flow
 - preemption by another process
 - resource sharing issues
 - page faults
 - power consumption issues caused by unexpected wake-ups

NOTE

- This analysis requires a direct access to the hardware. It does not work inside a Guest VM.
 - In most cases, the collection overhead in this mode is less than 10%. It can be higher if your application is IO or DRAM bound.
 - The Hardware Tracing mode does not require sampling drivers.
-

Configure and Run Analysis

To configure options for the System Overview analysis:

Prerequisites: Create a project.

1. Click the



Configure Analysis button on Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From **HOW** pane, click the

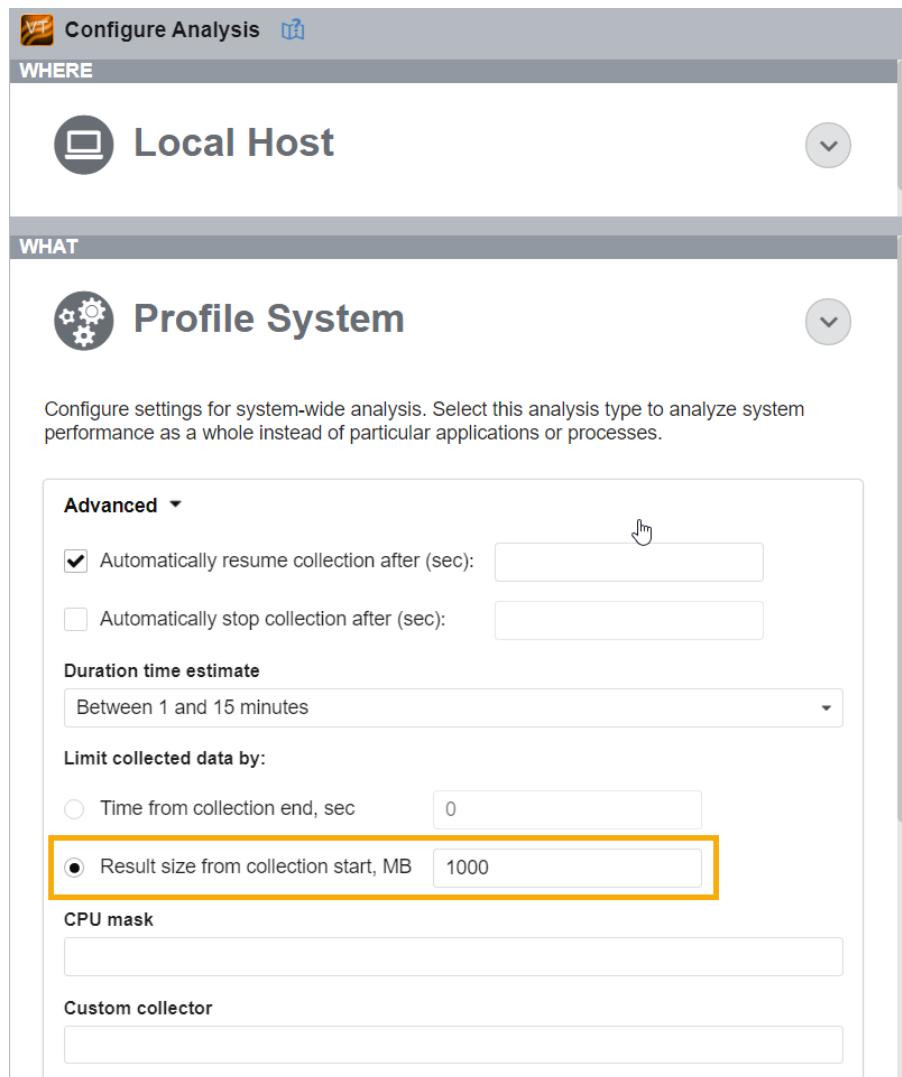


Browse button and select **System Overview**.

3. Select **Hardware Tracing** or **Hardware Event-Based Sampling** mode.

For the **Hardware Tracing** mode, you can also enable the **Analyze interrupts** option.

With the default **Hardware Tracing** configuration, Intel® VTune™ Profiler stops the data collection when a 1GB data limit is reached. You can change this limit in the **Advanced** section of the **WHAT** pane:



4. In the **HOW** pane, check options if you are interested in examining [power usage](#) or understanding reasons for [throttling behavior](#).
5. Click the



Start button to run the analysis.

VTune Profiler collects the data, generates a `rxxxxso` result, and opens it in the default System Overview viewpoint.

NOTE

To run this analysis from the command line, use the



Command Line button at the bottom.

Power Usage Analysis

Use the power consumption analysis capabilities of the System Overview analysis to get energy consumption characterization for your system.

To collect power usage data, check the **Analyze power usage** checkbox in the **HOW** pane of the **Configure Analysis** window. Then run the analysis.

The screenshot shows the Intel VTune Profiler interface. At the top, there's a navigation bar with tabs like 'How', 'Platform', 'Summary', etc. Below it is a title bar with a chart icon and the text 'System Overview'. To the right of the title is a circular button with a refresh symbol. The main area has a sub-header 'Analyze general behavior of target system to explore resource utilization and identify platform level factors that limit performance. [Learn more](#)'. Underneath, there's a section for 'CPU sampling interval, ms' with a input field containing '1'. Below this is a group of checkboxes: one checked ('Analyze power usage') and one unchecked ('Analyze throttling reasons'). At the bottom of this group is a 'Details' button with a right-pointing arrow. The entire 'How' pane is highlighted with a yellow border.

Once the data collection is finished, see the **Energy Consumption** section of the **Summary** window.

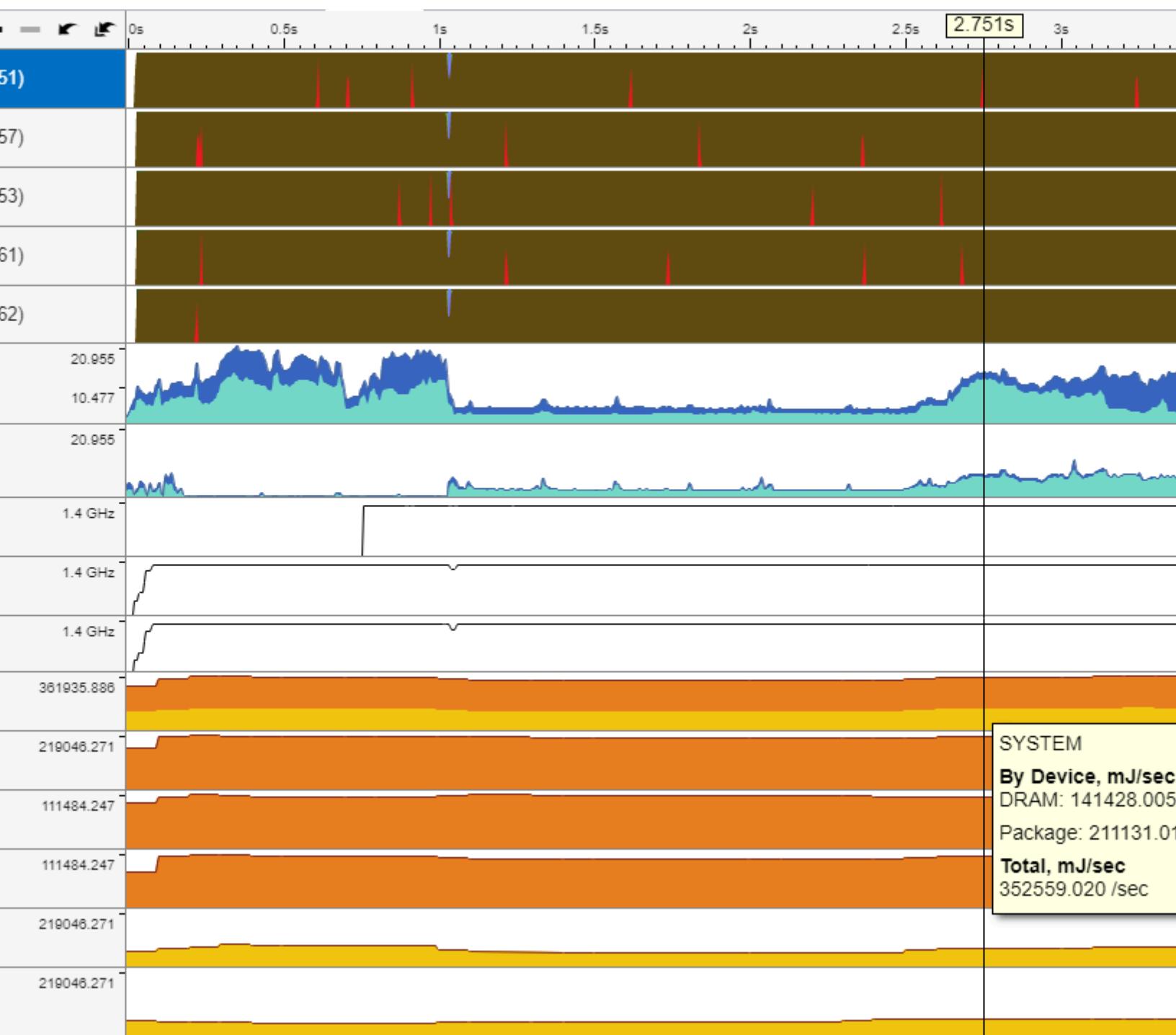
This section shows the total power consumed by the system during data collection, as well as the breakdown by CPU package and DRAM module.

Energy Consumption

Profiled Entity Hierarchy	Energy Consumption (mJ)
SYSTEM	1,236,923.096
CPU	756,935.913
Package_0	403,550.110
Package_1	353,385.803
DRAM_0	234,840.637
DRAM_1	245,146.545

*N/A is applied to non-summable metrics.

Switch to the **Platform** window to get a detailed view of power consumption over time. You can correlate different metrics, such as DRAM bandwidth, CPU frequency, and CPU utilization, with the amount of power consumed by each device.

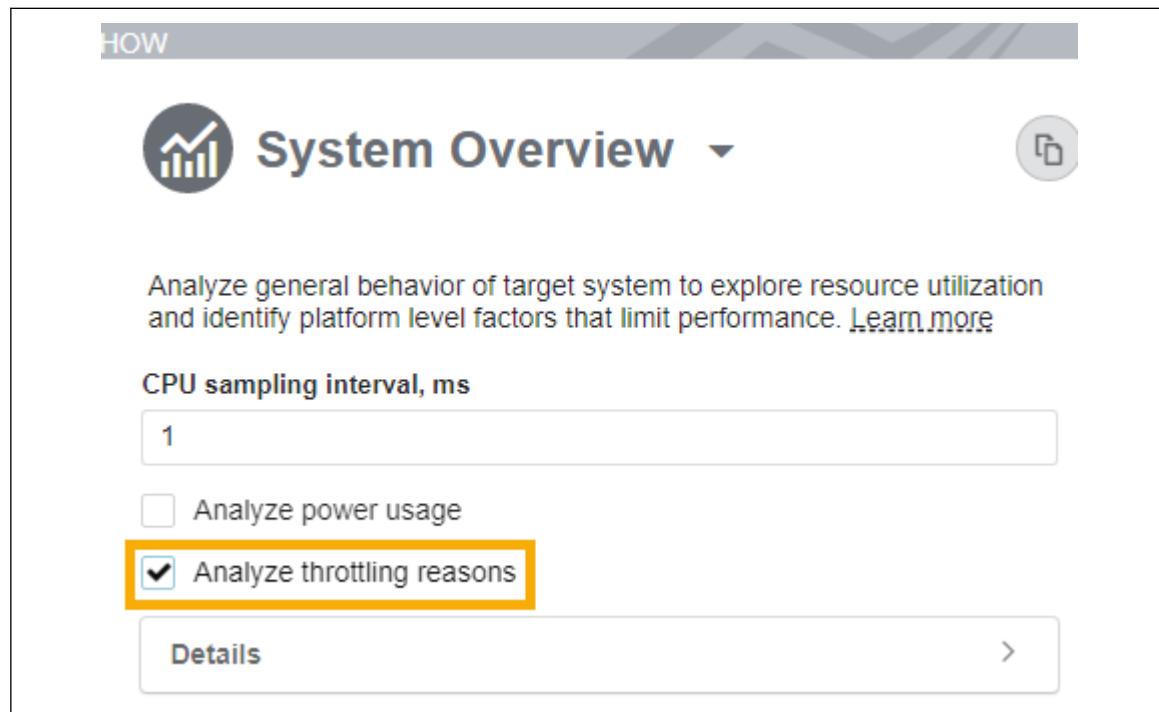
**NOTE**

On the timeline, device power is represented in millijoules per second, which is physically equivalent to milliwatts.

Throttling Analysis

If your CPU is operating at temperatures outside safe thermal limits, you may observe a significant drop in CPU frequency as the system attempts to stabilize. The drop in frequency to restore safe CPU operating temperature can result in significant performance loss. Run the System Overview analysis to analyze factors that can cause the CPU to throttle in this way.

In the **HOW** pane of the **Configure Analysis** window, check the **Analyze throttling reasons** checkbox. Then run the analysis.

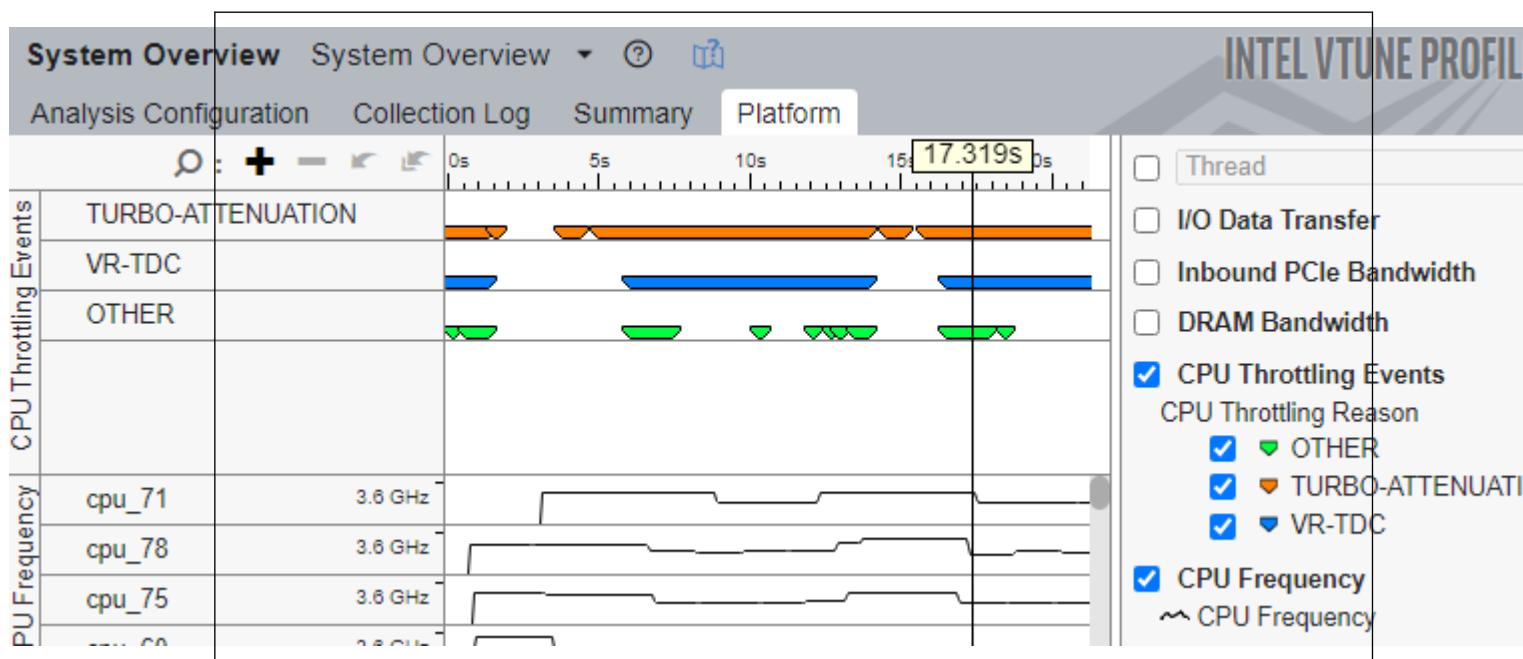


Once the data collection is finished, see the **CPU Throttling Reasons** section in the **Summary** window.

CPU Throttling Reasons	
This section displays the top reasons for CPU throttling and the percentage of samples affected. For more information, see a Description .	
Throttling Reason	% of Samples
BO-ATTENUATION	34.9%
TDC	24.2%
ER	5.1%

is applied to non-summable metrics.

Switch to the **Platform** window to see a breakdown of throttling events according to the reasons causing them.



CPU Throttling Reasons:

Use this information to understand throttling behavior and make necessary changes to your system configuration. In this table, frequency refers to the processor core frequency.

Reason	Description
PROCHOT	Frequency has dropped below the OS frequency due to assertion of external PROCHOT.
THERMAL	Frequency has dropped below the OS frequency due to a thermal event.
RSR-LIMIT	Frequency has dropped below the OS frequency due to a Residency State Regulation Limit violation.
RATL	Frequency has dropped below the OS frequency due to a Running Average Thermal Limit violation.
OTHER	Frequency has dropped below the OS frequency due to electrical or other constraints.
PBM-PL1	Frequency has dropped below the OS frequency due to package/platform-level power limiting PL1.
PBM-PL2	Frequency has dropped below the OS frequency due to package/platform-level power limiting PL2/PL3.
MAX-TURBO-LIMIT	Frequency has dropped below the OS frequency due to multi-core turbo limits.
TURBO-ATTENUATION	Frequency has dropped below the OS frequency due to turbo transition attenuation. This can cause performance degradation due to frequent changes in operating ratio.

For more information about these reasons, see the [Intel® 64 and IA-32 Architectures Software Development Manual](#).

See Also

[Analyze Interrupts](#)

Analyze Latency Issues

Task Analysis

Cookbook: Profiling Hardware without Sampling Drivers

Linux* and Android* Kernel Analysis

configuration

collect system-overview

vtune option for command line analysis

Analyze Interrupts

If you configured your collection to monitor IRQ Ftrace* events either by using the [System Overview analysis type](#) or [custom analysis](#), the Intel® VTune™ Profiler analyzes code performance inside IRQs and displays interrupts statistics in the default Hardware Events viewpoint. Follow the steps below to analyze the collected interrupt data:

- [Identify most critical interrupt handlers.](#)
- [Analyze slow interrupts on the timeline.](#)

Prerequisites

Analysis of interrupts requires access to the Linux Ftrace subsystem in /sys/kernel/debug/tracing. Typically, it is only accessible for the root user.

To analyze interrupts, either run the analysis as root, or edit permissions for /sys/kernel/debug/tracing as described in the Limitations section of the [Linux* and Android* Kernel Analysis](#) topic.

Identify Critical Interrupt Handlers

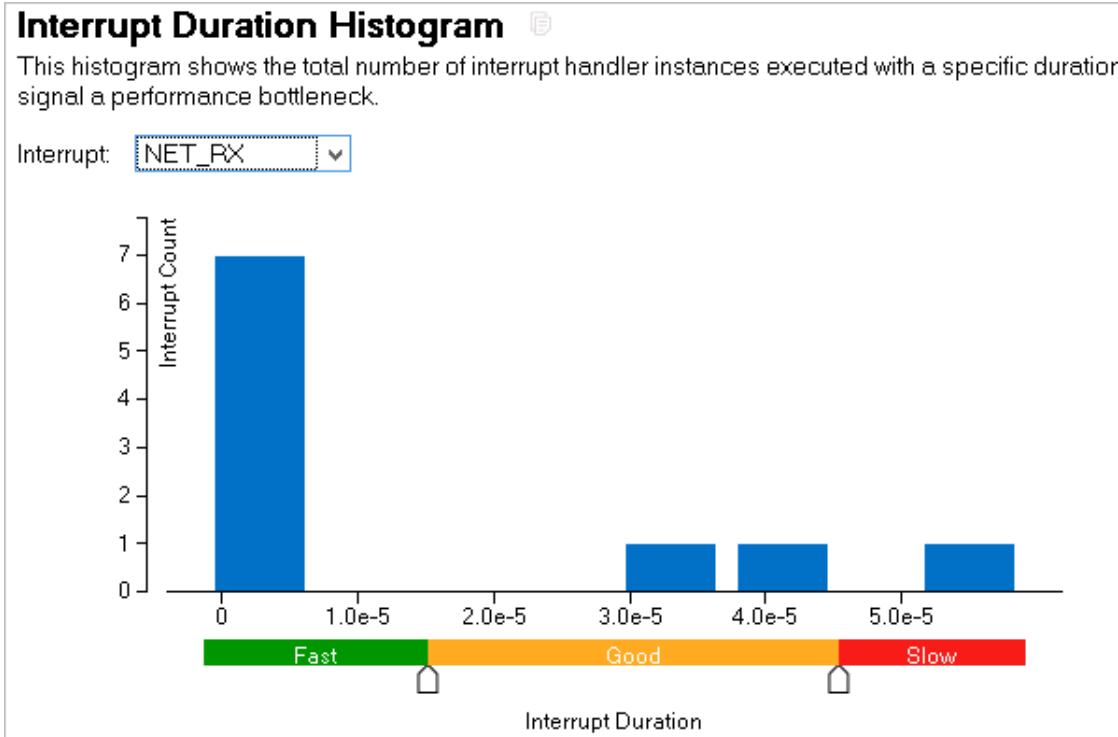
Start your analysis with the **Summary** window that provides overall interrupt handlers statistics in the following sections:

- **Top Interrupt Handlers** that shows the most active interrupt handlers sorted by [Interrupt Time](#).

Top Interrupt Handlers		
Interrupt	Interrupt Time ⓘ	Interrupt Count ⓘ
TASKLET	0.338s	7,952
nvidia	0.123s	4,825
NET_RX	0.034s	2,998
firewire_ohci	0.026s	4,825
TIMER	0.020s	10,028
[Others]	0.061s	21,943

Clicking an interrupt handler in the list opens the grid view grouped by **Interrupt/Interrupt Duration Type/Function/Call Stack** level.

- **Interrupt Duration Histogram** that shows a distribution of interrupt handler instances per duration types defined by the VTune Profiler. High number of slow instances may signal a performance bottleneck. Use the drop-down menu to view data for different interrupt handlers.

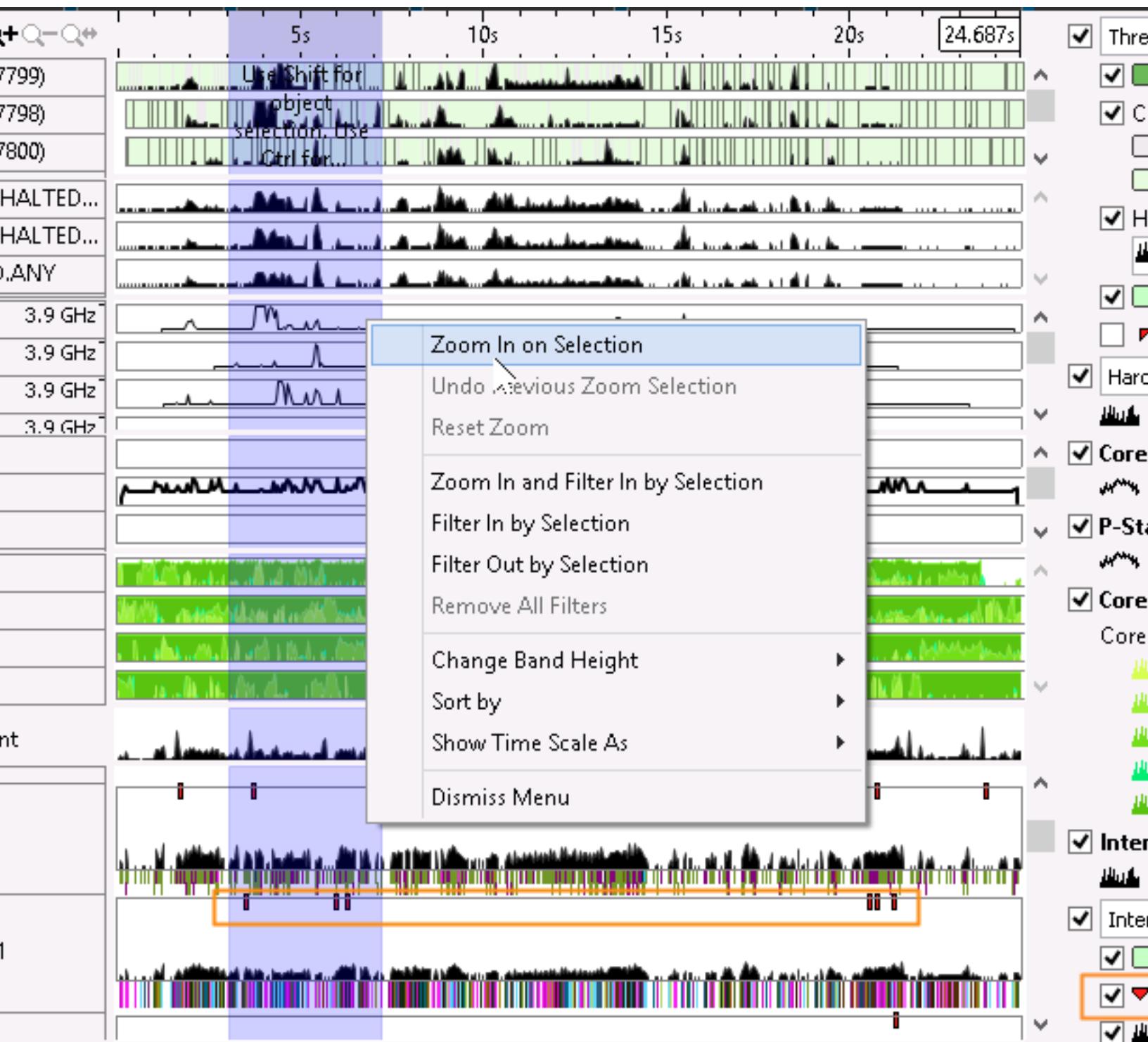


When you identified a slow interrupt in the **Summary** window, you may switch to the **Event Count** tab sorted by the **Interrupt/..** level, locate this interrupt, expand the hierarchy to view a function where slow interrupts occurred, and double-click the function to explore its source code in the Source view.

Grouping: Interrupt / Interrupt Duration Type / Function / Call Stack						
Interrupt / Interrupt Duration	Hardware Event Count by Hardware Event Type			Task Time	Task Count	Interrupt Count
	CPU_CLK_...	CPU_CLK_U...	INST_RETIRE...			
TIMER	7,000,000	21,000,000	0	0.003ms	0	10,028
TASKLET	17,500,000	17,500,000	7,000,000			7,952
nvidia	7,000,000	0	0			4,825
ehci_hcd:usb1	3,500,000	3,500,000	0			4,825
Fast						3,764
Good	3,500,000	3,500,000	0			1,055
Slow						6
firewire_ohci						4,825

Analyze Slow Interrupts on the Timeline

Switch to the **Platform** tab in the **Hardware Events** viewpoint to analyze CPU utilization, GPU usage and power consumption during your code execution and correlate this data with the time frames when slow interrupts occurred. You may enable the **Slow Interrupts** markers on the timeline, select a time frame with slow interrupts and zoom in to the selected region for detailed analysis:



See Also

[Linux* and Android* Kernel Analysis
for IRQ event collection](#)

[Window: Platform](#)

Analyze Latency Issues

Run the System Overview analysis in the Hardware Tracing mode to identify what caused latency issues in your application execution.

The System Overview viewpoint for [Hardware Tracing collection](#) provides the following data:

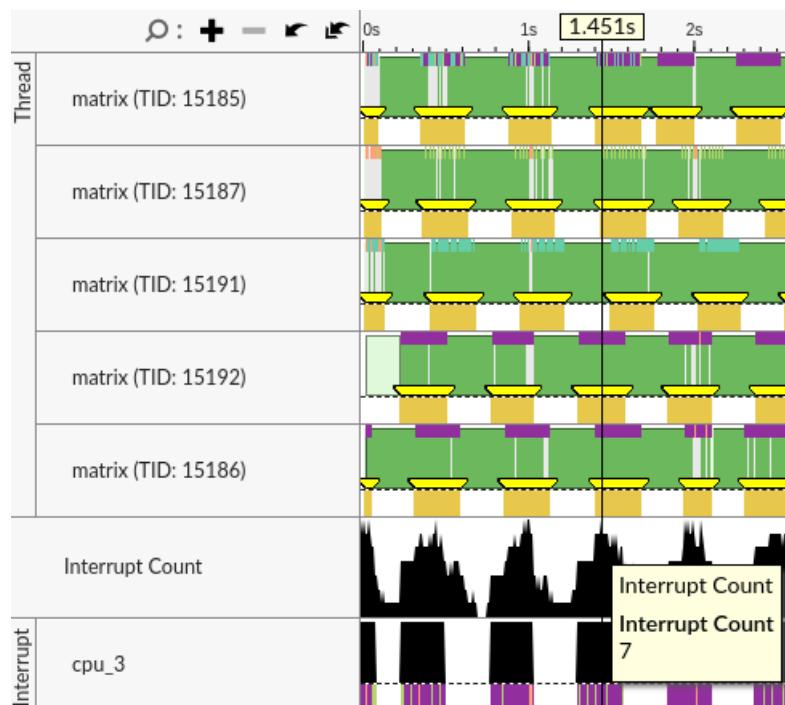
- system-wide statistics over time with the microsecond granularity
- module boundaries on the timeline
- function names for module entry points
- Active/Idle thread time
- interrupts on the timeline
- user-mode and kernel-mode execution times for modules and module entry points

You can use this data to:

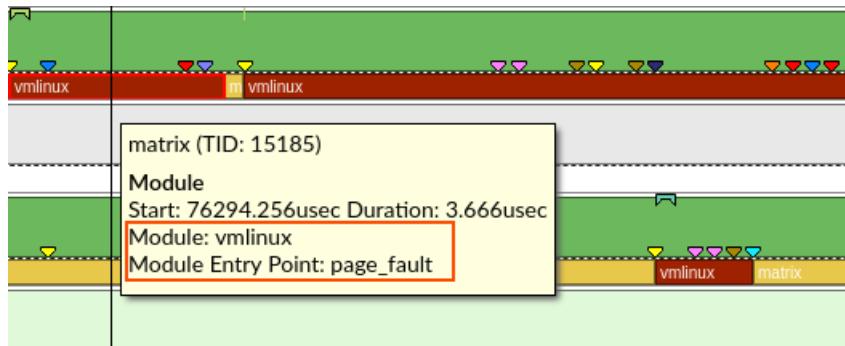
- [Explore the impact of interrupts on the application Elapsed Time](#)
- [Analyze thread activity at the microsecond level](#)
- [Explore kernel activity](#)

Explore the Impact of Interrupts on the Application Elapsed Time

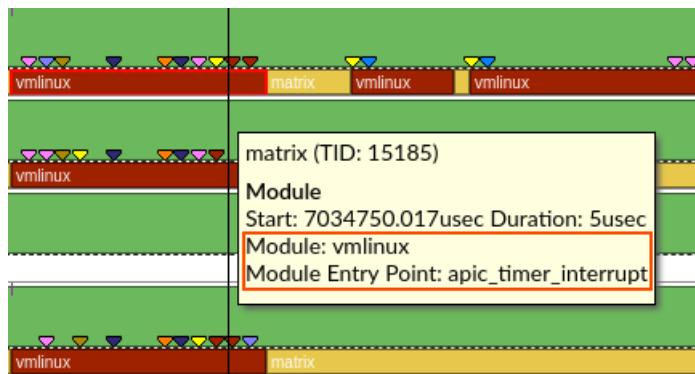
Open the **Platform** window to explore interrupts on the timeline view. The **Interrupt Count** chart provides a quick overview of the number of interrupts triggered during execution.



Locate the interrupt-intensive regions and zoom in. Hover over a module name to see the **Module Entry Point** that discovers a cause for an interrupt. For example, a page fault:



Or a timer interrupt:

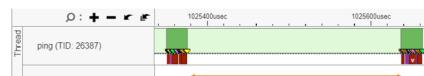


Analyze Thread Activity at the Microsecond Level

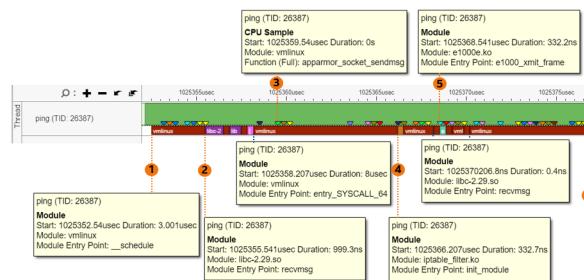
Hardware Tracing analysis enables you to analyze data at a high granularity level. This could be particularly useful, for example, to debug a network workload with a one-second duration between requests:



Zoom in to a single request. For example, the ping application measures and prints 250µs as reply time:



You can go deeper and analyze execution of each module:



1 Scheduler is becoming active after idle.

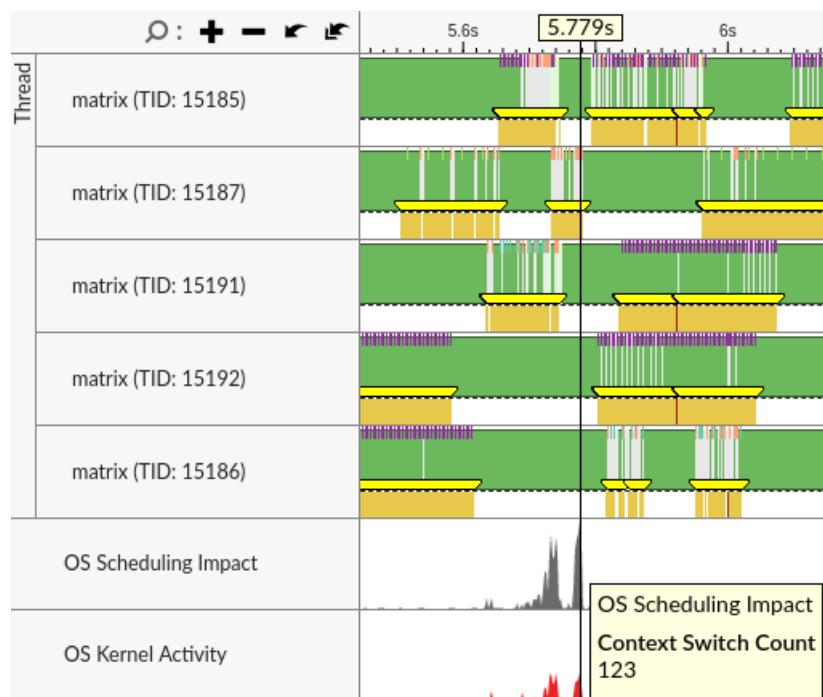


2 recvmsg is used to sleep for 1 second.

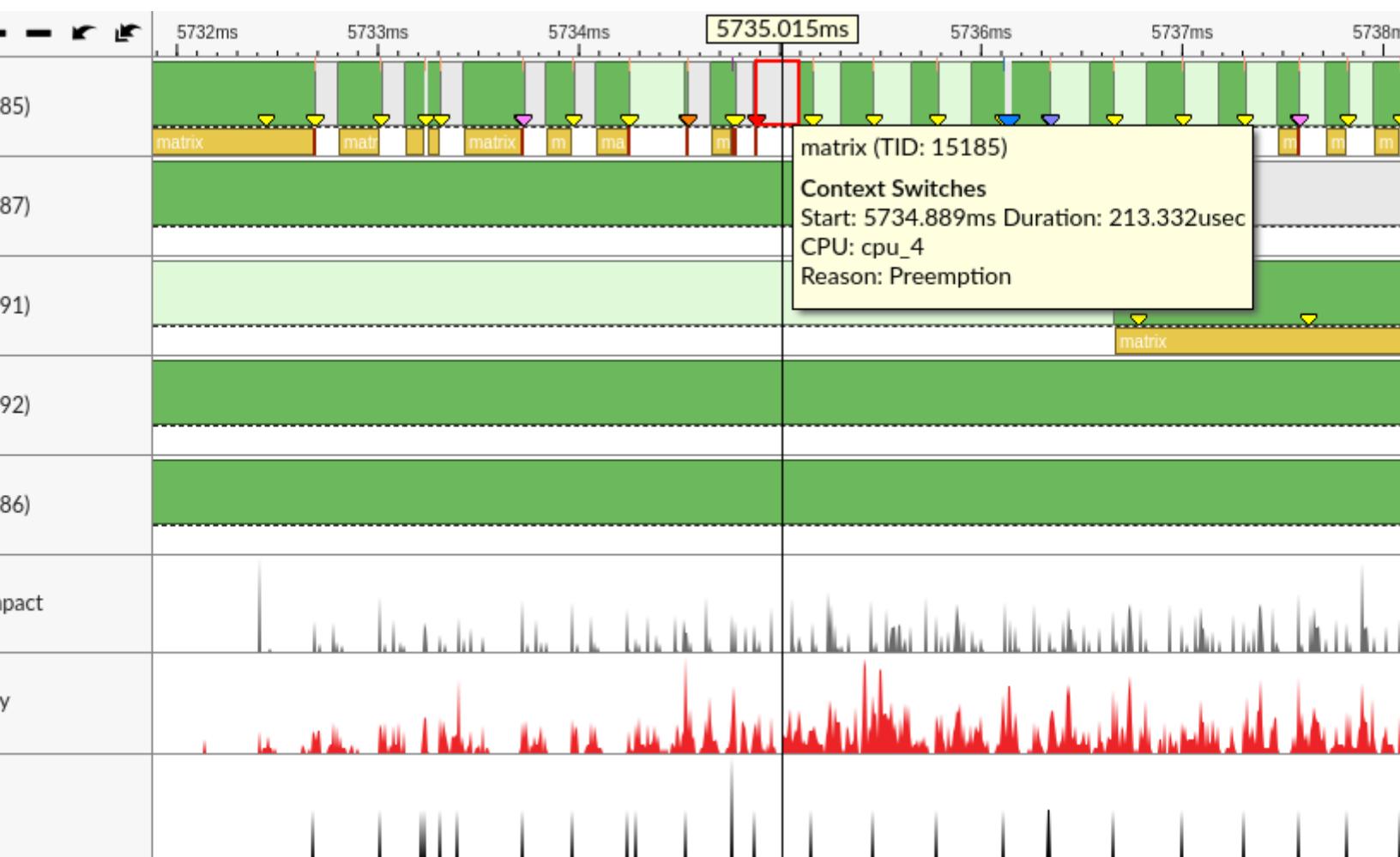
- 3 A new message is sent.
- 4 Iptable driver is active.
- 5 Network driver takes just 332 nanoseconds.
- 6 do_idle is executed.

Explore Kernel Activity

The **OS Scheduling Impact** and **OS Kernel Activity** charts highlight regions where the system has performed multiple context switches and kernel-mode entries.



Locate a region with multiple context switches or high kernel activity and zoom in to investigate. For example, in this case the operating system has rescheduled a thread multiple times due to various reasons, including preemption and synchronization. Hover over the markers to get additional details and to determine the root cause of the issue.



In the grid pane of the **Platform** window, use the **Process / Module / Module Entry Point** grouping to get a detailed view of user-mode and kernel activity. Expand a module and study the module entry points to determine the amount of time spent by the module in the kernel mode.

You can also examine the number and frequency of **Kernel-mode Entries** caused by a specific module and function to determine the performance impact of kernel activity.

Grouping: Process / Module / Module Entry Point					
Process / Module / Module Entry Point	CPU Time ▾			Kernel-mode Entries ▾	
	User	Kernel	Unknown	Count	Frequency, count/sec
matrix	42.118s	0.253s	-0.000s	74,160	6704.082
▶ matrix	42.118s	0s	0s	0	0.000
▼ vmlinux	0s	0.248s	0s	60,972	5511.884
▶ apic_timer_interrupt	0s	0.073s	0s	11,427	1033.004
▶ native_write_msr	0s	0.067s	0s	100	9.040
▶ page_fault	0s	0.043s	0s	20,397	1843.894
▶ call_function_interrupt	0s	0.026s	0s	12,963	1171.858
▶ perf_instruction_pointer	0s	0.015s	0s	4,195	379.229
▶ reschedule_interrupt	0s	0.008s	0s	2,649	239.470
▶ notifier_call_chain	0s	0.005s	0s	1,217	110.017

Hardware Tracing collection is more precise than event-based sampling and provides all the modules executed with their precise time.

See Also

[Analyze Interrupts](#)

Platform Analysis

NOTE Starting with the 2024.0 release of Intel® VTune™ Profiler, the **Platform Profiler** application is available as a separate download. Access this application from the [Intel Registration Center](#).

The **Platform Profiler** application will be discontinued in a future release. As a workaround, consider using the EMON data collector. To learn more, see this [transition article](#).

- To collect platform behavior data using the 2023.2 or newer versions of VTune Profiler, use the standalone Platform Profiler collector. You can then visualize collected data with the Platform Profiler server. See [Platform Analysis](#) for more information.
 - The Platform Profiler analysis type is not available in versions of VTune Profiler newer than 2023.2. To use the Platform Profiler analysis type (from the GUI or command line), switch to a version of VTune Profiler older than 2023.2. You can then follow procedures described in [Platform Profiler Analysis](#).
-

Use Intel® VTune™ Profiler-Platform Profiler to get a holistic view of system behavior. You can then perform system characterization on a deployed system that runs a full load over an extended period of time.

With Intel® VTune™ Profiler-Platform Profiler, you can get insights into these aspects:

- Platform configuration
- Utilization
- Performance
- Imbalances related to compute, memory, storage, IO, and interconnects

You can use Platform Profiler to conduct a coarse-grained, system-level analysis. Use the collected data to triage and characterize your system for a particular workload. This method differs from the **System Overview Analysis** in some important ways:

Aspect	System Overview Analysis	Analysis using Platform Profiler
Type of analysis	Fine-grained	Coarse-grained
Coverage	Hardware and software	Hardware only
Type of workload	Light workloads (runtime around a few minutes)	Heavy workloads (runtime running to several hours)

For heavy workloads with long runtimes, run Platform Profiler to ensure that you use available hardware in the most optimal way.

Platform Profiler consists of a command line data collector and a server implementing a RESTful interface to a time-series database. The collector ships with the VTune Profiler package. You can run Platform Profiler on Windows* and Linux* systems.

Platform Analysis Workflow

Here is the basic workflow to use this application:

1. Configure the collector environment.
2. Start Platform Profiler.
3. After data collection, stop Platform Profiler.
4. Import the collected data into Intel® VTune™ Profiler-Platform Profiler server.
5. View the collected data.

Configure the Collector Environment

When you configure your environment for the first time, you must have root/Administrator privilege.

To set up your environment, run `vpp-collect-vars`.

- For a Linux OS, source `vpp-collect-vars.sh` in the server directory.

```
$ source /opt/intel/oneapi/vtune/latest/vpp/collect/vpp-collect-vars.sh
```

- For a Windows OS, run the `vpp-collect-vars.cmd` script in the server directory.

```
$ C:\Program Files (x86)\Intel\oneAPI\vtune\latest\vpp\collector\vpp-collect-vars.cmd
```

Next, create the Platform Profiler server virtual Python* environment. At the command prompt, type:

```
$ vpp-server-config
```

Start Platform Profiler

1. In the command window, type:

```
$ vpp-collect start -c 'data collection comment'
```

where the comment argument is optional.

2. If you want to insert marks in the data metrics timeline, run:

```
vpp-collect mark 'optional comment'
```

Stop Platform Profiler

When you have finished collecting data, run this command to stop Platform Profiler:

```
vpp-collect stop
```

After data collection completes, Platform Profiler compresses the results into a `.tgz` (Linux) or `.zip` (Windows) file whose name contains the name of the target system and a date/time stamp.

Import Collected Data

Next, you import the collected data into Platform Profiler server. Use the Platform Profiler server to examine a performance overview of system behavior. Understand platform-level configuration, utilization and imbalance issues related to compute, memory, storage, IO and interconnects.

1. To start, set up the environment for Platform Profiler server. When you configure your environment for the first time, you must have root/Administrator privilege.

- For a Linux OS, source `vpp-server-vars.sh` in the server directory.

```
$ source /opt/intel/oneapi/vtune/latest/vpp/collect/vpp-server-vars.sh
```

- For a Windows OS, run the `vpp-server-vars.cmd` script in the server directory.

```
$ C:\Program Files (x86)\Intel\oneAPI\vtune\latest\vpp\server\vpp-server-vars.cmd
```

2. Next, create the virtual Python* environment for Platform Profiler server.

```
vpp-server config
```

NOTE

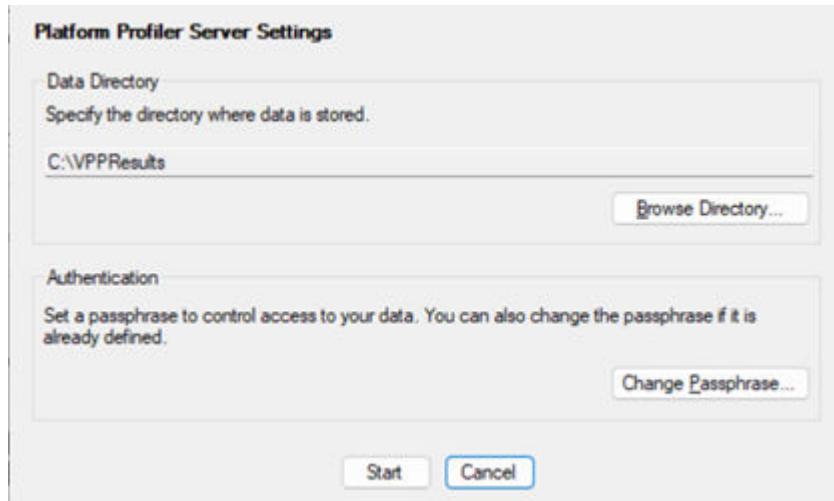
You can configure several command line options, if necessary:

Option	Purpose
--webserver-port PORT	Change the default port (originally 6543) on which the Platform Profiler web server listens for connections.
--database-port PORT	Change the default port (originally 8086) on which the database server of the Platform Profiler listens for connections.
--data-dir PATH	Change the default directory where the Platform Profiler data is stored.
--reset-passphrase	Display the server password prompt so that it can be changed.
--quiet	Silence all prompts and accept the default data directory.

3. Start the Platform Profiler server using default settings. Run this command:

```
vpp-server start
```

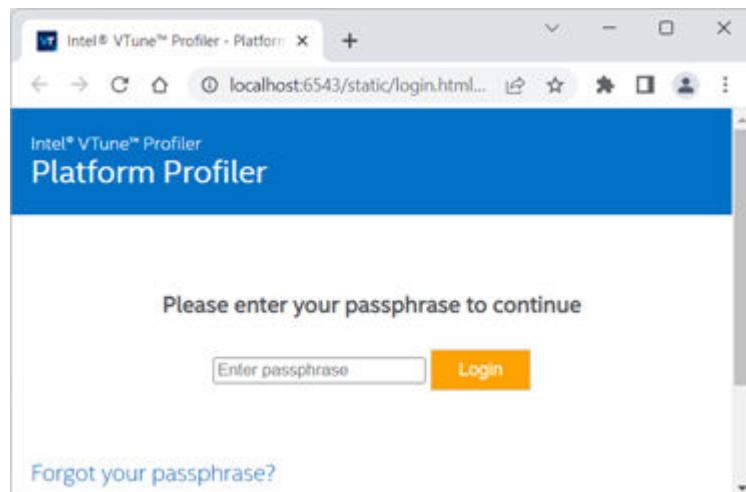
4. In the **Server Settings** dialog box, specify a directory for storage and/or authentication if necessary.



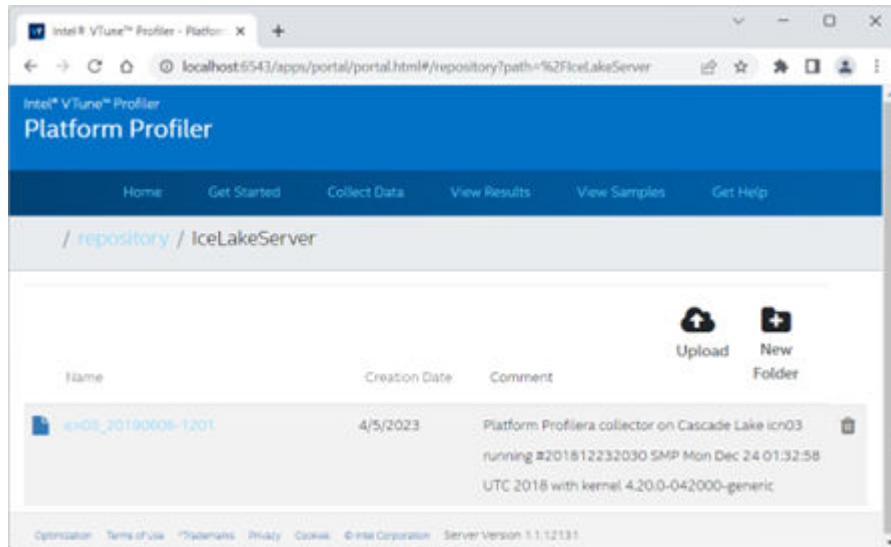
5. Open a web browser to the address and port number of the server instance of Intel® VTune™ Profiler-Platform Profiler. For example, in the address bar of the web browser, type:

```
localhost:6543
```

6. Enter your passphrase for the database and click **Login**. You can also create a new folder to store the imported results and then click on the folder name to open it.



7. Once you log in, go to the **View Results** tab and click the **Upload** button.



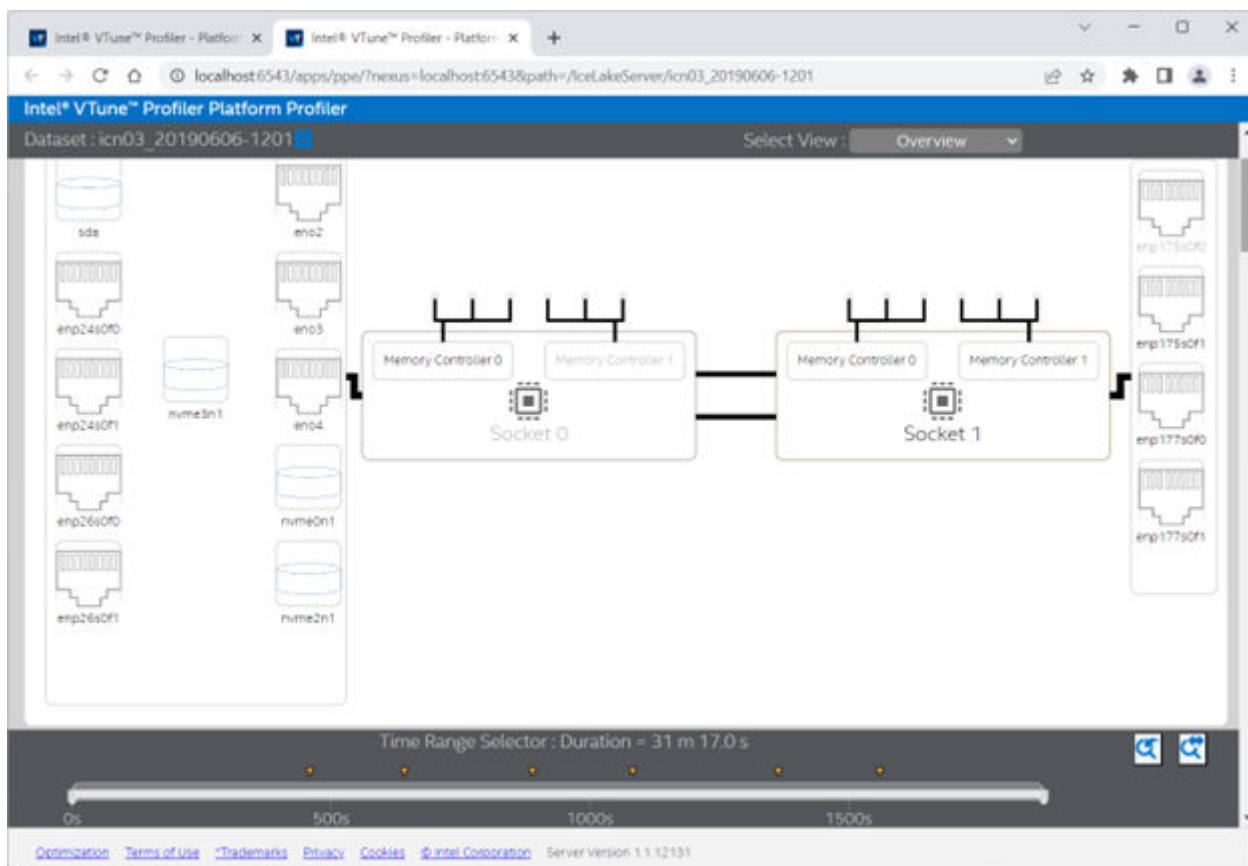
8. After the data import completes, you can view results by clicking on the results name.

View Collected Data

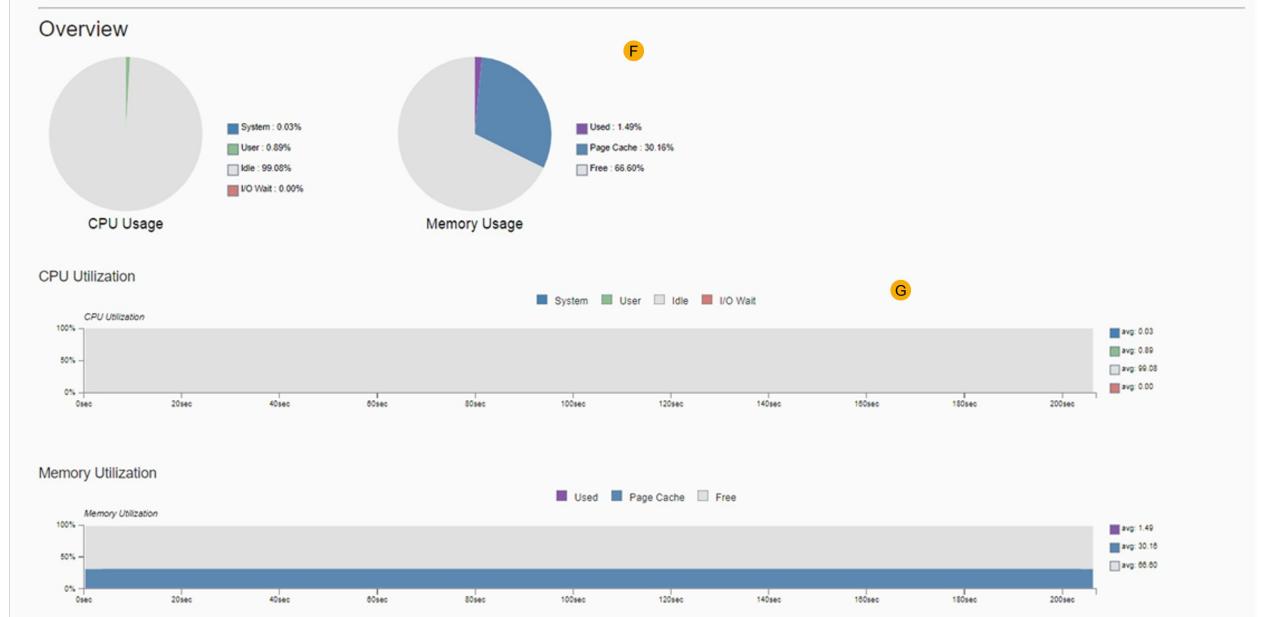
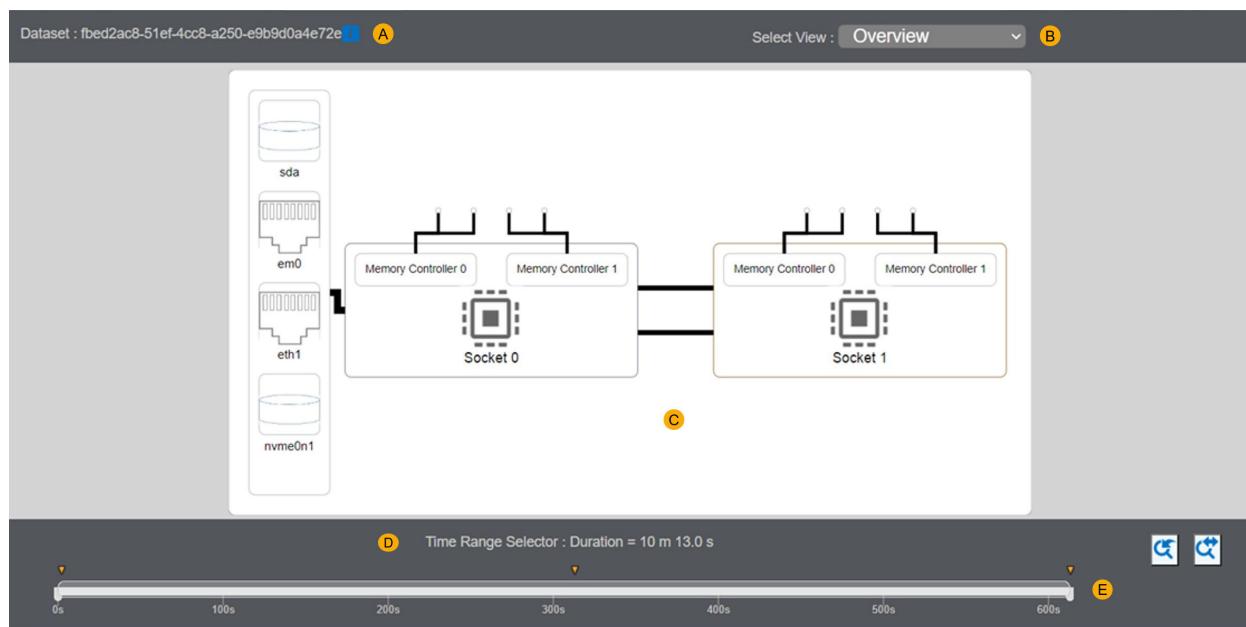
After data collection and import, VTune Profiler server displays information in three areas:

- The **Platform Configuration Diagram** - Use this diagram to get quick metrics for each subsystem (socket, core, memory, disk, etc.).
- An interactive timeline - Filter on a smaller range of collection time.
- Detailed performance charts

Start with the platform configuration diagram to see configuration details and key metrics. Hover over component icons to see additional details for a component.



There are several views to help you visualize and interpret the collected data. A good starting point is the **Overview**, which you can select from the **Select View** pulldown menu.



A

Hover over here to see information about the system used for data collection.

B

See different views for information on sockets, cores, memory, and storage devices.

C

Hover over here to see additional information about the platform configuration. Click on specific elements to switch views.

D

Filter the data for a specific time range.

E

Undo or reset a zoom level.

F

See summary information about CPU and memory utilization.

G

See performance information over time. Click and drag to select and zoom into a specific time range.

Next Steps

- Consider whether an upgrade to hardware components (CPU, memory, storage, network) could improve performance. After you install new hardware, repeat the platform analysis and compare performance between the older and newer components.
- Analyze the collected data to determine the most prevalent performance bottlenecks and the most impacted components. If a specific portion of the workload is causing performance issues, consider running the following analysis types using VTune Profiler using a targeted collection interval (seconds instead of hours):
 - **Microarchitecture Exploration:** Identify issues with CPU utilization, cache, or memory
 - **Memory Access:** Identify memory issues
 - **Input and Output:** Identify storage usage issues

Hybrid CPU Analysis

Understand how to use Intel® VTune™ Profiler to run analyses on hybrid CPUs with several types of cores.

A hybrid CPU combines several types of cores on the same die. For example, Intel® microarchitectures code named Alder Lake have two types of cores – Performance cores (P-Cores) and Efficient Cores(E-Cores). When you profile applications that run on hybrid CPUs, use these techniques to conduct a good performance analysis.

Group by Core Type

When you run hardware event-based sampling analysis, VTune Profiler detects the various types of CPU cores and provides you with an option to group or filter the profiling results by *Core Type* entity. Use this grouping to understand:

- The time spent by the application on each core type
- What portion of the code was executed in a certain location
- Moments when transitions happened

Group by Core Type in Grid

In the grid view in the Bottom-up window, select one of the groupings that feature 'Core Type'. For example, this table displays grouping by Core Type / Physical Core / Logical Core / Function / Call Stack.

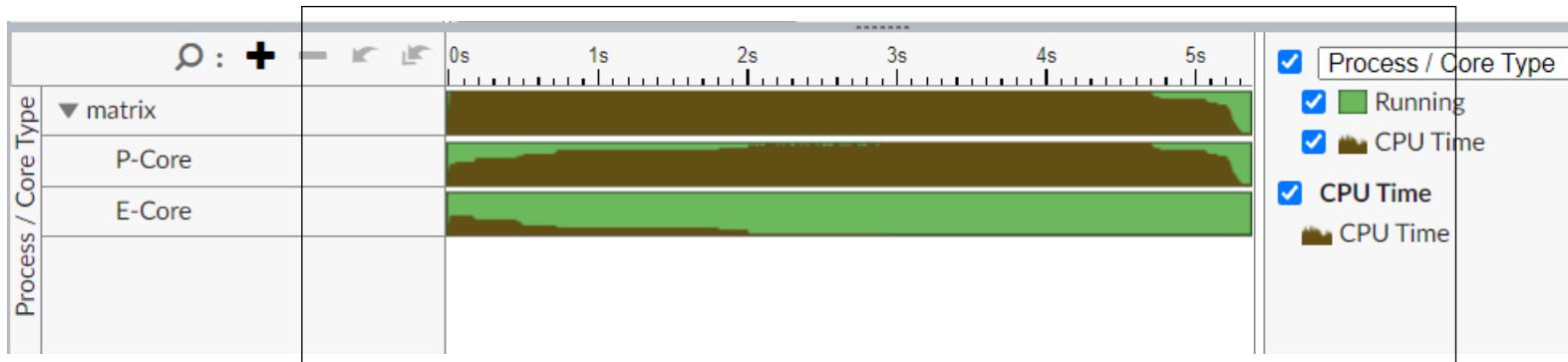
Microarchitecture Exploration				
Analysis Configuration Collection Log Summary Bottom-up Event Count Platform				
Grouping: Core Type / Physical Core / Logical Core / Function / Call Stack				
Core Type / Physical Core / Logical Core / Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate
▼ P-Core	72.159s	288,147,600,000	62,688,600,000	4.596
▶ core_5	10.337s	41,279,400,000	8,764,200,000	4.710
▶ core_6	10.119s	40,413,600,000	8,526,600,000	4.740
▶ core_4	9.889s	39,587,400,000	8,316,000,000	4.760
▶ core_0	9.717s	38,763,000,000	8,267,400,000	4.689
▶ core_1	9.403s	37,533,600,000	8,161,200,000	4.599
▶ core_2	8.025s	32,036,400,000	7,246,800,000	4.421
▶ core_3	7.782s	31,044,600,000	7,140,600,000	4.348
▶ core_7	6.887s	27,489,600,000	6,265,800,000	4.387
▼ E-Core	8.405s	26,778,600,000	6,953,400,000	3.851
▶ core_8	2.853s	9,315,000,000	2,415,600,000	3.856
▶ core_9	1.964s	6,357,600,000	1,567,800,000	4.055
▶ core_10	1.777s	5,736,600,000	1,434,600,000	3.999
▶ core_11	0.706s	2,160,000,000	520,200,000	4.152

You can also create your own grouping and include the 'Core Type' entity in it. To do this, use the **Customize Grouping** dialog box from the **Grouping** pulldown menu and select your combination of entities.

The screenshot shows the Intel VTune Profiler interface with the 'Project Navigator' on the left and the 'Microarchitecture Exploration' window on the right. In the 'Microarchitecture Exploration' window, the 'Grouping' dropdown is open, showing 'Function / Call Stack' as the current selection. A 'Customize the grouping:' dialog box is overlaid on the main window. This dialog has two main sections: 'Select grouping levels from:' and 'Customize the grouping:'. The 'Select grouping levels from:' section lists several entities: Class, Code Location, Core Type, Data Address, Frame, and Frame Domain. 'Core Type' is highlighted with a blue selection bar. The 'Customize the grouping:' section shows a mapping: 'Function' is mapped to 'Call Stack'. Below the dialog, a note says 'The 'Call Stack' level should close the grouping.' At the bottom right of the dialog are 'Save' and 'Cancel' buttons.

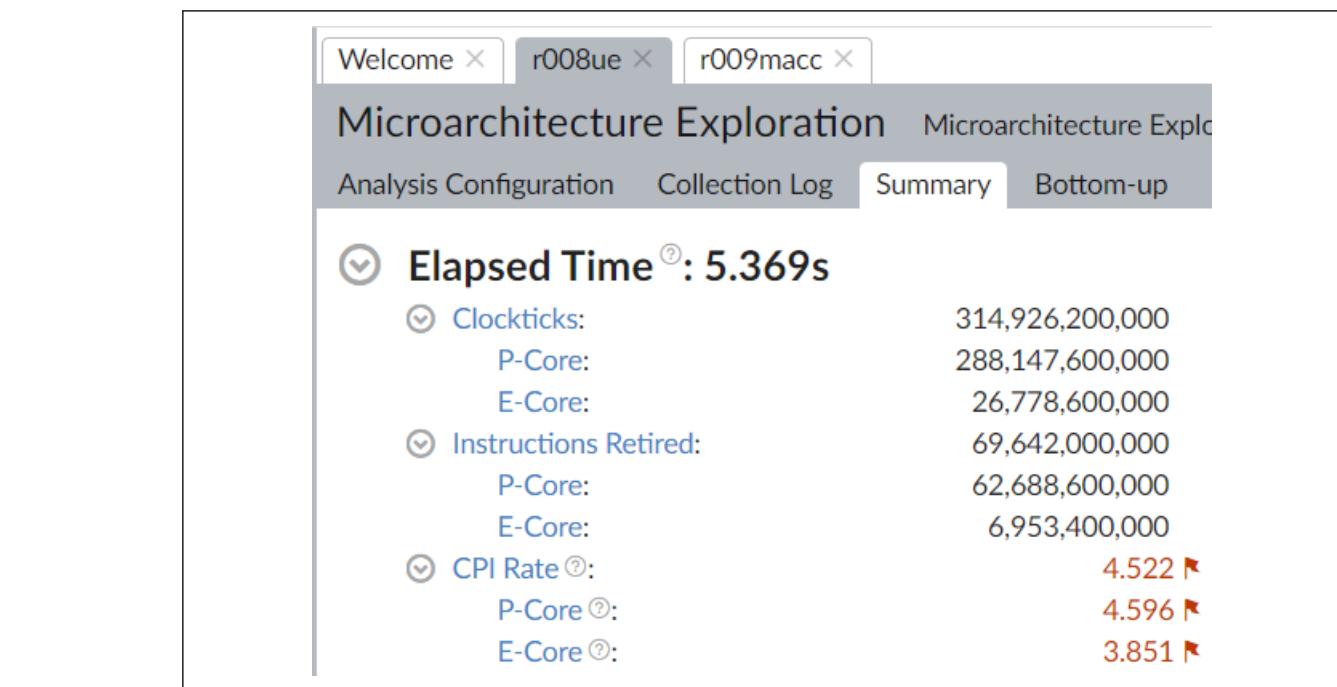
Group by Core Type in Timeline

You can also use the timeline view to group data by Core Type. To do this, select one of the available groupings (from the pulldown menu) that contain the Core Type entity. This example shows the **Process / Core Type** grouping in the timeline.



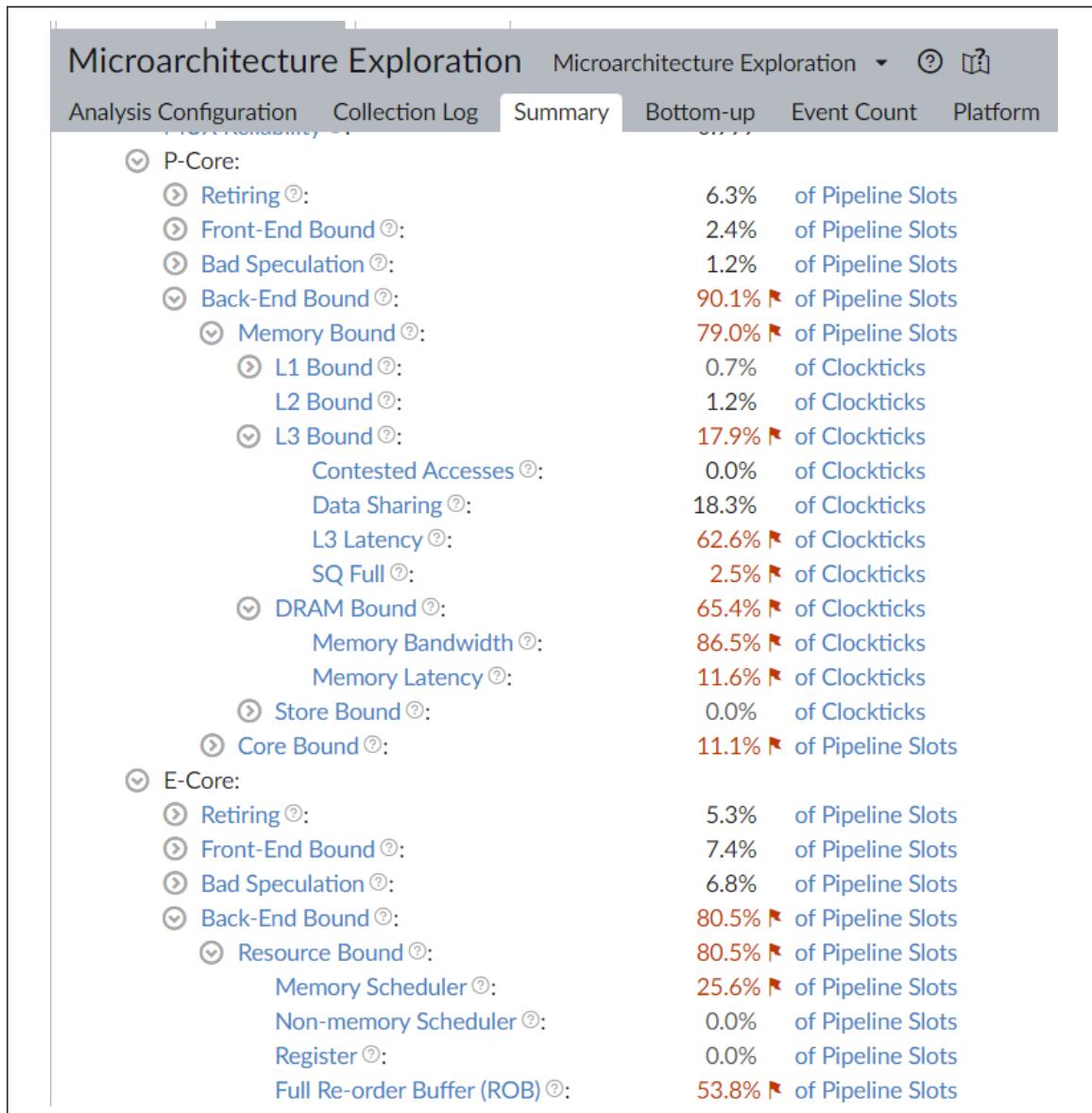
Metrics for Hybrid CPUs

When you profile applications on hybrid platforms, several metrics in the Summary window display data per core type as well as data that is aggregated across all core types.



Microarchitecture Exploration Metrics

This image displays the metric hierarchy in the results of a Microarchitecture Exploration analysis. The data is displayed per core type for hybrid processors.



Use this hierarchical display of data to analyze microarchitecture bottlenecks in P-Cores and E-Cores. You will also find a similar breakdown by core type in other analysis types (Memory Access or HPC Performance Characterization) since they share some of the same metrics.

Source Code Analysis

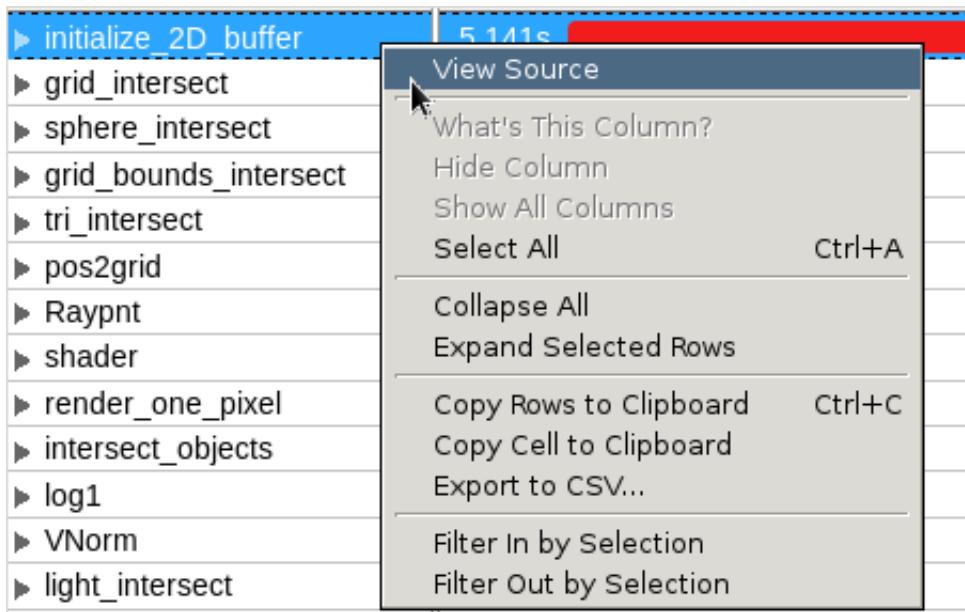
For better understanding of a performance problem, associate a hotspot with the source code and exact machine instruction(s) that caused this hotspot.

Prerequisites

Intel® VTune™ Profiler provides accurate source analysis if your code is compiled with the debug information and debug information is written correctly in the binary file (for [Linux* targets](#)) or debug information file/symbol file (for [Windows* targets](#)).

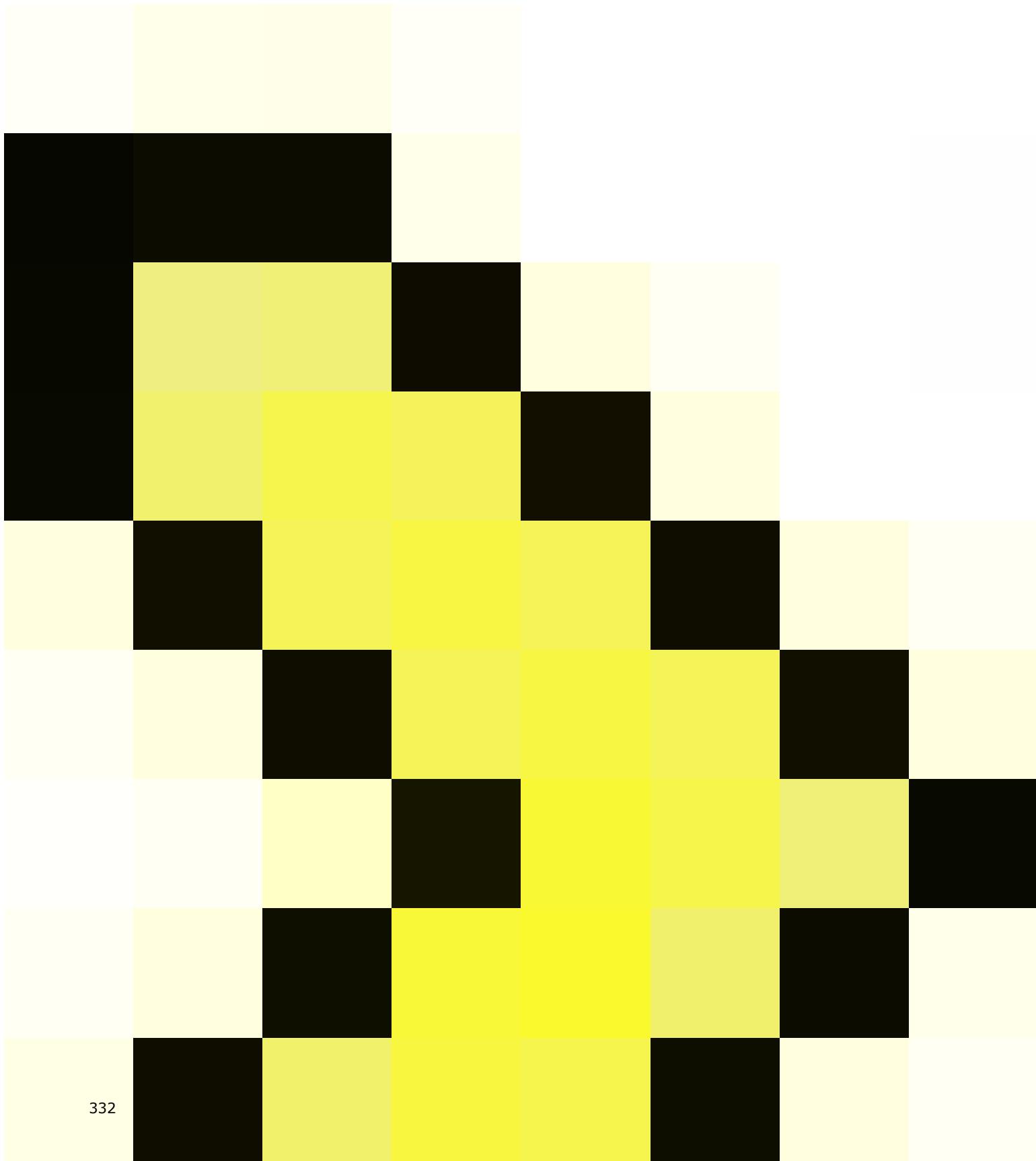
Access Source View

To open the source/assembly code of a specific item, either double-click the selected item in the grid view/**Call Stack/Timeline** pane, or select the **View Source** option from the context menu:



Depending on the route you used to access the Source view, the data representation on the panes may slightly differ:

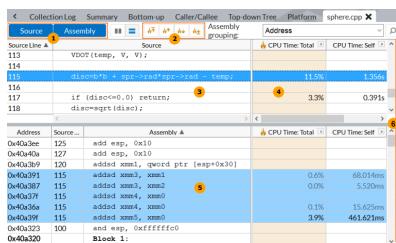
- If you access the Source view by clicking a function in the grid, the VTune Profiler opens the source at the *hottest* (with the highest value of the metric selected for hotspot navigation) line of this function in the **Source/Assembly** pane.
- When you click a call stack function, the VTune Profiler opens the source highlighting the *call site* (location where a function call is made) at the top of the call stack. The call site is marked with the yellow arrow



- If you click a wait in the **Timeline** pane, the VTune Profiler opens a wait function highlighting the waiting call site. If you double-click a transition (for Threading data), it highlights the signaling call site.

Analyze Code

The **Source/Assembly** window opens in a separate tab:



- 1 Source/Assembly toggle buttons.** By default, depending on the symbol information availability, the VTune Profiler opens one of the panes: Source or Assembly. But you can use the toggle **Source** and **Assembly** buttons on the toolbar to manage the view and enable both of them if required/possible.

The content displayed on the **Source** and **Assembly** panes is correlated. When you select an element on one pane, another pane scrolls to the corresponding elements and highlights them.

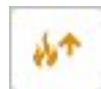
NOTE

- One source code line may have one or more related assembly instructions while one instruction has only one related code line.
- Synchronization is possible only if the debug line information is available for the selected function.

- 2 Hotspot navigation buttons.** Typically, the VTune Profiler opens the source code highlighting the most performance critical code line based on the key metric set up for this analysis. To go further and freely navigate between code lines that have the highest metric value (*hotspots*), use these buttons toolbar:



- Go to the code line that has the maximal metric value.



- Go to the previous (by metric value) hotspot line.



- Go to the next (by metric value) hotspot line.



- Go to the code line that has the minimal metric value.

- 3 The **Source** pane shows your code written on a high-level programming language, for example, C, C++, or Fortran. The **Source** pane opens if the symbol information for the selected function is available.
- 4 **Hotspot navigation metric** column. By default, the source view navigation is based on the key analysis metric like the CPU Time for the Hotspots analysis. Such a metric column is highlighted. To change the hotspot navigation metric, right-click the required column and select **Use for Hotspot Navigation** command from the context menu.
- 5 The **Assembly** pane displays disassembled code. This code shows the exact order of the assembly instructions executed by the processor. Instructions on the **Assembly** pane are grouped into basic blocks. To get help on a particular instruction, select it in the grid, right-click and choose **Instruction Reference** from the context menu.

For better navigation in the Assembly pane, you may select one of the available granularity levels in the **Assembly grouping** drop-down menu: **Address**, **Basic Block/Address**, or **Function Range/Basic Block/Address**. VTune Profiler updates the Assembly view grouping the instructions into collapsible nodes according to the selected hierarchy.

If there is no correct debug information, or symbol file is unavailable, the assembly data may be incorrect. In this case, the VTune Profiler uses heuristics to define function boundaries in the binary module.
- 6 **Heat map markers**. Use the blue markers to the right of the vertical scroll bar to quickly identify the hotspot lines (based on the hotspot navigation metric). To view a hotspot, move the scroll bar slider to the marker. The bright blue marker (

) indicates a hot line for the function you drilled down into. Light blue markers (

) indicate hot lines for other functions.

Edit Source

When tuning your target, you may need to modify the source code. VTune Profiler enables you to open the source files for editing directly from the Source/Assembly window.

To launch the source editor:

1. In the **Source** pane, select a line you want to edit.
2. Right-click the line and select **Edit Source** from the context menu, or click the **Open Source File Editor**



button on the Source/Assembly toolbar.

Your source code opens in the code editor set in your system as default. For example, on Linux the code editor is defined in the *EDITOR* environment variable (for example, *vi*) or *VISUAL* environment variables (for example, *gedit*, *emacs*). Depending on the editor application, the code may open exactly on the selected line.

After editing your code, rebuild your target and re-run the VTune Profiler analysis on the modified version to compare the performance results before and after optimization.

NOTE

The Source/Assembly analysis is not supported for the source code using the `#line` directive.

See Also

[Debug Information for Linux* Application Binaries](#)

[Debug Information for Windows* Application Binaries](#)

[Debug Information for Windows* System Libraries](#)

[View Source Objects from Command Line](#)

[Compare Source Code](#)

Custom Analysis

Create a new custom analysis type based on available predefined analysis configurations.

To create and run a new custom analysis type:

Prerequisites: Make sure a VTune Profiler [project](#) is created.

1. Click the



(standalone GUI)/



(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. From the **HOW** pane, click the



Browse button and choose an analysis type to customize, for example: Threading.

3. Click the



Copy button.

VTune Profiler creates an editable copy of the selected configuration and adds it to the **Custom Analysis** section.

4. Manage the custom configuration using the following controls:



Enable an editable mode for the configuration and specify the following analysis identifiers:

- **Analysis name:** Enter/edit a name of this custom analysis type.
- **Command line name:** Enter/edit a name of the custom analysis type that will be used as an identifier when analyzing the project from the command line. Keep it short for your convenience.
- **Analysis identifier:** Specify a shorthand identifier to be appended to the name of each result produced by this analysis type. For example, adding the `tr` identifier for the Threading analysis result produces the following result name: `r000tr`, where `000` is the result number.
- **Comments:** Provide a short meaningful description of the analysis type you create. This information may help you easily identify the analysis type specifics later.



Customize a copy of the selected analysis.

3

Delete the custom analysis.

Configuration options available for a new custom configuration depend on the original analysis you customize.

5. Click the **Start** button to run the analysis.

See Also

[Custom Analysis Options](#)

[runsa/runss Custom Command Line Analysis](#)

Custom Analysis Options

If you [create a copy](#) of a predefined analysis type, a new custom configuration inherits all options available for the original analysis and makes them editable.

This is a list of all available custom configuration options (knobs) in the alphabetical order:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A	
Analyze I/O waits check box	Analyze the percentage of time each thread and CPU spends in I/O wait state.
Analyze interrupts check box	Collect interrupt events that alter a normal execution flow of a program. Such events can be generated by hardware devices or by CPUs. Use this data to identify slow interrupts that affect your code performance.
Analyze loops check box	Extend loops analysis to collect advanced loops information, such as instructions set usage and display analysis results by loops and functions.
Analyze memory bandwidth check box	Collect events required to compute memory bandwidth.
Analyze memory consumption check box (for Linux targets only)	Collect and analyze information about memory objects with the highest memory consumption .
Analyze memory objects check box (for Linux* targets only)	Enable the instrumentation of memory allocation/de-allocation and map hardware events to memory objects.
Analyze OpenMP regions check box	Instrument the OpenMP* regions in your application to group performance data by regions/work-sharing constructs and detect inefficiencies such as imbalance, lock contention, or overhead on performing scheduling, reduction, and atomic operations. Using this option may cause higher overhead and increase the result size.
Analyze PCIe bandwidth check box	Collect the events required to compute PCIe bandwidth. As a result, you will be able to analyze the distribution of the read/write operations on the timeline and identify where your application could be stalled due to approaching the bandwidth limits of the PCIe bus.

A

In the **Device class** drop-down menu, you can choose a device class where you need to analyze PCIe bandwidth: processing accelerators, mass storage controller, network controller, or all classes of the devices (default).

NOTE

This analysis is possible only on the Intel microarchitecture code name Haswell EP and later.

Analyze power usage check box	Track power consumption by processor over time to see whether it can cause CPU throttling.
Analyze Processor Graphics hardware events drop-down menu	Analyze performance data from Intel HD Graphics and Intel Iris Graphics (further: Intel Graphics) based on the predefined groups of GPU metrics .
Analyze system-wide context switches check box	Analyze detailed scheduling layout for all threads on the system and identify the nature of context switches for a thread (preemption or synchronization).
Analyze user tasks, events, and counters check box	Analyze tasks, events, and counters specified in your code via the ITT API . This option causes a higher overhead and increases the result size.
Analyze user histogram check box	Analyze the histogram specified in your code via the Histogram API . This option increases both overhead and result size.
Analyze user synchronization check box	Enable User synchronization API profiling to analyze thread synchronization. This option causes higher overhead and increases result size.

C

Chipset events field	Specify a comma-separated list of chipset events (up to 5 events) to monitor with the hardware event-based sampling collector.
Collect context switches check box	Analyze detailed scheduling layout for all threads in your application, explore time spent on a context switch and identify the nature of context switches for a thread (preemption or synchronization).
	<p>NOTE</p> <p>The types of the context switches (preemption or synchronization) cannot be identified if the analysis uses Perf* based driverless collection.</p>
Collect CPU sampling data menu	Choose whether to collect information about CPU samples and related call stacks.
Collect highly accurate CPU time check box (for Windows targets only)	Obtain more accurate CPU time data. This option causes more runtime overhead and increases result size. Administrator privileges are required.

C

Collect I/O API data menu	Choose whether to collect information about I/O calls and related call stacks. This analysis option helps identify where threads are waiting or enables you to compute thread concurrency. The collector instruments APIs, which causes higher overhead and increases result size.
Collect Parallel File System counters check box	Enable collection of the Parallel File System counters to analyze Lustre* file system performance statistics, including Bandwidth, Package Rate, Average Packet Size, and others.
Collect signalling API data menu	Choose whether to collect information about synchronization objects and call stacks for signaling calls. This analysis option helps identify synchronization transitions in the timeline and signalling call stacks for associated waits. The collector instruments signalling APIs, which causes higher overhead and increases result size.
Collect stacks check box	Enable advanced collection of call stacks and thread context switches to analyze performance, parallelism, and power consumption per execution path.
Collect synchronization API data menu	Choose whether to collect information about synchronization wait calls and related call stacks. This analysis option helps identify where threads are waiting or enables you to compute thread concurrency. The collector instruments APIs, which causes higher overhead and increases result size.
Collect thread affinity check box	Analyze thread pinning to sockets, physical cores, and logical cores. Identify incorrect affinity that utilizes logical cores instead of physical cores and contributes to poor physical CPU utilization.

NOTE

Affinity information is collected at the end of the thread lifetime, so the resulting data may not show the whole issue for dynamic affinity that is changed during the thread lifetime.

CPU Events table	<ul style="list-style-type: none"> Specify hardware events to collect using the check boxes in the first column. By default, the table lists all events available for the target platform with events used for the original analysis configuration pre-selected. You may use the Search functionality to find events of interest. To get more details on an event, select it in the table and click the Explain button. Modify the Sample After value for an event to control the number of events after which the VTune Profiler interrupts the event data collection. The Sample After value depends on the target duration. Based on the duration value, the VTune Profiler adjusts the Sample After value with a multiplier.
CPU sampling interval, ms field	Specify an interval between collected CPU samples in milliseconds.

D

Disable alternative stacks for signal handlers check box (available for Linux targets)	Disable using alternative stacks for signal handlers. Consider this option for profiling standard Python 3 code on Linux.
--	---

E	
Enable driverless collection check box	Use driverless Perf*-based hardware event-based collection when possible.
Evaluate max DRAM bandwidth check box	Evaluate maximum achievable local DRAM bandwidth before the collection starts. This data is used to scale bandwidth metrics on the timeline and calculate thresholds.
Event mode drop-down list	Limit event-based sampling collection to USER (user events) or OS(system events) mode. By default, all event types are collected.
G	
GPU Profiling mode drop-down menu	Select a profiling mode to either characterize GPU performance issues based on GPU hardware metric presets or enable a source analysis to identify basic blocks latency due to algorithm inefficiencies, or memory latency due to memory access issues. Use the Computing task of interest table to specify the kernels of interest and narrow down the GPU analysis to specific kernels minimizing the collection overhead. If required, modify the instance step for each kernel, which is a sampling interval (in the number of kernels).
GPU sampling interval, ms field	Specify an interval between GPU samples.
GPU Utilization check box (for Linux* targets available with Intel HD Graphics and Intel Iris® Graphics only)	Analyze GPU usage and identify whether your application is GPU or CPU bound.
L	
Limit PMU collection to counting check box	Enable to collect counts of events instead of default detailed context data for each PMU event (such as code or hardware context). Counting mode introduces less overhead but gives less information.
Linux Ftrace events / Android framework events field	Use the kernel events library to select Linux Ftrace* and Android* framework events to monitor with the collector. The collected data show up as tasks in the Timeline pane. You can also apply the task grouping level to view performance statistics in the grid.
M	
Managed runtime type to analyze menu	Choose a type of the managed runtime to analyze . Available options are: <ul style="list-style-type: none"> for Windows targets: combined Java* and .NET* analysis; combined Java, .NET and Python* analysis; Python only analysis for Linux targets: Java only analysis; combined Java and Python analysis; Python only analysis
Minimal memory object size to track, in bytes spin box (for Linux targets only)	Specify a minimal size of memory allocations to analyze. This option helps reduce runtime overhead of the instrumentation.

P	
Profile with Hardware Tracing check box	Enable driver-less hardware tracing collection to explore CPU activities of your code at the microsecond level and triage latency issues.
S	
Stack size, in bytes field	Specify the size of a raw stack (in bytes) to process. Unlimited size value in GUI corresponds to 0 value in the command line. Possible values are numbers between 0 and 2147483647.
Stack type drop-down menu	Choose between software stack and hardware LBR-based stack types. Software stacks have no depth limitations and provide more data while hardware stacks introduce less overhead. Typically, software stack type is recommended unless the collection overhead becomes significant. Note that hardware LBR stack type may not be available on all platforms.
Stack unwinding mode menu	Choose whether collection requires online (during collection) or offline (after collection) stack unwinding. Offline mode reduces analysis overhead and is typically recommended.
Stitch stacks check box	For applications using Intel® oneAPI Threading Building Blocks(oneTBB) or OpenMP* with Intel runtime libraries, restructure the call flow to attach stacks to a point introducing a parallel workload.
T	
Trace GPU Programming APIs check box	Capture the execution time of OpenCL™ kernels, SYCL tasks and Intel Media SDK programs on a GPU, identify performance-critical GPU tasks, and analyze the performance per GPU hardware metrics.
U	
Uncore sampling interval, ms field	Specify an interval (in milliseconds) between uncore event samples.
Use precise multiplexing check box	Enable a fine-grain event multiplexing mode that switches events groups on each sample. This mode provides more reliable statistics for applications with a short execution time. You can also consider applying the precise multiplexing algorithm if the MUX Reliability metric value for your results is low.

NOTE

You may [generate the command line](#) for this configuration using the **Command Line...** button at the bottom.

See Also**collect-with**

vtune option to configure custom analysis from command line

Highly Accurate CPU Time Data Collection

Configure the Intel® VTune™ Profiler on Windows* OS to get highly accurate CPU time data in the user-mode sampling and tracing results.

By default, the VTune Profiler detects CPU time based on the OS scheduler tick granularity. As a result, the CPU time values may be inaccurate for targets that execute in short quanta less than the OS scheduler tick interval (for example, frame-by-frame computation in video decoders).

Accurate collection of CPU time information is available for the [user-mode sampling and tracing](#) analysis types (Hotspots and Threading) and enabled by default in the predefined analysis configurations when you run both the VTune Profiler and your application to analyze with [administrator privileges](#).

To collect more accurate CPU time information, the VTune Profiler uses the Event Tracing for Windows* (ETW) capability. For example, without ETW, a sample is taken every 10ms. For each sample, the OS is queried for the amount of time the thread executed and the difference is calculated between the samples, resulting in the delta. The information returned by the OS via this mechanism has a coarse granularity. VTune Profiler totals the deltas and displays it in the user interface. However, with ETW enabled, the VTune Profiler can filter out any time spent executing other threads and accurately calculate time for monitored threads within each 10ms sample based on the context switch information acquired from ETW. Based on this additional information, the CPU time metric calculated for the function/thread will be more accurate.

VTune Profiler needs exclusive access to the Microsoft* NT Kernel Logger. Therefore, only one VTune Profiler collection can run in this mode on the system and no other tools can use the service. If the VTune Profiler cannot get access to the NT Kernel Logger, the collection will continue with this mode disabled.

This type of collection takes more processing time and disk space. VTune Profiler may generate up to 5 MB of temporary data per minute per logical CPU depending on the system configuration and the profiled target.

Enabling or disabling the accurate CPU time collection depends on what is executing on the system during data collection and the structure of your application. In specific cases, there may be about a 3% variation between "normal" and "highly accurate" CPU time. But, there are corner cases where the difference could be as high as 30% or 40%. If the thread is executing, but happens to be inactive every 10ms that a sample is taken without ETW, the results would grossly misrepresent the execution time. Or, if the thread is mostly inactive, but runs exactly on the frequency of the 10ms samples, it may appear to consume large amounts of time, when in reality it does not. The best thing to do is to test it yourself, if possible. That is, collect the Baic Hotspots data with and without this option on and compare the resulting data. This can tell you if running without the highly accurate CPU time option produces results accurate enough to direct your optimization efforts, or if you need to have Administrative privileges so that you can enable this option. However, if you are restricted from using highly accurate CPU time because of your corporation's policies, you can, in general, be confident that analysis of your application's performance is valid using "normal" Hotspots data collection.

To disable highly accurate CPU time collection for custom analysis:

1. [Create a new custom analysis](#) (based on an existing configuration such as Hotspots or Threading).
2. Deselect the **Collect highly accurate CPU time** option.

See Also

[knob](#)

accurate-cpu-time-detection option

[Warnings about Accurate CPU Time Collection](#)

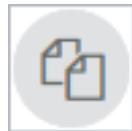
[Custom Analysis Options](#)

Hardware Event List

If required, edit a list of PMU events monitored by the Intel® VTune™ Profiler for your processor by modifying an existing or creating a new hardware event-based sampling (EBS) analysis configuration.

To add events:

1. In the **HOW** pane, select an existing hardware event-based analysis (for example, Microarchitecture Exploration) and click the



Copy button to create a custom copy of this configuration.

The new analysis type shows up under the **Custom Analysis** group in the **HOW** pane.

2. From the list of PMU events supported for the current platform, select the events you want the VTune Profiler to monitor in your new configuration.

Events configured for CPU: Intel(R) Processor code named Skylake ULT

NOTE: For analysis purposes, Intel VTune Amplifier 2018 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the target configuration window.

	Event Name	Sample After	Description
<input checked="" type="checkbox"/>	CPU_CLK_UNHALTED.THREAD	2400000	Core cycles when the thread is not in ...
<input checked="" type="checkbox"/>	CPU_CLK_UNHALTED.REF_TSC	2400000	Reference cycles when the core is not...
<input checked="" type="checkbox"/>	INST_RETired.ANY	2400000	Instructions retired from execution.
<input checked="" type="checkbox"/>	MEM_TRANS_RETired.LOAD_LATENCY...	10003	Counts loads when the latency from fir...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_L1D_MISS	2000003	Execution stalls while L1 cache miss d...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_L2_MISS	2000003	Execution stalls while L2 cache miss d...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_L3_MISS	2000003	Execution stalls while L3 cache miss d...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_MEM_ANY	2000003	Execution stalls while memory subsys...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.1_PORTS_UTIL	2000003	Cycles total of 1 uop is executed on al...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.2_PORTS_UTIL	2000003	Cycles total of 2 uops are executed on...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.BOUND_ON_STORES	2000003	Cycles where the Store Buffer was full...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.EXE_BOUND_0_PORTS	2000003	Cycles where no uops were executed,...

Explain

You may select an event and click the **Explain...** button at the bottom to open the [Intel Processor Event Reference](#) and read more details on the selected event.

To filter in/out the event list for particular event(s), specify search keywords (applied to both the **Event Name** and **Event Description** columns) in the **Filter** field.

NOTE

Usually [precise events](#) have a _PS postfix (for example, UOPS_RETired.RETIRE_SLOTS_PS) and/or a clear indication (Precise Event) in the Event Description column.

3. Click **Start** to run your new analysis configuration.

NOTE

You may configure the VTune Profiler to monitor all the events in a single collection run using event multiplexing or [allow multiple runs](#) to collect more precise event data.

See Also

[Custom Analysis Options](#)

knob

event-config option to specify events from CLI

Hardware Event Skid

Event skid is the recording of an event not exactly on the code line that caused the event.

Event skids may even result in a caller function event being recorded in the callee function.

Event skid is caused by a number of factors:

- The delay in propagating the event out of the processor's microcode through the interrupt controller (APIC) and back into the processor.
- The current instruction retirement cycle must be completed.
- When the interrupt is received, the processor must serialize its instruction stream which causes a flushing of the execution pipeline.

Intel® processors support accurate event location for some events. These events are called **precise events**.

Caution

The event skid affects the accuracy of your analysis results. When the **grouping level** is very small (for example, instruction, source line, or basic block), the Intel® VTune™ Profiler attributes performance results incorrectly. For example, when row A induces a problem, row B shows up as a hotspot. If different CPU events in the formula of a **hardware event-based metric** have different skids, the VTune Profiler may attribute data to different blocks, which makes all metrics invalid. This type of issue typically does not show up at the function granularity.

[Example: Interpreting Jump and Call Instructions](#)

Events that happen in the execution time of the `jmp` or `call` instruction, may appear on an instruction that is one or two instructions away from original `jmp`/ `call` in the execution flow. In this example, the `mov` instruction at the top of the loop is not responsible for the 1.02% of the events because the `mov` instruction is the target of the branch at the bottom of the loop. The real source of the events is the `jmp` instruction at the bottom of the loop.

Event %	Instructions
1.02%	<pre>top_of_loop: mov (any number of lines) end_of_loop: jnz <to someplace> jmp top_of_loop</pre>

See Also

[Hardware Event-based Sampling Collection](#)

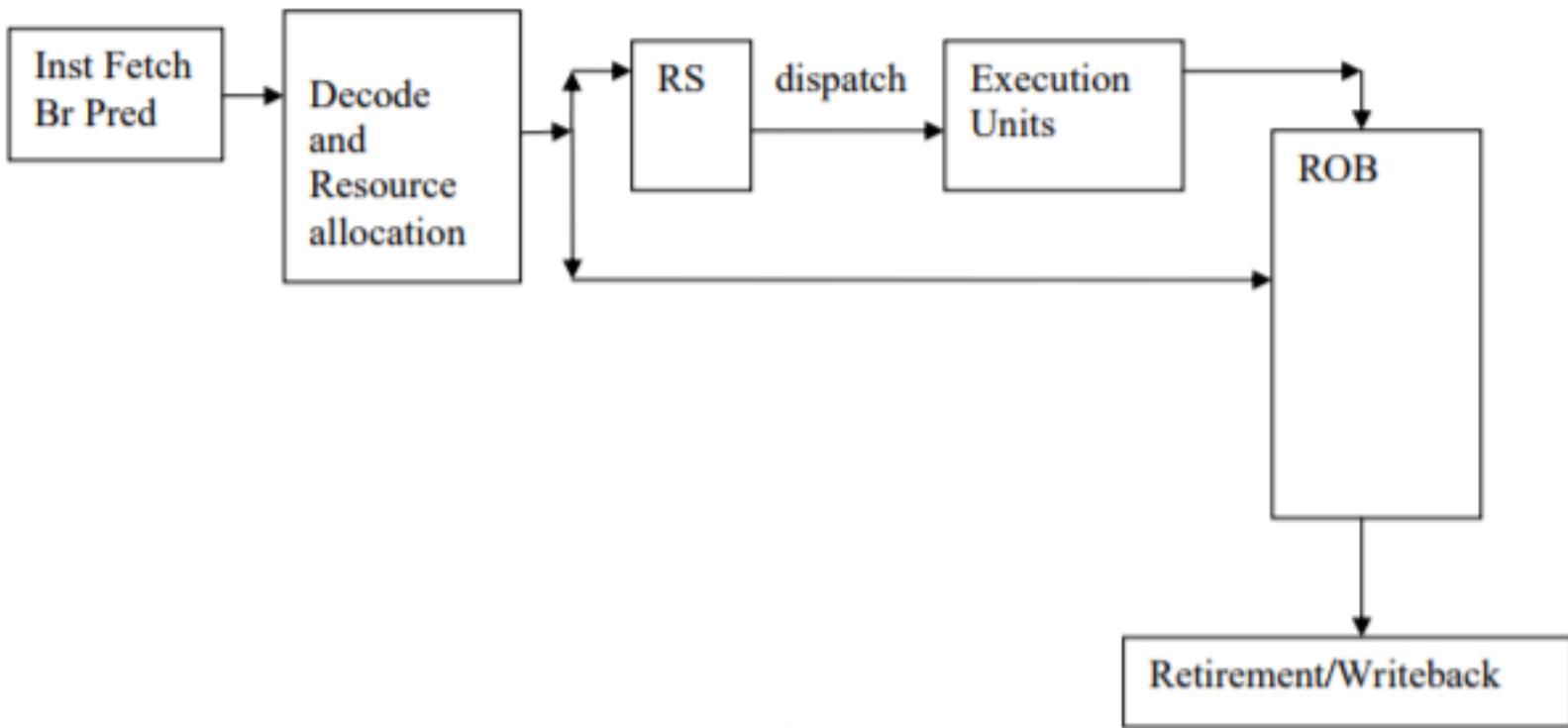
[Understanding How General Exploration Works in Intel® VTune™ Profiler](#)

Instructions Retired Event

The Instructions Retired is an important hardware performance event that shows how many instructions were completely executed.

Modern processors execute much more instructions than the program flow needs. This is called a *speculative execution*. Instructions that were "proven" as indeed needed by the program execution flow are "retired".

In the Core Out Of Order pipeline leaving the Retirement Unit means that the instructions are finally executed and their results are correct and visible in the architectural state as if they execute in-order:



Retirement and write back of state to visible registers is only done for instructions and uops that are on the correct execution path. Instructions and uops of incorrectly predicted paths are flushed upon identification of the misprediction and the correct paths are then processed. Retirement of the correct execution path instructions can proceed when two conditions are satisfied:

- The uops associated with the instruction to be retired have completed, allowing the retirement of the entire instruction, or in the case of instructions that generate very large number of uops, enough to fill the retirement window.
- Older instructions and their uops of correctly predicted paths have retired.

Intel® VTune™ Profiler monitors the Instructions Retired event for all analysis types based on the hardware event-based sampling (EBS), also known as Performance Monitoring Counter (PMC) analysis in the sampling mode. The Instructions Retired event is also part of the basic [Clockticks per Instructions Retired \(CPI\)](#) metric that shows how much latency affected an application execution.

For performance analysis, you may check how many instructions started their execution in OOO pipeline ([ISSUED](#) counter or [EXECUTED](#) counter) and compare the number with the count of retired operations. High difference shows that CPU does a lot of useless work and uses excess power.

See Also

[Hardware Event-Based Sampling Collection](#)

[Hardware Event Skid](#)

Precise Events

Precise events are events for which the exact instruction addresses that caused the event are available.

You can configure these events to collect extended information, the values of all the registers evaluated at the IP of the interrupt, on IA-32 and Intel® 64 architecture systems. For example, on Intel Core™ 2 processor family, an L2 load miss that retrieves a cacheline can be identified with the MEM_LOAD_RETIRE.L2_LINE_MISS event. The register values and the disassembly allows the reconstruction of the linear address of the memory operation that caused the event.

Check the **HOW** configuration pane in the **Configure Analysis** window to make sure the events you use are precise. Usually precise events have a _PS postfix (for example, MEM_LOAD_RETIRE.FB_HIT_PS) in the **Description** column as follows:

Events configured for CPU: Intel(R) Processor code named Skylake ULT			
NOTE: For analysis purposes, Intel VTune Amplifier 2018 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the target configuration window.			
<input type="text"/> Search.. Explain			
▼	Event Name	Sample After	Description
<input checked="" type="checkbox"/>	MEM_LOAD_L3_HIT_RETIRE.XSNP ...	20011	Retired load instructions which data sources were L3 hit and cr...
<input checked="" type="checkbox"/>	MEM_LOAD_RETIRE.FB_HIT_PS	100003	Retired load instructions which data sources were load missed...
<input checked="" type="checkbox"/>	MEM_LOAD_RETIRE.L1_HIT_PS	2000003	Retired load instructions with L1 cache hits as data sources
<input checked="" type="checkbox"/>	MEM_LOAD_RETIRE.L1_MISS_PS	100003	Retired load instructions missed L1 cache as data sources
<input checked="" type="checkbox"/>	MEM_LOAD_RETIRE.L2_HIT_PS	100003	Retired load instructions with L2 cache hits as data sources
<input checked="" type="checkbox"/>	MEM_LOAD_RETIRE.L3_HIT_PS	50021	Retired load instructions with L3 cache hits as data sources
<input checked="" type="checkbox"/>	MEM_LOAD_RETIRE.L3_MISS_PS	100007	Retired load instructions missed L3 cache as data sources

See Also

[Hardware Event-based Sampling Collection](#)

[HOW: Analysis Types](#)

Linux* and Android* Kernel Analysis

Use an event library provided in the Custom Analysis configuration to select Linux Ftrace* and Android* framework events to monitor with the event-based sampling collector.*

To choose events from the library:

1. Create a new hardware event-based sampling analysis type.

The new analysis type shows up under **Custom Analysis** in the **HOW** pane of the **Configure Analysis** window.

2. In the new custom configuration, use the **Linux Ftrace events** or **Android framework events** area to specify events for monitoring a system behavior:

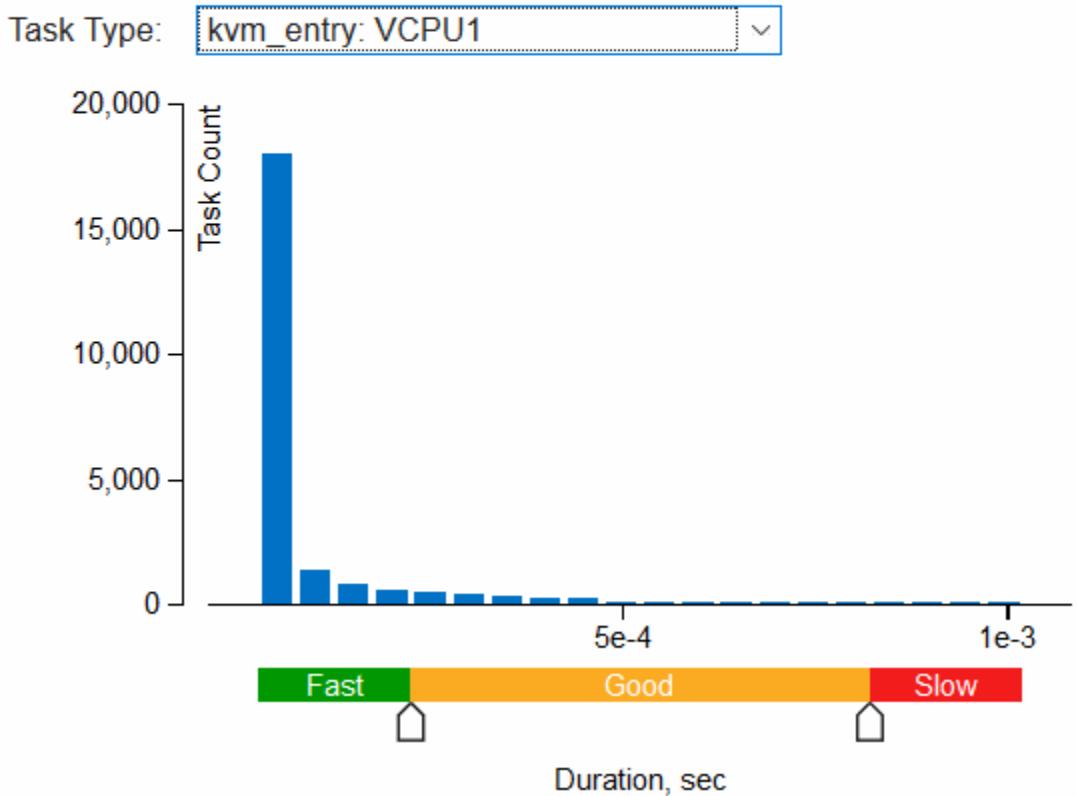
Linux Ftrace events		
<input type="text"/> Search..		
▼	Event Name	Description
<input checked="" type="checkbox"/>	Software Interrupts...	Trace Software Interrupts events to analyze Linux software in...
<input checked="" type="checkbox"/>	Hardware Interrupt...	Trace Hardware interrupts events to analyze Linux hardware ...
<input checked="" type="checkbox"/>	Linux Kernel Workq...	Trace Kernel Workqueues events to analyze Linux Kernel Wo...
<input checked="" type="checkbox"/>	CPU Idle (idle)	Trace CPU Idle events to analyze CPU sleep states transitions.
<input checked="" type="checkbox"/>	CPU Frequency (fr...	Trace CPU Frequency events to analyze CPU clock frequenc...
<input checked="" type="checkbox"/>	CPU Scheduling (s...	Trace CPU Scheduling events to analyze Linux Kernel Sched...
<input type="checkbox"/>	Intel Processor Gra...	Trace Intel Processor Graphics Scheduling events to analyze ...
<input type="checkbox"/>	Collect soft/hard pa...	Page Faults
<input type="checkbox"/>	File System Input a...	Trace File System events to analyze EXT4 filesystem operati...
<input type="checkbox"/>	Kernel-based Virtu...	Trace Kernel-based Virtual Machine events to analyze Linux ...
<input type="checkbox"/>	Synchronization M...	Trace Synchronization Manager events to analyze Android S...
<input type="checkbox"/>	Disk Input and Out...	Trace Disk Input and Output events to analyze block devices ...

For example, for [KVM guest OS profiling](#) consider selecting the following Linux Ftrace events to track IRQ injection process: kvm, irq, sofirq and workq.

The collected data shows up as tasks in the default viewpoint. Start with the **Summary** window to identify the most time-consuming tasks in the **Top Tasks** section. Analyze task duration statistics presented by task type in the **Task Duration** histogram:

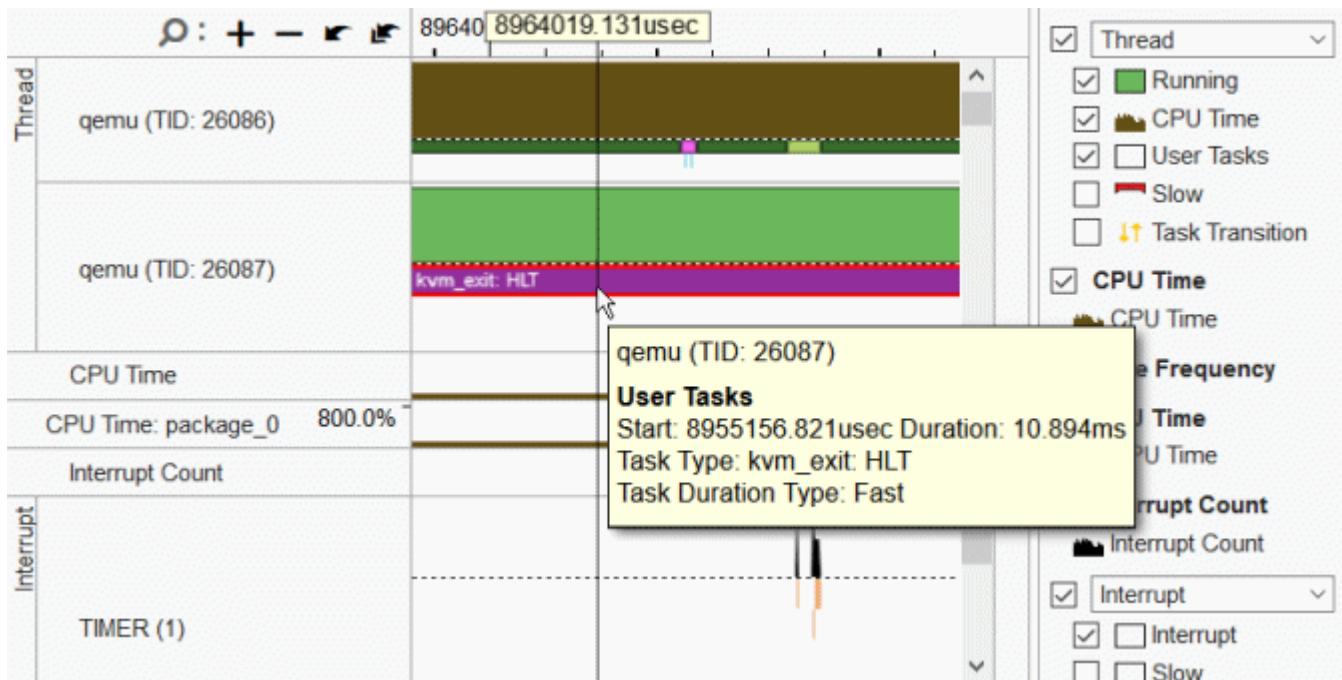
Task Duration Histogram

This histogram shows the total number of task instances executed with a specific duration. High number of slow instances may signal a performance bottleneck.



Use the sliders to set up thresholds for high and slow task instances.

Clicking a task in the **Top Tasks** section opens the **Bottom-up** window grouped by tasks. To analyze tasks over time, switch to the **Platform** window:



Limitations

On some systems, the Linux Ftrace subsystem, located in the debugfs partition in `/sys/kernel/debug/tracing`, may be accessible for the root user only. In this case, the VTune Profiler provides an error message: *Ftrace collection is not possible due to a lack of credentials. Root privileges are required.* To enable Ftrace events collection on such a system, you may either run the VTune Profiler with root privileges or change permissions manually by using the `chown` command under the root account, for example:

```
$ chown -R <user>:<group> /sys/kernel/debug/tracing
```

You can automate this process by using the `prepare_debugfs.sh` script located in the `bin` directory. The script mounts debugfs, changes permissions to a desired group, and updates the install boot script to apply this change automatically on the system startup. To execute this script, make sure to use root privileges.

To change permissions automatically with the `prepare_debugfs.sh` script, enter:

```
$ ./install/bin64/prepare-debugfs.sh [option]
```

where `[option]` is one of the following:

Option	Description
<code>-h --help</code>	Display usage information.
<code>-i --install</code>	Configure the autoload debugfs boot script and install it in the appropriate system directory.
<code>-c --check</code>	Mount without options, script will configure debugfs and check permissions.
<code>-u --uninstall</code>	Uninstall a previously installed debugfs boot script and revert configuration.
<code>-g --group <group></code>	Specify group other than vtune.
<code>-r --revert</code>	Revert debugfs configuration.

Option	Description
-b --batch	Run in a non-interactive mode (exiting in case of already changed permissions) without options. The script will configure debugfs.

See Also

[Custom Analysis](#)

[Task Analysis](#)

[Analyze Interrupts](#)

knob

atrace-config/ftrace-config option for CLI

[Problem: No GPU Utilization Data Is Collected](#)

Sampling Interval

Configure the amount of wall-clock time the Intel® VTune™ Profiler waits before collecting each sample (sampling interval).

The sampling interval is used to calculate the target number of samples and the [Sample After value](#) (SAV). Increasing the sampling interval may be useful for profiles with long durations or profiles that create large results. Typically, the size of the collected result is affected with such factors as duration, thread and core counts, selected analysis type, additional collection knobs, and application behavior.

You may change the default sampling interval as follows:

1. Click the



(standalone GUI)/



(Visual Studio* IDE) **Configure Analysis** button on the VTune Profiler toolbar.

2. Select a predefined analysis type from the **HOW** pane or [create a custom analysis type](#).
3. Use the **CPU sampling interval, ms** field to specify the required interval.

For [user-mode sampling and tracing types](#), specify a number (in milliseconds) between 1 and 1000. Default: 10ms. For [hardware event-based sampling types](#), specify a number between 0.01 and 1000. Default: 1ms.

NOTE

For hardware event-based sampling types, the sampling interval serves as a simple SAV multiplier so that the default interval value of 1ms just leaves the SAV intact. The sampling interval value of 0.1ms divides the SAV for all events by 10 making them overflow 10 times more frequently. The sampling interval value of 10ms multiplies the SAV for all events by 10 making them overflow 10 times less frequently.

To determine an appropriate sampling interval, consider the duration of the collection, the speed of your processors, and the amount of software activity. For instance, if the duration of sampling time is more than 10 minutes, consider increasing the sampling interval to 50 milliseconds. This reduces the number of interrupts and the number of samples collected and written to disk. The smaller the sampling interval, the larger the number of samples collected and written to disk.

The minimal value of the sampling interval for the user-mode sampling and tracing collection depends on the system:

- 10 milliseconds for Windows* systems with a single CPU
- 15 milliseconds for Windows* systems with multi-core CPUs
- 10 milliseconds for Linux* 2.4 kernels
- 1,2,4 milliseconds for new Linux >= 2.6 kernels depending on the vendor

NOTE

For [driverless Perf*-based data collection](#) on the targets running under Xen Hypervisor, the VTune Profiler automatically sets the sampling interval to 0 to switch to the integrated Perf sampling interval. This configuration provides more precise performance statistics in the hypervisor environment.

See Also

[knob sampling-interval](#)
[vtune option](#)

Sample After Value

For a custom event-based sampling data collection, set up the Sample After Value (SAV) that is a frequency with which the Intel® VTune™ Profiler interrupts the processor to collect a sample during hardware event-based data collection. SAV is measured as the number of events it takes to trigger a sample collection.

A Sample After Value that is too small causes the sampling interrupts to occur too frequently, which can lead to performance degradation and system instability. VTune Profiler enforces a floor value to prevent such a behavior. The recommended value is 1000 samples per second per processor.

VTune Profiler sets the Sample After value for hardware events automatically. For predefined hardware-level analysis types, the Sample After value is displayed in the **Configure Analysis** window > **HOW** pane. You cannot edit the Sample After value provided in the table for each event. But during the data collection the VTune Profiler may adjust it by a multiplier. The multiplier depends on the [sampling interval](#) value specified in the **HOW** pane.

To edit the default Sample After value, you need to create a custom hardware event-based analysis type (based on an existing type), [add events](#), if required, and edit a Sample After value in the events table by selecting it as follows:

Events configured for CPU: Intel(R) Processor code named Skylake ULT

NOTE: For analysis purposes, Intel VTune Amplifier 2018 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the target configuration window.

Events configured for CPU: Intel(R) Processor code named Skylake ULT			
Event Name Sample After Description			
<input checked="" type="checkbox"/>	BR_MISP_RETIREDA.LL_BRANCHES_PS	400009	Mispredicted macro branch instructions retired.
<input checked="" type="checkbox"/>	CPU_CLK_UNHALTED.REF_TSC	2400000	Reference cycles when the core is not in halt state.
<input checked="" type="checkbox"/>	CPU_CLK_UNHALTED.THREAD	2400000	Core cycles when the thread is not in halt state
<input checked="" type="checkbox"/>	CPU_CLK_UNHALTED.THREAD_P	2000003	Thread cycles when thread is not in halt state

See Also

[Custom Analysis Options](#)

Running runsa/runss Custom Analysis from the Command Line

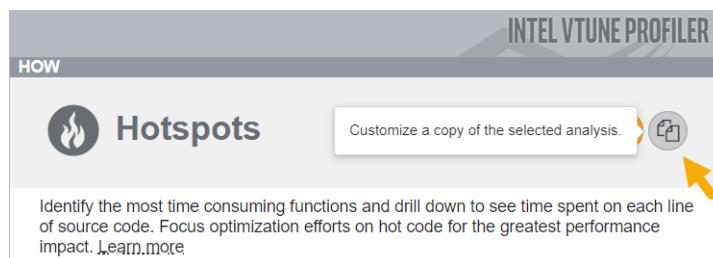
Energy Analysis

Use Intel® SoC Watch and Intel® VTune™ Profiler to collect and analyze power and energy consumption metrics. You can collect data on Windows, Linux, or Android systems. Use this data to identify system behaviors that waste energy.

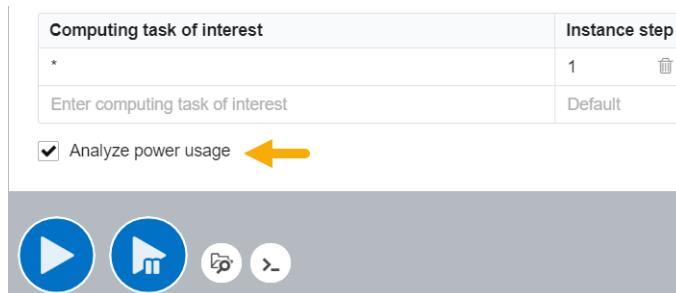
Get Snapshot through Intel® VTune™ Profiler

In the snapshot view, you can observe package power consumption over time when you run any analysis type in Intel® VTune™ Profiler. This option is available when you run VTune Profiler on Windows, Linux, or Android systems.

1. On the VTune Profiler welcome screen, click **Configure Analysis**.
2. In the **HOW** pane, select an analysis type. In this example, we use the Hotspots analysis.
3. Customize a copy of the analysis. Click this icon:



4. Select options as necessary.
5. At the bottom of the analysis type, check **Analyze power usage**.

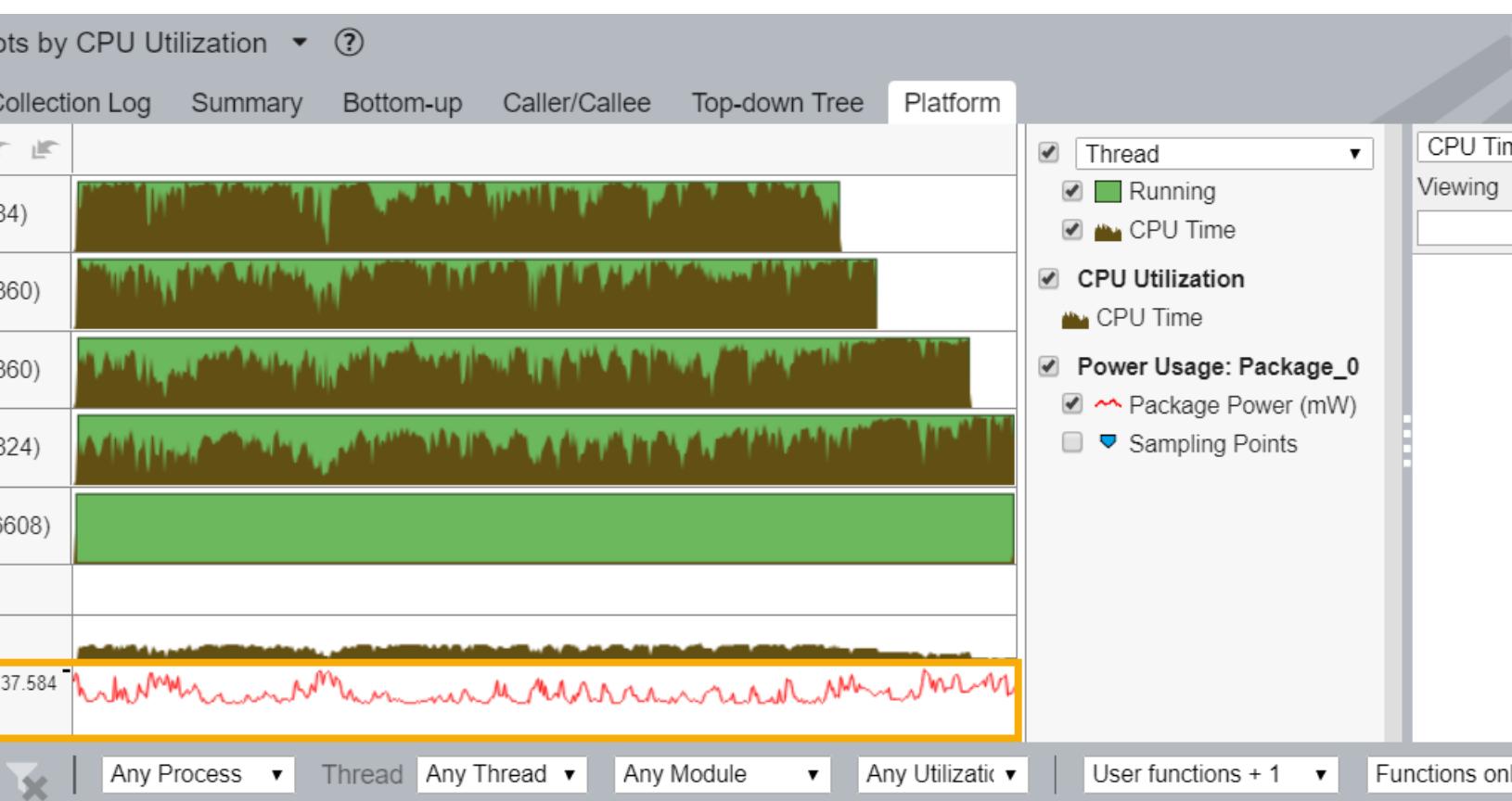


6. Click **Start(**



) to run the analysis.

When the analysis completes, VTune Profiler displays package power usage information (collected by Intel® SoC Watch) in the Platform tab.



Track package power usage to see if the CPU is likely to enter a throttling phase. If that happens, you can run a [throttling analysis](#) to explore possible causes.

For detailed information about using Intel SoC Watch, see the [Energy Analysis User Guide](#).

Run Energy Analysis

To analyze the power consumption of your Android*, Windows*, or Linux* platform, run the Intel® SoC Watch collector and view the results using Intel VTune Profiler.

Using the data visualization provided by Intel VTune Profiler, along with the detailed reports generated by Intel SoC Watch, a user can measure, debug, and optimize system power consumption. Data collection can occur on the system where Intel VTune Profiler is installed or on a remote target system.

Prerequisites: The Intel SoC Watch collector is installed on the target system. For detailed instructions on configuring your environment, see the *Installation* section of the Intel SoC Watch Release Notes for your target system's operating system. The latest Intel SoC Watch documents are [available online](#) at the Intel Developer Zone site. You can also find a copy on the target system in the product's documentation directory after extracting the Intel SoC Watch package.

1. Set up the scenario to be analyzed for energy usage and run the data collection using Intel SoC Watch, including the option to write a result file that can be imported to VTune Profiler (-f vtune). Data collection can occur on an idle system or run concurrently with a workload that is started at any time before or during the collection.

NOTE Users in Linux environments do not require root privileges to run energy analysis. Once your system administrator installs VTune Profiler sampling drivers and configures them with the necessary permissions, users without root privileges can collect energy data when profiling with VTune Profiler. On Windows systems, you must have administrator privileges to collect data on energy consumption.

For example, to run a collection for 1 minute (-t 60), gather data about how much time the CPU spends in low power states (-f cpu-cstate), include trace data (-m), and store the reports in a specified directory location with the specified file name (-o results/test), you would use:

```
socwatch -t 60 -f cpu-cstate -m -o results/test -r vtune
```

The import file is saved to the results directory as test.pwr.

For detailed descriptions of options and the different metrics that can be collected, see the *Getting Started* section of the Intel SoC Watch User's Guide ([Linux and Android](#) | [Windows](#)).

Tip

- Use feature group names as a shorthand for specifying several features (metrics) that should be collected at the same time. For instance, -f sys collects many commonly used metrics, including low power state residency for CPU, GPU, and devices, CPU temperature and frequency, and memory bandwidth.
 - Use the --help option to discover all of the available metrics that can be collected on the system (found under feature and feature group names) as well as other options for controlling data collection and reporting.
-

2. If running on a remote target system, copy the import file to the system where VTune Profiler is installed. The import file has a (*.pwr) extension, such as results/test.pwr from the example command.
3. Launch VTune Profiler.
4. Open



or create



a project.

5. Click the



Import Result button on the toolbar and browse to the import file that you copied from the target system.

When the import completes, the **Platform Power Analysis** viewpoint opens automatically.

See Also

[Interpret Energy Analysis Data with Intel® VTune™ Profiler](#)

[Run Command Line Analysis](#)

View Energy Analysis Data with Intel® VTune™ Profiler

NOTE

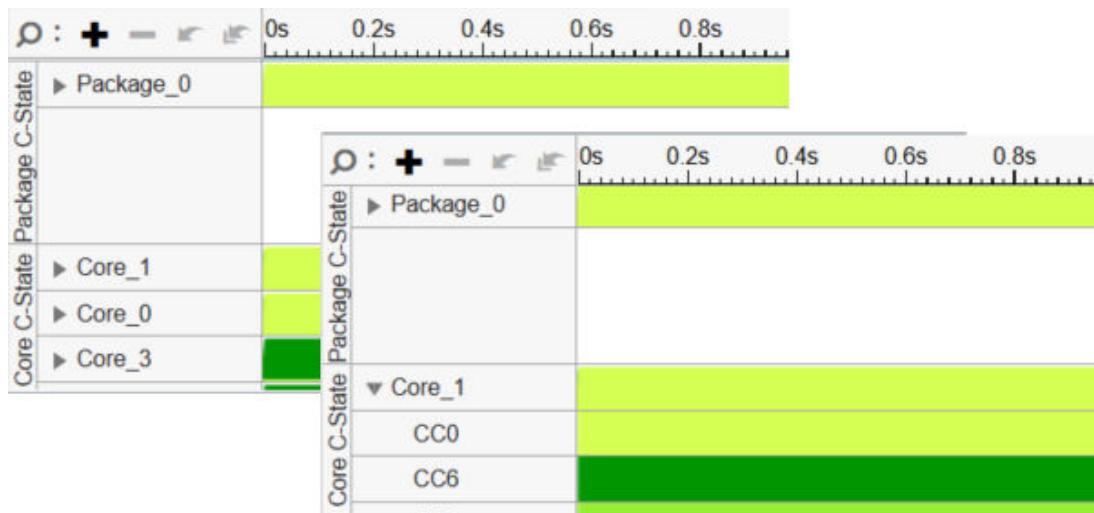
Collecting [energy analysis](#) data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

After you collect energy analysis data on your target system, using the Intel® SoC Watch collector, you can import a result file (*.pwr) to Intel® VTune™ Profiler on your host system, and view [Platform Power analysis](#) data with the following windows. The windows that appear depend on which metrics are collected:

- [Summary Window](#) displays a summary of the data collected. This window is a good starting point for identifying energy issues.
- [Correlate Metrics window](#) displays timelines for all collected data in the same time scale. This window is a good starting point for identifying energy issues.
- [Bandwidth window](#) displays the DDR SDRAM memory events and bandwidth usage over time.
- [Core Wake-ups window](#) displays wake-up events that caused the core to switch from a sleep state to an active state.
- [CPU C/P States window](#) displays CPU sleep state and processor frequency data correlated. The data is displayed according to the hierarchy for the platform on which the data was collected, and over time.
- [Graphics C/P States window](#) displays graphics sleep state, and P-state data collected. The data is displayed by device and over time.
- [NC Device window](#) displays the different D0ix sleep states for North Complex devices, overall counts and over time.
- [SC Device window](#) displays the different D0ix sleep states for South Complex devices, overall counts and over time.
- [Thermal Sample window](#) displays the temperature readings from the cores and SoC.
- [Timer Resolution](#) (Windows* OS only) displays the timer resolution and requests to change it, including the process requesting the change.
- [Wakelocks window](#) (Android* OS only) displays wakelock data indicating why the system can or cannot enter the ACPI S3 (Suspend-To-RAM) state.

View Component Rows in the Timeline Pane

Some rows can expand and reveal component rows. Look for an arrow next to the row name, as in the timeline shown below.



Zoom in on a Specific Section in the Timeline

Click and drag horizontally across the rows to select a time interval within the total collection. Release the mouse button to see a list of options for zooming in on this interval. **Zoom In and Filter In by Selection** is particularly useful because it will not only zoom, but also recalculate all of the grid's summary data based on the current selection. Once a filter has been applied in one tab it will persist across all tabs within that viewpoint, highlighting the selected time interval on each tab. Right-click and select **Remove All Filters** to restore the original grid and clear the selection from the timeline.



See Also

[Android* Targets](#)

[Remote Linux Target Setup](#)

[Collecting Data Remotely on Android* from Command Line](#)

[Search Directories](#)

[Manage Data Views](#)

Interpret Energy Analysis Data with Intel® VTune™ Profiler

Identify causes of energy waste on target systems by opening your energy analysis results with Intel VTune Profiler.

NOTE

Collecting [energy analysis](#) data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

After you collect energy analysis data on your target system, using the Intel® SoC Watch collector, you can import a result file (*.pwr) to Intel® VTune™ Profiler on your host system. Energy analysis data is opened in the [Platform Power analysis](#) viewpoint.

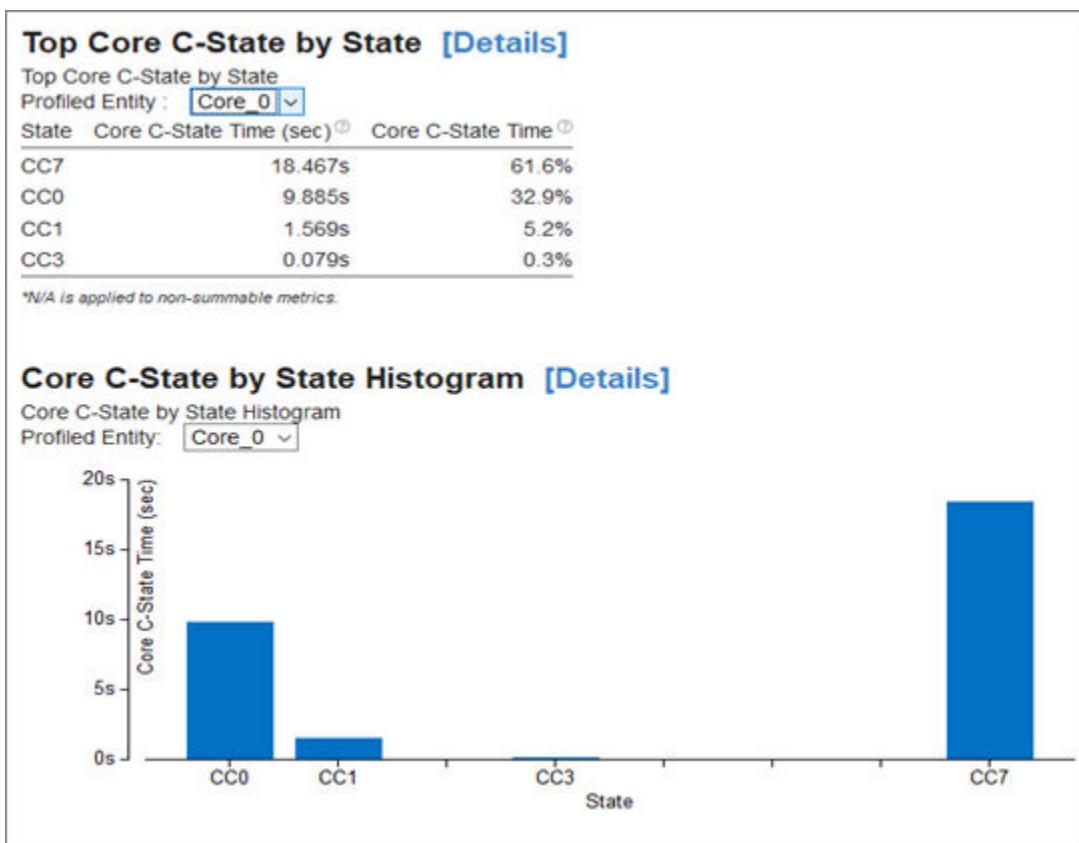
To interpret the performance data provided during the energy analysis, you may follow the steps below:

1. Analyze overall statistics.
2. Identify cores with the highest time spent in C0 state.

Analyze Overall Statistics

Use the [Summary window](#) to view statistics on the overall collection run time execution per power analysis metrics. Viewing the top statistics and histograms for a particular metric is a good starting point. Focus on decreasing the causes of core wake-up and increasing the time that the core spends in the higher sleep states.

For example, this **Top Core C-State by State** diagram shows that the core was awake for approximately 10 seconds of the total collection time (time in CC0 state). Use the **Profiled Entity** drop-down to view the C-state data for other cores. Explore the histogram to analyze the time spent in each sleep state.



Tip

Click the **Details** link next to the table or graph title on the **Summary** tab to view more information about that metric in another tab.

Identify Cores with Highest Time Spent in C0 States

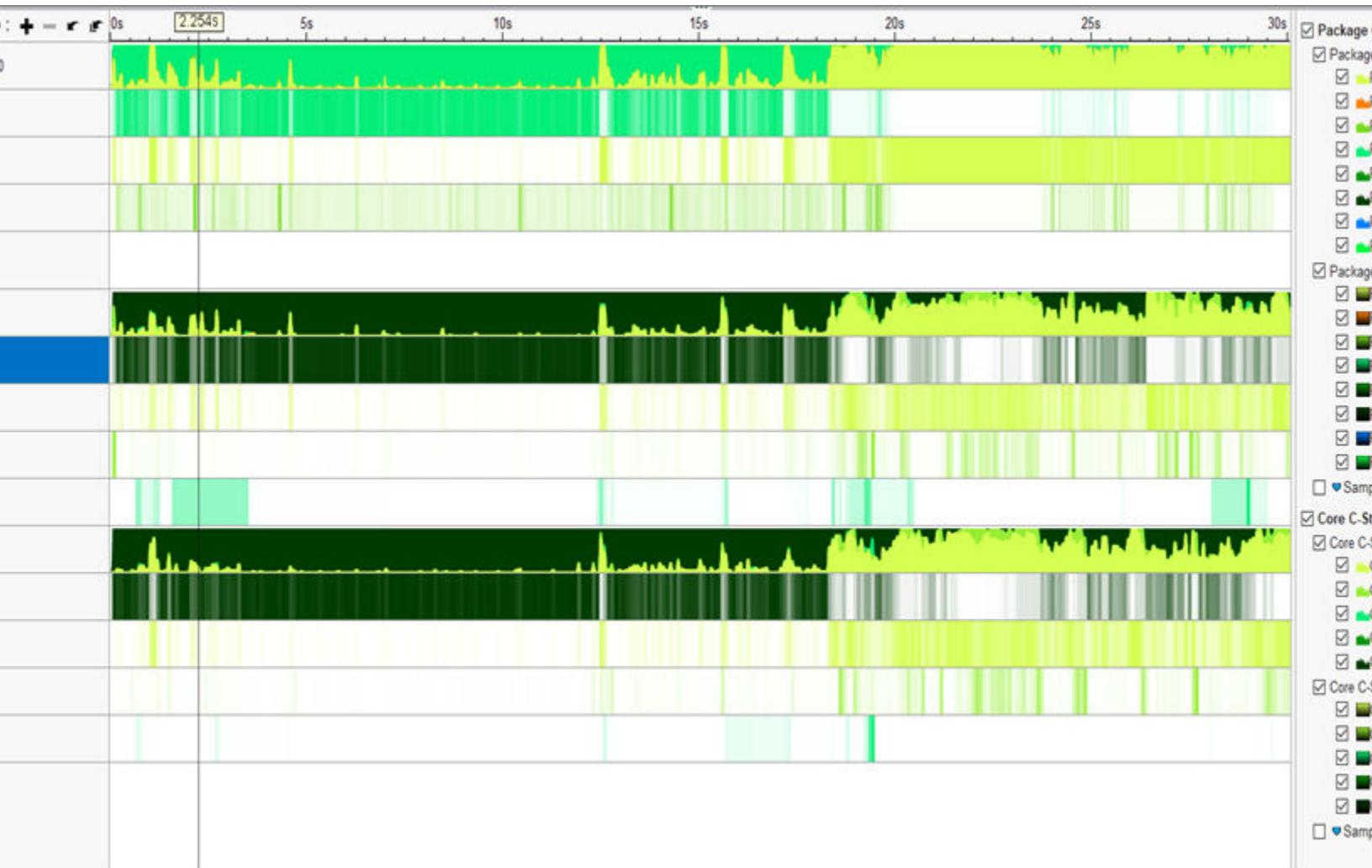
Switch to the C-State Residency/Wakeups window and the Core C-State tab to identify cores with the highest time spent in the active C0 state. Spending more time in deeper sleep states (C1-Cn) provides greater power savings.

By default, VTune Profiler displays data grouped by core and sorted by CPU time spent in the deepest C-state in the ascending order. For the example below, over 30% of the time was spent in the CC0 active state for both cores.

The screenshot shows the 'Platform Power Analysis viewpoint' in VTune Profiler. The top navigation bar includes 'Correlated Metrics', 'Temperature Metrics', and 'C-State Residency/Wakeups' (which is selected). Below the navigation bar, there are tabs for 'Package C-State' and 'Core C-State' (also selected). The main area is titled 'Grouping Profiled Entity Hierarchy'. A table titled 'Core C-State Time: Self by State' lists the percentage of time spent in various C-states for Core_0 and Core_1. The data is as follows:

Profiled Entity Hierarchy	Core C-State Time: Self by State				
	CC0	CC1	CC3	CC6	CC7
Total → SYSTEM → CPU → Package_0	0	0	0	0	0
Core_0	32.9%	5.2%	0.3%	0.0%	61.6%
Core_1	31.8%	4.4%	0.3%	0.0%	63.5%

Use the timeline view to understand when state transitions occur. Hover over a chart point to view the sleep states details for the particular moment of time. The deeper the color of the chart, the deeper the sleep state of the CPU. Select a region of the graph and zoom into the selection to see detailed sleep state transitions.



See Also

[Viewing Source](#)

Code Profiling Scenarios

Explore end-to-end performance analysis scenarios for managed code profiling and applications using Intel® runtime libraries:

- [Java* Code Analysis](#)
- [Python* Code Analysis](#)
- [Intel® Threading Building Blocks Code Analysis](#)
- [MPI Code Analysis](#)
- [OpenSHMEM* Code Analysis with Fabric Profiler](#)
- [GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)
- [Frame Data Analysis](#)
- [Task Analysis](#)

NOTE

For additional use cases, explore the Intel® VTune™ Profiler Performance Analysis Cookbook.

See Also

[Command Line Usage Scenarios](#)

Java* Code Analysis

Use the Intel® VTune™ Profiler to analyze Java applications executed with Oracle* or OpenJDK* (Linux* only).*

Even though Java code execution is handled with a Managed Runtime Environment, it can be as ineffective in terms of data management as in programs written using native languages. For example, if you are conscious about performance of your data mining Java application, you need to take into consideration your target platform memory architecture, cache hierarchy and latency of access to memory levels. From the platform microarchitecture point of view, profiling of Java applications is similar to profiling of native applications but with one major difference: to see performance metrics against their program source code, the profiling tool must be able to map metrics of the binary code either compiled or interpreted by the JVM back to the original source code in Java or C/C++.

VTune Profiler provides a low-overhead analysis of the JIT compiled code that is available for both user-mode sampling and tracing and hardware event-based sampling analysis types. The analysis of the interpreted Java methods is [limited](#).

To enable the Java code analysis with the Intel® VTune™ Profiler and interpret data:

- [Configure Java data collection](#).
 - [Launch Application](#)
 - [Attach to Process](#)
 - Linux* only: [Attach to Process Running under Low-privilege Account](#)
- [Identify hottest methods](#).
- [Analyze stacks for mixed code](#).
- [Analyze hardware metrics](#).
- [Understand limitations](#).

Configuring Java Data Collection

To configure your performance analysis for Java code, you may use either GUI or command line (`vtune`) configuration. You may run Java code analysis using one of the following modes:

To configure Java analysis in the Launch Application mode:

1. Embed your `java` command in a batch file or executable script.

For example, create a `run.bat` file on Windows* or `run.sh` file on Linux* with the following command:

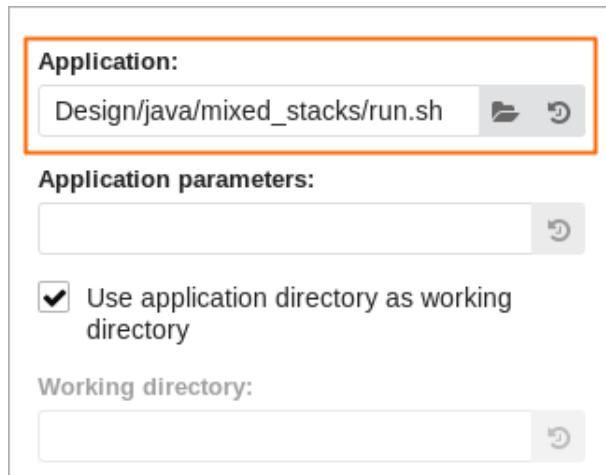
Windows:

```
> java.exe -Xcomp -Djava.library.path=native_lib\ia32 -cp C:\Design\Java\mixed_stacks
MixedStacksTest 3 2
```

Linux:

```
$ java -Xcomp -Djava.library.path=native_lib/ia32 -cp /home/Design/Java/mixed_stacks
MixedStacksTest 3 2
```

2. Create a [project](#).
3. In the **Configure Analysis** window > **WHERE** pane, specify your analysis system, for example, **Local Host**.
4. In the **WHAT** pane, choose the **Application to Launch** target type.
5. In the **Application** field, specify a path to this `run` file . For example, on Linux:



- 6.** In the **Advanced** section, select the **Auto Managed code profiling** mode and enable the **Analyze child processes** option.

Similarly, you can configure an analysis with the VTune Profiler command line interface, `vtune`. For example, for the Hotspots analysis on Linux run the following command line:

```
$ vtune -collect hotspots -- run.sh
```

or directly:

```
$ vtune -collect hotspots -- java -Xcomp -Djava.library.path=native_lib/ia32 -cp home/Design/Java/mixed_stacks MixedStacksTest 3 2
```

To configure Java analysis in the Attach to Process mode:

In case your Java application needs to run for some time or cannot be launched at the start of this analysis, you may attach the VTune Profiler to the standalone Java process. On Linux, you can also attach the VTune Profiler to a C/C++ application with an embedded JVM instance for hardware event-based sampling analysis types. To do this, select the **Attach to Process** target type in the **WHAT** pane and specify the `java` process name or PID.

You may use the command line interface to attach the analysis to the Java process. For example, the following command attaches the Hotspots analysis to the Java process:

```
$ vtune -collect hotspots -target-process java
```

The following command line example attaches the Hotspots analysis to the Java process by its PID:

```
$ vtune -collect hotspots -target-pid 1234
```

NOTE The dynamic attach mechanism is supported only with the Java Development Kit (JDK).

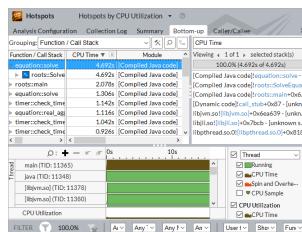
To configure Java analysis in the Attach to Process mode under Low-privilege Account (Linux* Only):

For hardware event-based sampling analysis types, you can attach the VTune Profiler running under the superuser account to a Java process or a C/C++ application with embedded JVM instance running under a low-privileged user account. For example, you may attach the VTune Profiler to Java based daemons or services.

To do this, run the VTune Profiler under the root account, select the **Attach to Process** target type and specify the `java` process name or PID.

Identifying Hottest Methods

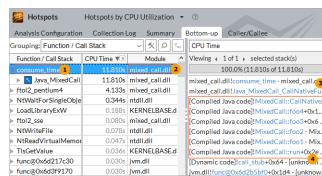
You may run the Hotspot analysis to get a list of the hottest methods along with their timing metrics and call stacks. The workload distribution over threads is also displayed in the **Timeline** pane. Thread naming helps to identify where exactly the most resource consuming code was executed. For example, on Linux*:



Analyzing Stacks for Mixed Code

If you are pursuing maximum performance on a platform, consider writing and compiling performance critical modules of your Java project in native languages like C or even assembly. This way of programming helps to employ powerful CPU resources like vector computing (implemented via SIMD units and instruction sets). In this case, compute-intensive functions become hotspots in the profiling results, which is expected as they do most of the job. However, you might be interested not only in hotspot functions, but in identifying locations in Java code these functions were called from via a JNI interface. Tracing such cross-runtime calls in the mixed language algorithm implementations could be a challenge.

To analyze mixed code profiling results, the VTune Profiler is "stitching" the Java call stack with the subsequent native call stack of C/C++ functions. The reverse call stacks stitching works as well. For example, on Windows*:



- 1 Native function
- 3 Mixed native/Java call stack
- 2 Native module
- 4 Compiled methods in the Java call stack

NOTE

Due to [inlining](#) during the compilation stage, some functions may not appear in the stack by default. Make sure to select the **Show inline functions** option for the **Inline Mode** on the filter bar.

Analyzing Hardware Metrics

VTune Profiler also provides an advanced profiling option of optimizing Java applications for the CPU microarchitecture utilized in your platform. Although Java and JVM technology is intended to free a developer from hardware architecture specific coding, once Java code is optimized for the current Intel microarchitecture, it will most probably keep this advantage for future generations of CPUs. You may use the [hardware event-based sampling](#) data collection that monitors hardware events in the CPU's pipeline and can identify coding pitfalls limiting the most effective execution of instructions in the CPU. The [CPU metrics](#) are available and can be displayed against the application modules, functions, and Java code source lines. You may also run the [hardware event-based sampling collection with stacks](#) when you need to find out a call path for a function called in a driver or middleware layer in your system.

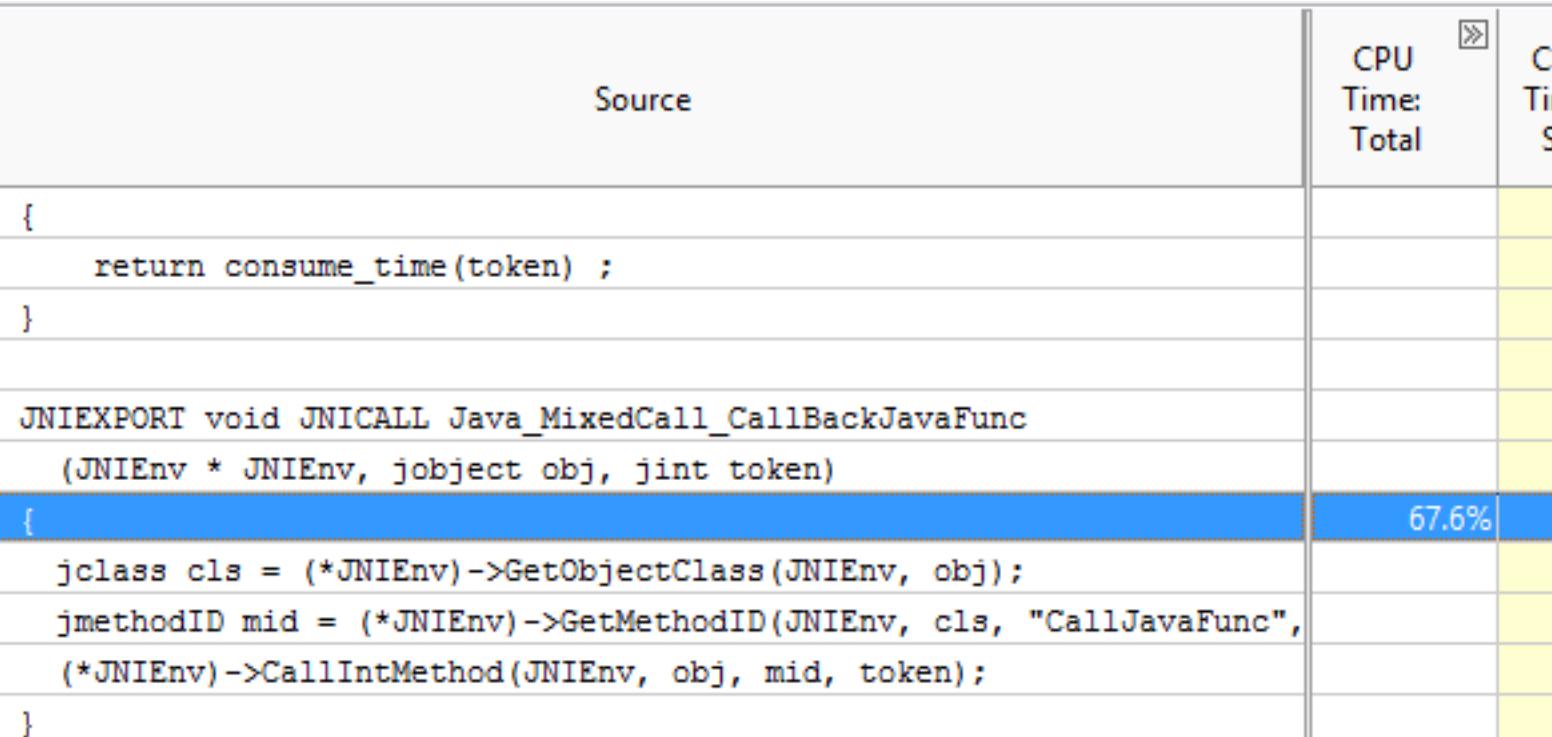
Limitations

VTune Profiler supports analysis of Java applications with some limitations:

- System-wide profiling is not supported for managed code.
- The JVM interprets some rarely called methods instead of compiling them for the sake of performance. VTune Profiler does not recognize interpreted Java methods and marks such calls as !Interpreter in the restored call stack.

If you want such functions to be displayed in stacks with their names, force the JVM to compile them by using the `-Xcomp` option (show up as [Compiled Java code] methods in the results). However, the timing characteristics may change noticeably if many small or rarely used functions are being called during execution.

- When opening source code for a hotspot, the VTune Profiler may attribute events or time statistics to an incorrect piece of the code. It happens due to JDK Java VM specifics. For a loop, the performance metric may slip upward. Often the information is attributed to the first line of the hot method's source code. In the example below, a real hotspot line consuming most CPU time is line 35.



- Consider events and time mapping to the source code lines as approximate.
- For the Hotspots analysis type in the user-mode sampling mode, the VTune Profiler may display only a part of the call stack. To view the complete stack on Windows, use the `-Xcomp` additional command line JDK Java VM option that enables the JIT compilation for better quality of stack walking.

To view the complete stack on Linux, use additional command line JDK Java VM options that change behavior of the Java VM:

- Use the `-Xcomp` additional command line JDK Java VM option that enables the JIT compilation for better quality of stack walking.
- On Linux* x86, use client JDK Java VM instead of the server Java VM: either explicitly specify `-client`, or simply do not specify `-server` JDK Java VM command line option.
- On Linux x64, specify `-XX:-UseLoopCounter` command line option that switches off on-the-fly substitution of the interpreted method with the compiled version.

- Java application profiling is supported for the Hotspots and Microarchitecture analysis types. Support for the Threading analysis is limited as some embedded Java synchronization primitives (which do not call operating system synchronization objects) cannot be recognized by the VTune Profiler. As a result, some of the timing metrics may be distorted.
- There are no dedicated libraries supplying a user API for collection control in the Java source code. However, you may want to try applying the native API by wrapping the `_itt` calls with JNI calls.

See Also

[Enable Java* Analysis on Android* System](#)

[Stitch Stacks for Intel® oneAPI Threading Building Blocks or OpenMP* Analysis](#)

Python* Code Analysis

Explore performance analysis options provided by the Intel® VTune™ Profiler for Python applications to identify the most time-consuming code sections and critical call paths.*

VTune Profiler supports the **Hotspots**, **Threading**, and **Memory Consumption (Linux only)** analyses for Python* applications through the **Launch Application** mode. For example, when your application does excessive numerical modeling, you need to know how effectively it uses available CPU resources. A good example of the effective CPU usage is when the calculating process spends most time executing native extension and not interpreting Python glue code.

To get the maximum performance out of your Python application, consider using native extensions, such as NumPy or writing and compiling performance critical modules of your Python project in native languages, such as C or even assembly. This will help your application take advantage of vectorization and make complete use of powerful CPU resources.

To analyze the Python code performance with the VTune Profiler and interpret data:

- [Configure Python data collection](#)
- [Identify hot spots](#)
- [Understand limitations](#)

Configure Python Data Collection

Configure VTune Profiler through the GUI or command-line (`vtune`) interface to analyze the performance of your Python code.

In the GUI:

- Click the



Configure Analysis button on the toolbar.

The [Configure Analysis](#) window opens.

- Choose a target system and target type, like **Local Host** and **Launch Application**.

NOTE

You can profile Windows* and Linux* target systems only.

- In the **WHERE** pane, provide these details:

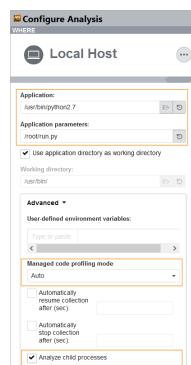
- In the **Application** field, enter the path to the installed Python interpreter.

- In the **Application parameters** field, enter the path to your Python script.

NOTE

If you specify a relative path to your Python script, VTune Profiler completely resolves the full function or method names for the imported modules only. The names inside the main script are not resolved. To avoid this, specify the absolute path to your Python script.

- In the **Advanced** settings, in the **Managed code profiling mode** drop-down menu, select **Auto**. This way, VTune Profiler automatically detects the type of target executable (managed or native) and switches to the corresponding mode.
- If necessary, select **Analyze child processes** to collect data on processes launched by the target process.



NOTE

When you attach the VTune Profiler to the Python process, make sure you initialize the Global Interpreter Lock (GIL) inside your script before you start the analysis. If GIL is not initialized, the VTune Profiler collector initializes it only when a new Python function is called.

4. If your Python application should run before you start profiling or if you cannot run the application at the start of the analysis, attach VTune Profiler to the Python process. To do this, in the **WHAT** pane, select the **Attach to Process** target type. Specify the Python process name or PID.



5. In the **HOW** pane on the right, select the **Hotspots**, **Threading**, or **Memory Consumption** analysis type.
6. If necessary, configure these options or use their default settings:

User-Mode Sampling mode	Select to enable the user-mode sampling and tracing collection for hotspots and call stack analysis (formerly known as Basic Hotspots). This collection mode uses a fixed sampling interval of 10ms. If you need to change the interval, click the Copy button and create a custom analysis configuration.
Show additional performance insights check box	Get additional performance insights, such as vectorization, and learn next steps. This option collects additional CPU events, which may enable the multiplexing mode. The option is enabled by default.

Details button

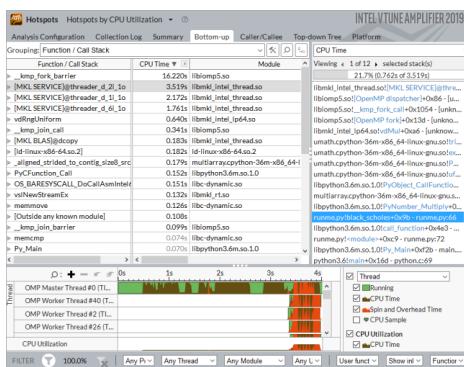
Expand/collapse a section listing the default non-editable settings used for this analysis type. If you want to modify or enable **additional settings** for the analysis, you need to [create a custom configuration](#) by copying an existing predefined configuration. VTune Profiler creates an editable copy of this analysis type configuration.

- Click **Start** to run the analysis.

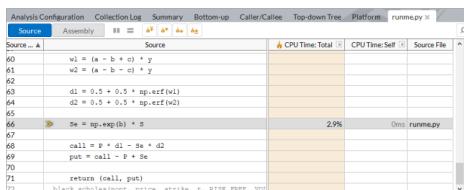
Identifying Hot Spots

Hotspots analysis in the user-mode sampling mode helps identify sections of your Python code that take a long time to execute (hotspots), along with their timing metrics and call stacks. It also displays the workload distribution over threads in the [Timeline pane](#).

By default, the VTune Profiler uses the **Auto** managed code profiling mode, that enables you to view and analyze mixed stacks for Python/C++ applications. In the example below, you can see a native hotspot Intel® oneAPI Math Kernel Library(oneMKL) function on the left pane. The mixed call stack analysis on the right pane reveals a Python `black_scholes` function that actually calls the hotspot function:



Double-click the `black_scholes` function on the **Call Stack** pane to open the source view on call site line 66:



To view call stacks only inside your Python code, filter out Python core and system functions by selecting **Only user functions** option for the **Call Stack Mode** on the filter bar.

Python Code Profiling Considerations

- Profiling support exists for Python distribution 2.6 and newer versions.
- If you use Python extensions that compile Python code to the native language (JIT, C/C++), VTune Profiler may show incorrect analysis results. Consider using [JIT Profiling API](#) to solve this problem.
- You can profile Python code on Windows and Linux target systems.
- In some cases, VTune Profiler may not resolve full names of Python functions and modules on Windows OS. However, the source information displays properly. You can view the source directly from viewpoints in VTune Profiler.
- The Timeline pane does not always display proper thread names.
- If your application has very low stack depth, which includes called functions and imported modules, the VTune Profiler does not collect Python data. Consider using deeper calls to enable the profiling.

- When collecting data remotely, VTune Profiler may not resolve full function or method names, and display the source code of your Python script. To solve this problem for Linux targets, copy the source files to a directory on your host system with a path identical to the path on your target system before running the analysis.

See Also

knob

`mrtpe-type=python` option

Hotspots View

Memory Consumption and Allocations View

Intel® Threading Building Blocks Code Analysis

Use the Intel® VTune™ Profiler for performance analysis of application targets using Intel® oneAPI Threading Building Blocks(oneTBB).

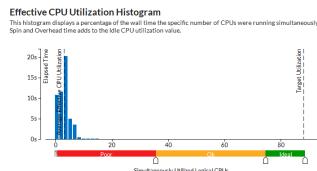
If you used the Intel® Runtime libraries in your application, you can run:

- Hotspots and Threading analysis to explore the application parallelization efficiency based on oneTBB parallel or synchronization constructs.
- Threading analysis to get detailed information on oneTBB synchronization objects that limited the parallel performance of your multithreaded application.

NOTE

Using Intel C++ compiler is recommended to get more comprehensive diagnostics from the VTune Profiler.

Start exploration of oneTBB parallelization efficiency with Hotspots. Look at the **Effective CPU Utilization Histogram** to see the parallelization level of your application. Note that the histogram reflects the parallelization levels of your application based on the effective time spent subtracting time spent in threading runtimes.



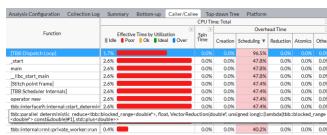
If you see a significant portion of your elapsed time spent with Idle or Poor CPU utilization, explore the **Top Hotspots** table. Flagged oneTBB functions might mean that the application spends CPU time in the oneTBB runtime because of parallel inefficiencies like scheduling overhead or imbalance. To discover the reason, hover over the flag.

Top Hotspots		
This table lists the most active functions in your application. Optimizing these hotspot functions typically results in improved application performance.		
Function	Module	Time
[TBB Scheduler] [TBB Scheduler Thread]	libtbb.so.0.2	300.241*
[TBB Scheduler Internal]	libtbb.so.0.2	154.166
[TBB Scheduler Internal]	libtbb.so.0.2	144.544
[TBB Parallel Data-Parallel Reducer]	libtbb.so.0.2	112.005
[TBB Parallel Data-Parallel Reducer Body]	vector-reduce	108.695
[Others]		807.511*

*Not applicable to non-executable memory.

The **Bottom-up** tab can give you more details about synchronization or overhead in particular oneTBB constructs. Expand the **Spin Time** and **Overhead Time** columns in the grid to determine why a particular oneTBB runtime function had a higher than usual execution time. oneTBB runtime functions are flagged when they consume more than 5% of the CPU time.

For example, an oneTBB runtime function with a high Scheduling value may indicate that your application has threading work divided into small pieces, which leads to excessive scheduling overhead as the application calls to the runtime. You can resolve this issue by increasing the threading chunk size.



If there is an idle wait time when the oneTBB runtime does not burn the CPU on synchronization, it is useful to run the Threading analysis to explore synchronization bottlenecks that can prevent effective CPU utilization. VTune Profiler recognizes all types of Intel TBB synchronization objects. If you assign a meaningful name to an object you create in the source code, the VTune Profiler recognizes and represents it in the Result tab. For performance reasons, this functionality is not enabled by default in oneTBB headers. To make the user-defined objects visible to the VTune Profiler, recompile your application with **TBB USE THREADING TOOLS** set to 1.

To display an overhead introduced by oneTBB library internals, the VTune Profiler creates a pseudo synchronization object `TBB Scheduler` that includes all waits from the oneTBB runtime libraries.

See Also

Cookbook: OpenMP* Code Analysis Method

Threading Efficiency View

Cookbook: Scheduling Overhead in oneTBB Apps

MPI Code Analysis

Explore using Intel® VTune™ Profiler command line interface (vtune) for profiling an MPI application.

Parallel High Performance Computing (HPC) applications often rely on multi-node architectures of modern clusters. Performance tuning of such applications must involve analysis of cross-node application behavior as well as single-node performance analysis. Intel® Parallel Studio Cluster Edition includes such performance analysis tools as [Application Performance Snapshot](#), Intel Trace Analyzer and Collector, and Intel VTune Profiler that can provide important insights to help in MPI application performance analysis. For example:

- Application Performance Snapshot provides a quick MPI application performance overview.
 - Intel Trace Analyzer and Collector explores message passing interface (MPI) usage efficiency with communication hotspots, synchronization bottlenecks, load balancing, etc.
 - Intel VTune Profiler focuses on intra-node performance with threading, memory, and vectorization efficiency metrics.

NOTE

NOTE The version of the Intel MPI library included with the Intel Parallel Studio Cluster Edition makes an important switch to use the Hydra process manager by default for `mpirun`. This provides high scalability across the big number of nodes.

This topic focuses on how to use the VTune Profiler command line tool to analyze an MPI application. Refer to the [Additional Resources](#) section below to learn more about other analysis tools.

Use the VTune Profiler for a single-node analysis including threading when you start analyzing hybrid codes that combine parallel MPI processes with threading for a more efficient exploitation of computing resources. [HPC Performance Characterization analysis](#) is a good starting point to understand CPU utilization, memory access, and vectorization efficiency aspects and define the tuning strategy to address performance gaps. The CPU Utilization section contains the MPI Imbalance metric, which is calculated for MPICH-based MPIs. Further steps might include Intel Trace Analyzer and Collector to look at MPI communication efficiency, [Memory Access analysis](#) to go deeper on memory issues, [Microarchitecture Exploration analysis](#) to explore microarchitecture issues, or Intel Advisor to dive into vectorization tuning specifics.

Use these basic steps required to analyze MPI applications for imbalance issues with the VTune Profiler:

1. Configure installation for MPI analysis on Linux host.
 2. Configure and run MPI analysis with the VTune Profiler.

- [3. Control collection with the MPI_Pcontrol function.](#)
- [4. Resolve symbols for MPI modules.](#)
- [5. View collected data.](#)

Explore additional information on MPI analysis:

- [• MPI implementations supported by VTune Profiler](#)
- [• MPI system modules recognized by VTune Profiler](#)
- [• Analysis limitations](#)
- [• Additional resources](#)

Configure Installation for MPI Analysis on Linux* Host

For MPI application analysis on a Linux* cluster, you may enable the **Per-user Hardware Event-based Sampling** mode when installing the Intel Parallel Studio Cluster Edition. This option ensures that during the collection the VTune Profiler collects data only for the current user. Once enabled by the administrator during the installation, this mode cannot be turned off by a regular user, which is intentional to preclude individual users from observing the performance data over the whole node including activities of other users.

After installation, you can use the respective `vars.sh` files to set up the appropriate environment (PATH, MANPATH) in the current terminal session.

Configure MPI Analysis with the VTune Profiler

To collect performance data for an MPI application with the VTune Profiler, use the command line interface (`vtune`). The collection configuration can be completed with the help of the target configuration options in the VTune Profiler user interface. For more information, see [Arbitrary Targets Configuration](#).

Usually, MPI jobs are started using an MPI launcher such as `mpirun`, `mpiexec`, `srun`, `aprun`, etc. The examples provided use `mpirun`. A typical MPI job uses the following syntax:

```
mpirun [options] <program> [<args>]
```

VTune Profiler is launched using `<program>` and your application is launched using the VTune Profiler command arguments. As a result, launching an MPI application using VTune Profiler uses the following syntax:

```
mpirun [options] vtune [options] <program> [<args>]
```

There are several options for `mpirun` and `vtune` that must be specified or are highly recommended while others can use the default settings. A typical command uses the following syntax:

```
mpirun -n <n> -l vtune -quiet -collect <analysis_type> -trace-mpi -result-dir <my_result> my_app [<my_app_options>]
```

The `mpirun` options include:

- `<n>` is the number of MPI processes to be run.
- `-l` option of the `mpiexec/mpirun` tools marks stdout lines with an MPI rank. This option is recommended, but not required.

The `vtune` options include:

- `-quiet / -q` option suppresses the diagnostic output like progress messages. This option is recommended, but not required.
- `-collect <analysis type>` is an analysis type you run with the VTune Profiler. To view a list of available analysis types, use `VTune Profiler-help collect` command.
- `-trace-mpi` adds a per-node suffix to the result directory name and adds a rank number to a process name in the result. This option is required for non-Intel MPI launchers.
- `-result-dir <my_result>` specifies the path to a directory in which the analysis results are stored.

If a MPI application is launched on multiple nodes, VTune Profiler creates a number of result directories per compute node in the current directory, named as `my_result.<hostname1>`, `my_result.<hostname2>`, ..., `my_result.<hostnameN>`, encapsulating the data for all the ranks running on the node in the same directory. For example, the Hotspots analysis (hardware event-based sampling mode) run on 4 nodes collects data on each compute node:

```
mpirun -n 16 -ppn 4 -l vtune -collect hotspots -k sampling-mode=hw -trace-mpi -result-dir my_result -- my_app.a
```

Each process data is presented for each node they were running on:

```
my_result.host_name1 (rank 0-3)
my_result.host_name2 (rank 4-7)
my_result.host_name3 (rank 8-11)
my_result.host_name4 (rank 12-15)
```

If you want to profile particular ranks (for example, outlier ranks defined by Application Performance Snapshot), use selective rank profiling. Use multi-binary MPI run and apply VTune Profiler profiling for the ranks of interest. This significantly reduces the amount of data required to process and analyze. The following example collects Memory Access data for 2 out of 16 processes with 1 rank per node:

```
export VTUNE_CL=vtune -collect memory-access -trace-mpi -result-dir my_result
mpirun -host myhost1 -n 7 my_app.a : -host myhost1 -n 1 $VTUNE_CL -- my_app.a :-host myhost2 -n 7 my_app.a : -host myhost2 -n 1 $VTUNE_CL -- my_app.a
```

Alternatively, you can create a configuration file with the following content:

```
# config.txt configuration file
-host myhost1 -n 7 ./a.out
-host myhost1 -n 1 vtune -quiet -collect memory-access -trace-mpi -result-dir my_result ./a.out
-host myhost2 -n 7 ./a.out
-host myhost2 -n 1 vtune -quiet -collect memory-access -trace-mpi -result-dir my_result ./a.out
```

To run the collection using the configuration file, use the following command:

```
mpirun -configfile ./config.txt
```

If you use Intel MPI with version 5.0.2 or later you can use the `-gtool` option with the Intel MPI process launcher for easier selective rank profiling:

```
mpirun -n <n> -gtool "vtune -collect <analysis type> -r <my_result>:<rank_set>"<my_app> [my_app_options]
```

where `<rank_set>` specifies a ranks range to be involved in the tool execution. Separate ranks with a comma or use the `"-"` symbol for a set of contiguous ranks.

For example:

```
mpirun -gtool "vtune -collect memory-access -result-dir my_result:7,5" my_app.a
```

Examples:

1. This example runs the HPC Performance Characterization analysis type (based on the sampling driver), which is recommended as a starting point:

```
mpirun -n 4 vtune -result-dir my_result -collect hpc-performance -- my_app [my_app_options]
```

2. This example collects the Hotspots data (hardware event-based sampling mode) for two out of 16 processes run on myhost2 in the job distributed across the hosts:

```
mpirun -host myhost1 -n 8 ./a.out : -host myhost2 -n 6 ./a.out : -host myhost2 -n 2 vtune -result-dir foo -c hotspots -k sampling-mode=hw ./a.out
```

As a result, the VTune Profiler creates a result directory in the current directory `foo.myhost2` (given that process ranks 14 and 15 were assigned to the second node in the job).

3. As an alternative to the previous example, you can create a configuration file with the following content:

```
# config.txt configuration file
-host myhost1 -n 8 ./a.out
-host myhost2 -n 6 ./a.out
-host myhost2 -n 2 vtune -quiet -collect hotspots -k sampling-mode=hw -result-dir foo ./a.out
```

and run the data collection as:

```
mpirun -configfile ./config.txt
```

to achieve the same result as in the previous example: `foo.myhost2` result directory is created.

4. This example runs the Memory Access analysis with memory object profiling for all ranks on all nodes:

```
mpirun n 16 -ppn 4 vtune -r my_result -collect memory-access -knob analyze-mem-objects=true -
my_app [my_app_options]
```

5. This example runs Hotspots analysis (hardware event-based sampling mode) on ranks 1, 4-6, 10:

```
mpirun -gtool "vtune -r my_result -collect hotspots -k sampling-mode=hw : 1,4-6,10" -n 16 -ppn 4
my_app [my_app_options]
```

NOTE

The examples above use the `mpirun` command as opposed to `mpiexec` and `mpiexec.hydra` while real-world jobs might use the `mpiexec*` ones. `mpirun` is a higher-level command that dispatches to `mpiexec` or `mpiexec.hydra` depending on the current default and options passed. All the listed examples work for the `mpiexec*` commands as well as the `mpirun` command.

Control Collection with Standard MPI_Pcontrol Function

By default, VTune Profiler collects statistics for the whole application run. In some cases, it is important to enable or disable the collection for a specific application phase. For example, you may want to focus on the most time consuming section or disable collection for the initialization or finalization phases. This can be done with VTune Profiler [instrumentation and tracing technology](#) (ITT). Starting with the Intel VTune Profiler 2019 Update 3 version, VTune Profiler provides ability to control data collection for MPI application with the help of standard `MPI_Pcontrol` function.

Common syntax:

- Pause data collection: `MPI_Pcontrol(0)`
- Resume data collection: `MPI_Pcontrol(1)`
- Exclude initialization phase: Use with the `VTune Profiler-start-paused` option by adding the `MPI_Pcontrol(1)` call right after initialization code completion. Unlike with ITT API calls, using the `MPI_Pcontrol` function to control data collection does not require a link to a profiled application with a static ITT API library and therefore changes in the build configuration of the application.

Resolve Symbols for MPI Modules

After data collection, the VTune Profiler automatically finalizes the data (resolves symbols and converts them to the database). It happens on the same compute node where the command line collection was executing. So, the VTune Profiler automatically locates binary and symbol files. In cases where you need to point to symbol files stored elsewhere, adjust the search settings using the `-search-dir` option:

```
mpirun -np 128 vtune -q -collect hotspots -search-dir /home/foo/syms ./a.out
```

View Collected Data

Once the result is collected, you can open it in the graphical or command line interface of the VTune Profiler.

To view the results in the command line interface:

Use the `-report` option. To get the list of all available VTune Profiler reports, enter `VTune Profiler-help report`.

To view the results in the graphical interface:

Click the



menu button and select **Open > Result...** and browse to the required result file (*.vtune).

Tip

You may copy a result to another system and view it there (for example, to open a result collected on a Linux* cluster on a Windows* workstation).

VTune Profiler classifies MPI functions as system functions similar to Intel® oneAPI Threading Building Blocks (oneTBB) and OpenMP* functions. This approach helps you focus on your code rather than MPI internals. You can use the VTune Profiler GUI [Call Stack Mode](#) filter bar combo box and CLI `call-stack-mode` option to enable displaying the system functions and thus view and analyze the internals of the MPI implementation. The call stack mode **User functions+1** is especially useful to find the MPI functions that consumed most of CPU Time (Hotspots analysis) or waited the most (Threading analysis). For example, in the call chain `main() -> foo() -> MPI_Bar() -> MPI_Bar_Impl() -> ..., MPI_Bar()` is the actual MPI API function you use and the deeper functions are MPI implementation details. The call stack modes behave as follows:

- The **Only user functions** call stack mode attributes the time spent in the MPI calls to the user function `foo()` so that you can see which of your functions you can change to actually improve the performance.
- The default **User functions+1** mode attributes the time spent in the MPI implementation to the top-level system function - `MPI_Bar()` so that you can easily see outstandingly heavy MPI calls.
- The **User/system functions** mode shows the call tree without any re-attribution so that you can see where exactly in the MPI library the time was spent.

NOTE

VTune Profiler prefixes the profile version of MPI functions with `P`, for example: `PMPI_Init`.

VTune Profiler provides oneTBB and OpenMP support. Use these thread-level parallel solutions in addition to MPI-style parallelism to maximize the CPU resource usage across the cluster, and to use the VTune Profiler to analyze the performance of that level of parallelism. The MPI, OpenMP, and oneTBB features in the VTune Profiler are functionally independent, so all usual features of OpenMP and oneTBB support are applicable when looking into a result collected for an MPI process. For hybrid OpenMP and MPI applications, the VTune Profiler displays a summary table listing top MPI ranks with OpenMP metrics sorted by [MPI Busy Wait](#) from low to high values. The lower the Communication time is, the longer a process was on a critical path of MPI application execution. For deeper analysis, explore [OpenMP analysis](#) by MPI processes laying on the critical path.

Example:

This example displays the performance report for functions and modules analyzed for any analysis type. Note that this example opens per-node result directories (`result_dir.host1`, `result_dir.host2`) and groups data by processes -mpi ranks encapsulated in the per-node result:

```
vtune -R hotspots -group-by process,function -r result_dir.host1
```

```
vtune -R hotspots -group-by process,module -r result_dir.host2
```

MPI Implementations Support

You can use the VTune Profiler to analyze both Intel MPI library implementation and other MPI implementations. But beware of the following specifics:

- Linux* only: Based on the `PMI_RANK` or `PMI_ID` environment variable (whichever is set), the VTune Profiler extends a process name with the captured rank number that is helpful to differentiate ranks in a VTune Profiler result with multiple ranks. The process naming schema in this case is `<process_name>(rank <N>)`. To enable detecting an MPI rank ID for MPI implementations that do not provide the environment variable, use the `-trace-mpi` option.
- For the Intel MPI library, the VTune Profiler classifies MPI functions/modules as system functions/modules (the **User functions+1** option) and attributes their time to system functions. This option may not work for all modules and functions of non-Intel MPI implementations. In this case, the VTune Profiler may display some internal MPI functions and modules by default.
- You may need to adjust the command line examples in this help section to work for non-Intel MPI implementations. For example, you need to adjust command lines provided for different process ranks to limit the number of processes in the job.
- An MPI implementation needs to operate in cases when there is the VTune Profiler process (`vtune`) between the launcher process (`mpirun/mpiexec`) and the application process. It means that the communication information should be passed using environment variables, as most MPI implementations do. VTune Profiler does not work on an MPI implementation that tries to pass communication information from its immediate parent process.

MPI System Modules Recognized by the VTune Profiler

VTune Profiler uses the following regular expressions in the Perl syntax to classify MPI implementation modules:

- `impi*.dll`
- `impid*.dll`
- `impidmt*.dll`
- `impil*.dll`
- `impilmt*.dll`
- `impimt*.dll`
- `libimalloc*.dll`
- `libmpi_ilp64*.dll`

NOTE

This list is provided for reference only. It may change from version to version without any additional notification.

Analysis Limitations

- Intel VTune Profiler does not support MPI dynamic processes like the `MPI_Comm_spawn` dynamic process API.

Additional Resources

For more details on analyzing MPI applications, see the Intel Parallel Studio Cluster Edition and online MPI documentation at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library-documentation.html>. For information on installing VTune Profiler in a cluster environment, see the [Intel VTune Profiler Installation Guide for Linux](#).

There are also other resources available online that discuss usage of the VTune Profiler with other Parallel Studio Cluster Edition tools:

- *Tutorial: Analyzing an OpenMP* and MPI Application* available from <https://www.intel.com/content/www/us/en/docs/vtune-profiler/tutorial-vtune-itac-mpi-openmp/2020/overview.html>
- *Hybrid applications: Intel MPI Library and OpenMP* at <https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-applications-mpi-openmp.html>

See Also

[Cookbook: Profiling MPI Applications](#)

[Specify Search Directories from Command Line
from command line](#)

[HPC Performance Characterization Analysis](#)

[HPC Performance Characterization View](#)

OpenSHMEM* Code Analysis with Fabric Profiler

On Linux systems, analyze the runtime behavior of OpenSHMEM or Intel® SHMEM applications with Fabric Profiler (preview feature).

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

Fabric Profiler is a performance analysis application that you use to profile OpenSHMEM or Intel® SHMEM code. The application collects and displays diagnostics, trace files, and runtime information for these types of code. Fabric Profiler runs on Linux* platforms only.

Get Started with Fabric Profiler

The Fabric Profiler application has two components:

- The **Data collector** monitors the behavior of the application and network when the OpenSHMEM application is running.
- The **Analyzer** is a collection of tools that executes after the OpenSHMEM application has finished running. These tools display profiling results with interactive features to help you explore communication-centric behaviors.

Fabric Profiler operates in two modes:

- Use the **OpenSHMEM** mode to profile OpenSHMEM (SoS)-based applications.
- Use the **Intel® SHMEM** mode to profile OpenSHMEM applications along with a subset of the Intel® SHMEM API. If you use any subset of the Intel® SHMEM API, you must use Fabric Profiler in this mode.

Access Fabric Profiler

The Fabric Profiler package is bundled with the installation package of Intel® VTune™ Profiler. Access Fabric Profiler in the `vtune\<vtune_profiler_version>\fabric_profiler` directory.

In addition to the Fabric Profiler application, this directory contains:

- Product documentation
- Examples
- Sample trace files

Install Fabric Profiler

To install Fabric Profiler, you set up the data collector and the analyzer components.

Set Up the Data Collector

The Fabric Profiler data collector is implemented as a library that intercepts the OpenSHMEM calls and/or Intel® SHMEM host calls of the application. The data collector also monitors network activity. The data collector populates binary trace files with this information.

Prerequisites:

- Unzip the data collector package.
- Set the `ESP_ROOT` environment variable to point to the location where you unzipped the data collector.
- Install these libraries:
 - Fabric Profiler uses **PAPI** to gather system metrics at runtime. To add PAPI to your environment, run `module load papi`. You can also download PAPI from <https://icl.utk.edu/papi/> and build it.
 - The **Libfabric** library helps to obtain access to the fabric of the OpenSHMEM code. This library should be present in the cluster with CXI support. To do this, run `module load libfabric`. For more information, see the [Libfabric Programmer's Manual](#).
 - To track portions of the Intel® SHMEM API, Fabric Profiler uses **Intel® Pin**. Download a version of Intel® Pin that is 3.28 or newer.

Set Up the Analyzer

The Fabric Profiler analyzer is a collection of MATLAB* programs that run in the MATLAB runtime environment. These programs read trace files and display results.

Prerequisite:

To set up the analyzer, you must have the **MATLAB Runtime Environment**. Download the environment from <https://www.mathworks.com/products/compiler/mcr.html>. Select a version that is R2021b(9.11) or newer.

You can find the Fabric Profiler analyzer executable (`fpro`) in `$ESP_ROOT/bin/analyzer/fpro_analyzer`.

Fabric Profiler Workflow

In the Fabric Profiler workflow, you perform these steps:

1. Build and run an application using the data collector.
2. Generate trace files.
3. View trace files using the analyzer.

Step 1: Build and Run an Application

Once you have installed Fabric Profiler on a Linux machine, complete these steps to build and run an application. This procedure describes how you build an OpenSHMEM as well as an Intel® SHMEM application.

1. Define Fabric Profiler regions in the source code. This way, you can see named regions in the analyzer displays for easier analysis.
 - a. Include the header file `esp.h`.
 - b. Mark regions of interest:

```
esp_enter("<region_A name>");  
esp_exit("<region_A name>");
```

Make sure to use the same name of the region in the enter and exit calls.

- c. Rebuild the application.

NOTE You cannot nest or interleave regions.

2. Build an application with Fabric Profiler instrumentation.

Make sure that you have set the required environment variables. To do this, edit `setMyVars.sh` as needed.

- **OpenSHMEM Applications:**

Fabric Profiler uses `LD_PRELOAD` at runtime to link in the data collector library before the SHMEM library. If you did not add Fabric Profiler regions to your source code, you do not need to rebuild your application.

For example, to build the `$ESP_ROOT/examples/SHMEM/sanity` application, run `make` on the makefile of the `sanity` application.

- **Intel® SHMEM Applications:**

For Intel® SHMEM applications, Fabric Profiler uses a combination of Intel® Pin and `LD_PRELOAD` at runtime to link in the data collector library before the SHMEM library. If you did not add Fabric Profiler regions to your source code, you do not need to rebuild your application.

3. Run the application.

Use the `$ESP_ROOT/bin/collector/fpro` script. This script adds the data collector library to the `LD_PRELOAD` variable. Since the data collector library uses the PAPI library. You may need to run `module load papi`, or add PAPI to your library paths.

- **OpenSHMEM Applications:**

For example, to run `fpro` on the `sanity` application,

1. Go to `$ESP_ROOT/bin/collector/` directory.
2. Run

```
./fpro -j pbs -r "R1234" -n 1 -p 2 -1 1 $ESP_ROOT/examples/SHMEM/sanity/sanity
```

The `-I 1` indicates that `sanity` is an OpenSHMEM application. Make sure to use your own PBS reservation number.

- **Intel® SHMEM Applications:**

For example, to run `fpro` on the Intel SHMEM `sycl_sanity` application,

1. Go to `$ESP_ROOT/bin/collector/` directory.
2. Run

```
./fpro -j pbs -r "R1234" -n 1 -p 2 -l 0 $ESP_ROOT/examples/iSHMEM/sycl_sanity/sycl_sanity
```

The `-I 0` indicates that `sycl_sanity` is an Intel® SHMEM application.

Step 2: Generate Trace Files

The data collector monitors network activity and the execution of your application. Once the execution completes, the data collector writes output to the trace files. This phase can add an additional 10% to your wall time.

To generate trace files,

1. Check the output of the application. Make sure that the code instrumentation by the data collector was successful. To do this,
 - a. Ensure that the `ESP_VERTOSITY_LEVEL` environment variable is greater than 0.
 - b. Call `shmemp_init` (OpenSHMEM applications) or `ishmem_init` (Intel® SHMEM applications). The start banner of Fabric Profiler displays.
 - c. Call `shmemp_finalize` (OpenSHMEM applications) or `ishmem_finalize` (Intel® SHMEM applications). The stop banner of Fabric Profiler displays.
2. The `Fpro` script merges the trace files. This script uses the following tools in sequence:
 - a. `mergeFuncFile`

- b.** mergeProfileFile
 - c.** mergePutFile
3. Copy the merged trace files from the root level of the traces directory to the machine where you have installed the analyzer.

You can now use the analyzer to view trace files.

Step 3 : View Trace Files using the Analyzer Suite

Fabric Profiler provides a set of five different analyzers to help you read trace files. In the Fabric Profiler application, you can find all of the analyzers in the \$ESP_ROOT/bin/analyzer directory. The analyzers are:

- ba - Barrier analyzer
- fbla - Fabric backlog analyzer
- la - Fabric latency analyzer
- msa - Message straggler analyzer
- r - An HTML report that contains a summary of all analyzer results.

Access the Analyzer Set

Access all of these analyzers through the `fpro_analyzer` executable in the \$ESP_ROOT/bin/analyzer/ directory. You can run the executable from the command prompt.

Command Line Options

To access the help menu, run:

```
$ ./fpro_analyzer --help
```

Command	Purpose
<code>fpro_analyzer --help</code>	Display usage
<code>fpro_analyzer --version -v -V</code>	Display information about version and build
<code>fpro_analyzer - start</code>	Start the fabric backlog analyzer
<code>fpro_analyzer {ba fbla la msa r}</code>	Start with ba, fbla, la, msa, or r analyzers
<code>fpro_analyzer <trace file></code>	Open fabric backlog analyzer with the specified trace file
<code>fpro_analyzer {ba fbla la msa r} <trace file></code>	Open the selected analyzer with the specified trace file
<code>fpro_analyzer {ba fbla la msa r} <fabric select> <trace file></code>	Open the selected analyzer and selected fabric with the specified trace file.
	For the fabric, select Cray-Slingshot11 (or 1) or Cray-Aries (or 2).

You can specify the trace file by providing the full path to the trace or the directory that contains traces. For example, these are both valid commands:

```
fpro_analyzer fbla /path/to/traces/
fpro_analyzer fbla /path/to/traces/my_trace.ucl.put
```

Here are some more examples that use the options described in this section:

- Run the `fpro_analyzer` executable and start the function backlog analyzer:

```
$ ./fpro_analyzer
```

- Run the `fpro_analyzer` executable and start the message straggler analyzer:

```
$ ./fpro_analyzer msa
```

- Run the `fpro_analyzer` executable and start the function latency analyzer to review traces with the Cray-Slingshot11 fabric:

```
$ ./fpro_analyzer la Cray-Slingshot11 /path/to/traces/
```

Contents of Trace Files

When your application calls `shmem_finalize` or `ishmem_finalize`, the data collector writes five trace files which contain information about application behavior.

Trace File	Format	Contents
{trace-file-prefix}.uc1.func	Binary	Information about every profiled SHMEM function call. Each process writes out a separate function trace file. Once the job completes, the individual function trace files are merged into a single file with the <code>\$ESP_ROOT/bin/collector/mergeFuncFile</code> script. The analyzers require this merged file.
{trace-file-prefix}.uc1.hfi	Binary	When the SHMEM application is running, Fabric Profiler monitors send and receive counters on the host fabric interface card. The HFI file contains these time-stamped counter values.
{trace-file-prefix}.uc1.profile	Binary	When the SHMEM application is running, Fabric Profiler monitors system performance counters and gathers system information. This data is written into the profile files. Each process writes out a separate profile file. When the job completes, the individual profile trace files are merged into a single file with the <code>\$ESP_ROOT/bin/collector/mergeProfileFile</code> binary. The analyzers require this merged file.
{trace-file-prefix}.uc1.put	Binary	Fabric Profiler monitors the amount of data injected into the network with each <code>shmem_put</code> call and the destination node for each put operation. The put file contains these values. When the job is complete, the individual put trace files are merged into a single file with the <code>\$ESP_ROOT/bin/collector/mergePutFile</code> binary.
{trace-file-prefix}.uc1.ev.txt	Text	The environment file is a list of all environment variables defined at SHMEM application run-time.

Types of Analyzers

Analyzer Type	Name	Purpose	Available Operations
ba	Barrier Trace Analyzer	Reads the function trace file and displays barrier wait times for each barrier call in the source code for each PE.	<ul style="list-style-type: none"> Take these measurements: <ul style="list-style-type: none"> PE wait time PE arrival time

Analyzer Type	Name	Purpose	Available Operations
fbla	Fabric Backlog Analyzer	Reads the put trace file and correlates that with the HFI trace file to visualize fabric backlog at any point in time.	<ul style="list-style-type: none"> • Node wait density • PE percent Late • PE Outlier Late • Vary the threshold. • Restrict your results to a specific lexical occurrence (a particular source code line containing a barrier) <ul style="list-style-type: none"> • Select Show Region Bounds and choose regions of interest. If the SHMEM code defined code regions, the temporal regions are highlighted on the graph of network backlog against time. • Select an individual node to display its associated backlog. • View injection and/or ejection backlog <ul style="list-style-type: none"> • Injection requested: Data that is sent to a different node by the application • Injection actual: Data that is actually sent into the network by the Host Fabric Interface (HFI) • Ejection requested: Data that is received by the current node • Ejection actual: Data actually received from the network according to HFI • Zoom and pan to bring areas into focus. • Try offset adjustment modes. • Switch between toggle and rate displays. • Use the data cursor. Click on the widget first. Next click anywhere on the plot to see data values for that point.
la	Fabric (latency) Trace Analyzer	Reads the function trace file and displays fabric latency for all instrumented SHMEM calls. Trace files that contain ~100,000s of function calls can take several minutes to complete. The default	<ul style="list-style-type: none"> • Select individual function calls to display latency hot spots for each call. • If the application defined Fabric Profiler regions, click View Regions. Choose regions to highlight temporal spans on the graph which represent those regions of code.

Analyzer Type	Name	Purpose	Available Operations
msa	Message Straggler Analyzer	Reads the function trace file and correlates the activity in the trace file with network activity in the HFI trace file.	
r	Analyzer Report	A non-interactive report that gathers information about a SHMEM application run and displays it in HTML format. The report can take several minutes to be completed. When completed, the HTML report is saved in the same location as the profile trace file, with a matching file name.	

GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics

Use the Intel® VTune™ Profiler to profile graphics applications and correlate activities on both the CPU and GPU.

Consider following these steps for GPU analysis with the VTune Profiler:

1. Set up your system for GPU analysis.
2. Run the [GPU Offload](#) analysis to identify whether your application is GPU bound and how effectively your code is offloaded to the GPU.
3. Run the [GPU Compute/Media Hotspots](#) analysis for detailed analysis of the GPU-bound application with explicit support of SYCL, Intel® Media SDK, and OpenCL™ software technology:
 - [Analyze GPU hardware metrics](#)
 - [Explore execution of OpenCL™ kernels](#)
 - [Explore execution of Intel Media SDK tasks](#)
 - Investigate execution of SYCL computing tasks
(supported with VTune Profiler 2021)

NOTE

You may also configure a custom analysis to collect GPU usage data. To do this, select the **GPU Utilization** option in the analysis configuration. This option introduces the least overhead during the collection, while the **Analyze Processor Graphics hardware events** adds medium overhead, and the **Trace GPU Programming APIs** option adds the biggest overhead.

Analyze GPU Usage for GPU-Bound Applications

If you already identified that your application or some of its stages are GPU bound, run the GPU Compute/Media Hotspots analysis in the **Characterization** mode to see whether GPU engines are used effectively and whether there is some room for improvement. Such an analysis is possible with hardware metrics collected by the VTune Profiler for the **Render and GPGPU** engine of the Intel Graphics.

Explore GPU Hardware Metrics

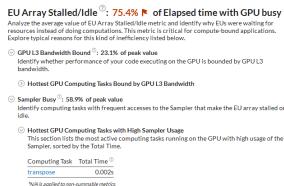
GPU hardware metrics can provide you with a next level of details to analyze GPU activity and identify whether any performance improvements are possible. You may configure the GPU Compute/Media Hotspots analysis to collect the following types of **GPU event metrics** on the Render and GPGPU engine of Intel Graphics:

- **Overview** (default) group analyzes general activity of GPU execution units, sampler, general memory, and cache accesses;
- **Compute Basic (with global/local memory accesses)** group analyzes accesses to different types of GPU memory;
- **Compute Extended** (for Intel® Core™ M processors and higher)
- **Full Compute** group combines metrics from the **Overview** and **Compute Basic** presets and presents them in the same view, which helps explore the reasons why the GPU execution units were waiting. To use this event set, make sure to enable the **multiple runs mode** in the target properties.

Start with the **Overview** events group and then move to the **Compute Basic (global/local memory accesses)** group. **Compute Basic** metrics are most effective when you analyze computing work on a GPU with the **GPU Utilization** events option enabled (default for the GPU Compute/Media Hotspots analysis), which allows you to correlate GPU hardware metrics with an exact GPU load.

When the data is collected, explore the **EU Array Stalled/Idle** section of the **Summary** window to identify the most typical reasons why the execution units could be waiting.

Depending on the event preset you used for the configuration, the VTune Profiler analyzes metrics for stalled/idle executions units. The GPU Compute/Media Hotspots analysis by default collects the **Overview preset** including the metrics that track general GPU memory accesses, such as Sampler Busy and Sampler Is Bottleneck, and GPU L3 bandwidth. As a result, the **EU Array Stalled/Idle** section displays the Sampler Busy section with a list of GPU computing tasks with frequent access to the Sampler and hottest GPU computing tasks bound by GPU L3 bandwidth:



If you select the **Compute Basic** preset during the analysis configuration, VTune Profiler analyzes metrics that distinguish accessing different types of data on a GPU and displays the **Occupancy** section. See information about GPU tasks with low occupancy and understand how you can achieve peak occupancy:

Welcome x r084gh x

GPU Compute/Media Hotspots (preview) GPU Compute/Media Hotspots (preview) ▾ ⓘ ? !

Analysis Configuration Collection Log Summary **Graphics**

Elapsed Time ?: 5.700s

If your application target was run more than once during the collection, this value includes elapsed time for all the runs.

GPU Time ?: 0.009s

EU Array Stalled/Idle ?: 84.3% ! !

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of being critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

- GPU L3 Bandwidth Bound ?: 0.0% !
- Occupancy ?: 53.2% ! !

Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the accesses and barriers may affect the maximum possible occupancy.

Hottest GPU Computing Tasks with Low Occupancy !

This section lists the most active computing tasks running on the GPU with the lowest occupancy.

Computing Task	Total Time ?	Global Size ?	Local Size ?	SIMD Width ?
workload	0.008s	4096	32	

*N/A is applied to non-summable metrics.

Occupancy, %

25.5%

0% 26%

red flag zone tuning potential

The normalized sum of all cycles on all slots when a slot has a thread scheduled value).

! Ineffective work scheduling can cause spikes in the occupancy metric.

Sampler Busy ?: 78.2% !

FPU Utilization ?: 1.9%

Bandwidth Utilization Histogram

If the **peak occupancy** is flagged as a problem for your application, inspect factors that limit the use of all the threads on the GPU. Consider modifying your code with corresponding solutions:

Factor responsible for Low Peak Occupancy	Solution
SLM size requested per workgroup in a computing task is too high	Decrease the SLM size or increase the Local size
Global size (the number of working items to be processed by a computing task) is too low	Increase Global size

Factor responsible for Low Peak Occupancy	Solution
Barrier synchronization (the sync primitive can cause low occupancy due to a limited number of hardware barriers on a GPU subslice)	Remove barrier synchronization or increase the Local size

EU Array Stalled/Idle: 79.9% ↗

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computations. This metric is critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

GPU L3 Bandwidth Bound: 9.5%

Occupancy: 78.8% ↗

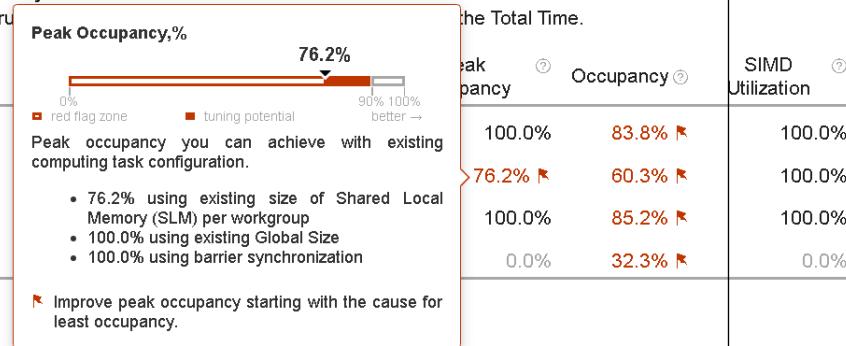
Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the scheduler. Note that frequent SLM accesses and barriers may affect the maximum possible occupancy.

Hottest GPU Computing Tasks with Low Occupancy

This section lists the most active computing tasks running on the GPU with a low Occupancy, sorted by the Total Time.

Computing Task	Total Time
kernel_ocl_path_trace_shader_evaluation	0.457s
kernel_ocl_path_trace_shader_sort	0.323s
kernel_ocl_path_trace_lamp_emission	0.208s
[Others]	0.033s

*N/A is applied to non-summable metrics.



If the **occupancy** is flagged as a problem for your application, change your code to improve hardware thread scheduling. These are some reasons that may be responsible for ineffective thread scheduling:

- A tiny computing task could cause considerable overhead when compared to the task execution time.
- There may be high imbalance between the threads executing a computing task.

EU Array Stalled/Idle: 79.9% ↗

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computations. This metric is critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

GPU L3 Bandwidth Bound: 9.5%

Occupancy: 78.8% ↗

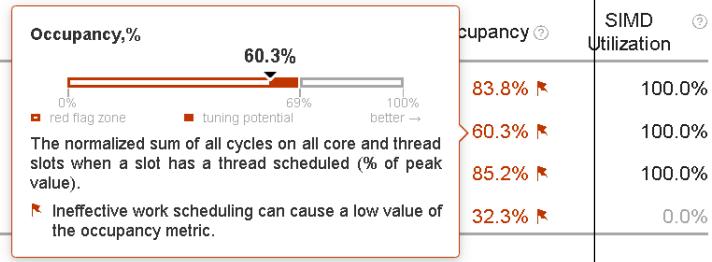
Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the scheduler. Note that frequent SLM accesses and barriers may affect the maximum possible occupancy.

Hottest GPU Computing Tasks with Low Occupancy

This section lists the most active computing tasks running on the GPU with a low Occupancy, sorted by the Total Time.

Computing Task	Total Time	Global Size
kernel_ocl_path_trace_shader_evaluation	0.457s	640 x 652
kernel_ocl_path_trace_shader_sort	0.323s	640 x 652
kernel_ocl_path_trace_lamp_emission	0.208s	640 x 652
[Others]	0.033s	

*N/A is applied to non-summable metrics.



The **Compute Basic** preset also enables an analysis of the DRAM bandwidth usage. If the GPU workload is DRAM bandwidth-bound, the corresponding metric value is flagged. You can explore the table with GPU computing tasks heavily using the DRAM bandwidth during execution.

If you select the **Full Compute** preset and **multiple run mode** during the analysis configuration, the VTune Profiler will use both **Overview** and **Compute Basic** event groups for data collection and provide all types of reasons for the EU array stalled/idle issues in the same view.

NOTE

To analyze Intel® HD Graphics and Intel® Iris® Graphics hardware events, make sure to [set up your system for GPU analysis](#)

To analyze GPU performance data per HW metrics over time, open the **Graphics** window, and focus on the **Timeline** pane. List of GPU metrics displayed in the **Graphics** window depends on the hardware events preset selected during the analysis configuration.

The example below shows the **Overview** group of metrics collected for the GPU bound application:



The first metric to look at is **GPU Execution Units: EU Array Idle** metric. Idle cycles are wasted cycles. No threads are scheduled and the EUs' precious computational resources are not being utilized. If **EU Array Idle** is zero, the GPU is reasonably loaded and all EUs have threads scheduled on them.

In most cases the optimization strategy is to minimize the **EU Array Stalled** metric and maximize the **EU Array Active**. The exception is memory bandwidth-bound algorithms and workloads where optimization should strive to achieve a memory bandwidth close to the peak for the specific platform (rather than maximize **EU Array Active**).

Memory accesses are the most frequent reason for stalls. The importance of memory layout and carefully designed memory accesses cannot be overestimated. If the **EU Array Stalled** metric value is non-zero and correlates with the **GPU L3 Misses**, and if the algorithm is not memory bandwidth-bound, you should try to optimize memory accesses and layout.

Sampler accesses are expensive and can easily cause stalls. Sampler accesses are measured by the **Sampler Is Bottleneck** and **Sampler Busy** metrics.

Explore Execution of OpenCL™ Kernels

If you know that your application uses OpenCL software technology and the **GPU Computing Threads Dispatch** metric in the **Timeline** pane of the **Graphics** window confirms that your application is doing substantial computation work on the GPU, you may continue your analysis and capture the timing (and other information) of OpenCL kernels running on Intel Graphics. To run this analysis, enable the **Trace GPU Programming APIs** option during analysis configuration. The GPU Compute/Media Hotspots analysis enables this option by default.

The **Summary** view shows OpenCL kernels running on the GPU in the **Hottest GPU Computing Tasks** section and flags the performance-critical kernels. Clicking such a kernel name opens the **Graphics** window grouped by **Computing Task (GPU) / Instance**. You may also want to group the data in the grid by the Computing Task. VTune Profiler identifies the following *computing task purposes*: **Compute** (kernels), **Transfer** (OpenCL routines responsible for transferring data from the host to a GPU), and **Synchronization** (for example, `clEnqueueBarrierWithWaitList`).

The corresponding columns show the overall time a kernel ran on the GPU and the average time for a single invocation (corresponding to one call of `clEnqueueNDRangeKernel`), working group sizes, as well as averaged GPU hardware metrics collected for a kernel. Hover over a metric column header to read the metric description. If a metric value for a computing task exceeds a threshold set up by Intel architects for the metric, this value is highlighted in pink, which signals a performance issue. Hover over such a value to read the issue description.

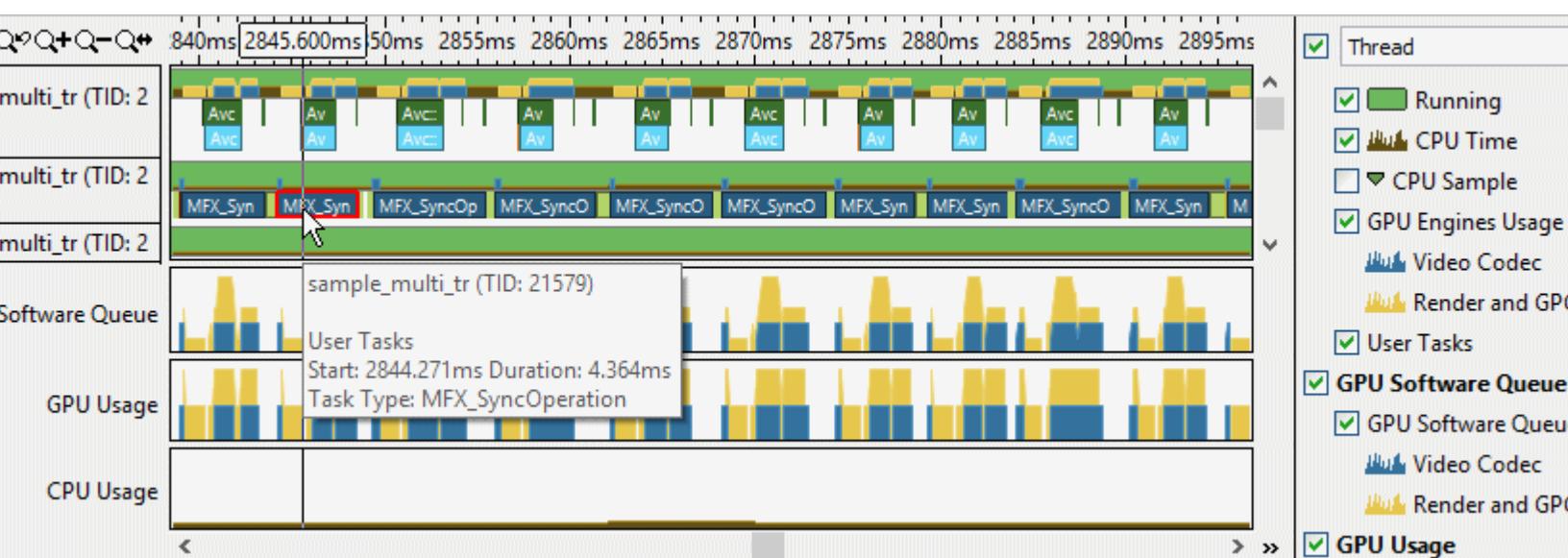
Analysis Configuration Collection Log Summary Graphics Platform Bottom-up											
Grouping: Computing Task / Instance											
Computing Task / Instance	Work Size		Computing Task					EU Array		L3 Shader Bandwidth,	
	Global	Local	Total Time ▼	Average...	Instanc...	SIMD ...	SV...	Active	Stalled	Idle	
▶ Accelerator_Intersect_R	1048576	64	53.398s	0.080s	669	16		85.6%	14.3%	0.1%	90.368
▶ AdvancePaths_MK_DL_	1048576	64	10.280s	0.015s	669	16		69.4%	30.5%	0.1%	71.860
▶ AdvancePaths_MK_SPL	1048576	64	10.243s	0.015s	669	16		30.1%	69.7%	0.2%	63.976
▶ AdvancePaths_MK_GEN	1048576	64	9.236s	0.014s	669	16		35.2%	64.7%	0.1%	65.490
▶ AdvancePaths_MK_GEN	1048576	64	8.226s	0.012s	669	16		24.8%	75.0%	0.2%	33.103
▶ AdvancePaths_MK_HIT_	1048576	64	7.804s	0.012s	669	16		32.5%	67.3%	0.1%	47.085
▶ AdvancePaths_MK_RT_I	1048576	64	7.160s	0.011s	669	16		43.8%	55.2%	1.0%	113.391
▶ AdvancePaths_MK_DL_	1048576	64	5.164s	0.008s	669	16		39.3%	60.5%	0.2%	58.673
▶ AdvancePaths_MK_NEX	1048576	64	4.812s	0.007s	669	32		17.2%	82.5%	0.3%	31.891
▶ AdvancePaths_MK_HIT_	1048576	64	2.991s	0.004s	669	32		11.8%	87.9%	0.3%	28.215
▶ AdvancePaths_MK_RT_	1048576	64	2.614s	0.004s	669	32		16.9%	82.8%	0.3%	28.784

Analyze and optimize hot kernels with the longest Total Time values first. These include kernels characterized by long average time values and kernels whose average time values are not long, but they are invoked more frequently than the others. Both groups deserve attention.

To [view details on OpenCL kernels submission and analyze the time spent in the queue](#), explore the **Computing Queue** data in the **Timeline** pane of the **Graphics** or **Platform** window.

Explore Execution of Intel Media SDK Tasks

If you enabled both the **GPU Utilization** and **Trace GPU Programming APIs** options for the [Intel Media SDK program analysis](#), use the **Graphics** window to correlate data for the Intel Media SDK tasks execution with the GPU software queue data.



Analyze GPU Kernels Per Code Line

You can run the GPU Compute/Media Hotspots Analysis in the **Code-Level Analysis** mode to narrow down your GPU analysis to a specific hot GPU kernel identified with the GPU Offload analysis. This analysis helps identify performance-critical basic blocks or issues caused by memory accesses in the GPU kernels providing performance statistics per code line/assembly instruction:

Source	Assembly	Estimated GPU Cycles	Address	Source	Assembly	Estimated GPU Cycles
257 // Read the node Inform	seende (0:160) r11:rd r4:r4 r11:0	0.4%	0x0f0 262	seende (0:160) r11:rd r4:r4 r11:0	0.4%	
258 const ushort lsz = 5; (node	0x0f0 263	0.4%	0x1000 265	mcv (0:160) r15:0,r13:rd r12:5,	0.2%	
259 const ushort lsz_t = (node	0x1000 265		0x1000 266	add (0:160) r126,0,r13:rd r124,	0.2%	
260 const float4 bboxes_min = 0.5;	0x1000 266		0x1000 267	r126,r125,r124,r123	0.2%	
261 const float4 bboxes_max = 0.5;	0x1000 267		0x1020 264	seende (0:160) r15:rd r4:r4 r11:2	0.4%	
262 const float4 bboxes_min = 0.5;	0x1020 264		0x1020 265	seende (0:160) r15:rd r4:r4 r17:0	0.4%	
263 const float4 bboxes_max = 0.5;	0x1020 265		0x1040 266	seende (0:160) r12:rd r4:r4 r10:4	0.4%	
264 const float4 bboxes_min = 0.5;	0x1040 266		0x1040 267	add (0:160) r12:rd r4:r4 r10:4	0.2%	
265 const float4 bboxes_max = 0.5;	0x1040 267		0x1050 268	r12:0,r11:rd r120,0c	0.2%	
266 const int4 children = 4;	0x1050 268		0x1050 269	add (0:160) r11:0,r13:rd r121,0	0.2%	
267	0x1050 269		0x1060 268	r11:0,r13:rd r122,0c	0.2%	
268 const int4 valid = 0;	0x1060 268	13.1%	0x1070 268	add (0:160) r119,0,r11:rd r121,	0.2%	
269 const maxs, min, bboxes	0x1070 268		0x1070 269	r119,0,r11:rd r121,0	0.2%	
270 bboxes_min, bboxes	0x1070 269		0x1080 268	add (0:160) r117,r13:rd r120,0c (0.2%)		
271 bboxes_max, bboxes	0x1080 268		0x1080 269	r117,r13:rd r120,0c (0.2%)		

See Also

[Intel® Media SDK Program Analysis](#)

(Linux* only)

[Configure GPU Analysis from Command Line](#)

[Error Message: Cannot Collect GPU Hardware Metrics](#)

[Rebuild and Install the Kernel for GPU Analysis](#)

GPU OpenCL™ Application Analysis

If you identified with the Intel® VTune™ Profiler that your application is **GPU-bound** and your application uses OpenCL™ software technology, you may enable the **Trace GPU Programming APIs** configuration option for your custom analysis to identify how effectively your application uses OpenCL kernels. By default, this option is enabled for the GPU Compute/Media Hotspots and GPU Offload analyses. To explore the performance of your OpenCL application, use the **GPU Compute/Media Hotspots** viewpoint.

Follow these steps to explore the data provided by the VTune Profiler for OpenCL application analysis:

1. Explore summary statistic:

- Analyze GPU usage.
- Identify why execution units (EUs) were stalled or idle.
- Identify OpenCL kernels overutilizing both Floating Point Units (FPUs).

2. Analyze hot GPU OpenCL kernels.
3. Correlate OpenCL kernels data with GPU metrics.
4. Explore the computing queue.
5. Analyze source and assembly code.

Explore Summary Statistics

Start your data analysis with the Summary window that provides application-level performance statistics. Typically, you focus on the primary baseline, which is the **Elapsed Time** metric that shows the total time your target ran:

The screenshot shows the VTune Profiler interface with the 'Summary' tab selected. A large box highlights the 'Elapsed Time' metric, which is displayed as 129.448s. Below this, there is descriptive text and a table showing GPU usage by engine type.

You can correlate this data with the **GPU Time** used by GPU engines while your application was running:

The screenshot shows the 'GPU Usage' section of the VTune Profiler interface. It displays a breakdown of GPU time by engine type. The table shows the following data:

GPU Engine / Packet Type	GPU Time (s)	(%)
Render and GPGPU	123.799s	95.6%
OpenCL	122.150s	94.4%
Unknown	1.649s	1.3%
Blitter	0.291s	0.2%
Unknown	0.291s	0.2%

If the GPU Time takes a significant portion of the Elapsed Time (95.6%), it clearly indicates that the application is GPU-bound. You see that 94.4% of the GPU Time was spent on the OpenCL kernel execution.

For OpenCL applications, the VTune Profiler provides a list of OpenCL kernels with the highest execution time on the GPU:

Hottest GPU Computing Tasks			
This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.			
Computing Task	Total Time	Average Time	Instance Count
Accelerator_Intersect_RayBuffer	53.398s	0.080s	669
AdvancePaths_MK_DL_ILLUMINATE *	10.280s	0.015s	669
AdvancePaths_MK_SPLAT_SAMPLE *	10.243s	0.015s	669
AdvancePaths_MK_GENERATE_NEXT_VERT	9.236s	0.014s	669
X_RAY *			
AdvancePaths_MK_GENERATE_CAMERA_RA	8.226s	0.012s	669
Y *			
[Others]	30.680s	N/A*	4,141

Mouse over the flagged kernels to learn what kind of performance problems were identified during their execution. Clicking such a kernel name in the list opens the **Graphics** window grouped by computing tasks, sorted by the Total Time, and with this kernel selected in the grid.

Depending on the **GPU hardware events preset** you used during the analysis configuration, the VTune Profiler explores potential reasons for stalled/idle GPU execution units and provides them in the Summary. For example, for the **Compute Basic** preset, you may analyze GPU L3 Bandwidth Bound issues:

EU Array Stalled/Idle: 43.2% of Elapsed time with GPU busy

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computations. This metric is critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

- GPU L3 Bandwidth Bound: 96.9% of peak value
Identify whether performance of your code executing on the GPU is bounded by GPU L3 bandwidth.
- Hottest GPU Computing Tasks Bound by GPU L3 Bandwidth
This section lists the most active computing tasks running on the GPU with high GPU L3 bandwidth, sorted by the Total Time.

Computing Task	Total Time
AdvancePaths_MK_DL_ILLUMINATE	10.280s
AdvancePaths_MK_SPLAT_SAMPLE	10.243s
AdvancePaths_MK_GENERATE_NEXT_VERTEX_RAY	9.236s
[Others]	20.128s

Or potential occupancy issues:

Occupancy: 97.0% of peak value

Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the scheduler. Note that frequent SLM accesses and barriers may affect the maximum possible occupancy.

- Hottest GPU Computing Tasks with Low Occupancy
This section lists the most active computing tasks running on the GPU with a low Occupancy, sorted by the Total Time.

Computing Task	Total Time	Global Size	Local Size	SIMD Width
Film_Clear	0.001s	640000	64	32

In this example, EU stalls are caused by GPU L3 high bandwidth. You may click the hottest kernels in the list to switch to the **Graphics** view, drill down to the **Source** or **Assembly** views of the selected kernel to identify possible options for cache reuse.

If your application execution takes more than 80% of collection time heavily utilizing floating point units, the VTune Profiler highlights such a value as an issue and lists the kernels that overutilized the FPUs:

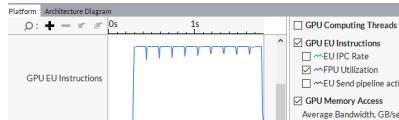
FPU Utilization: 82.2% of Elapsed time with GPU busy

Identify computing tasks with high utilization of the floating point execution units.

- Hottest GPU Computing Tasks with High FPU Utilization
This section lists the most active computing tasks that ran on the GPU heavily using the floating point execution units. Tasks in the table are sorted by the Total Time.

Computing Task	Total Time
workload	1.488s

You can switch to the Timeline pane on the **Graphics** tab and explore the distribution of the **GPU EU Instructions** metric that shows the FPU usage during the analysis run:



Analyze Hot GPU OpenCL Kernels

To view detailed information about all OpenCL kernels running on the GPU, switch to the **Graphics** window. By default, the grid data is grouped by **Computing Task / Instance** that shows Compute tasks only. Data collected for program units outside any OpenCL computing tasks are attributed to the [Outside any task] entry.

In the **Computing Task** columns explore the overall time a kernel ran on the GPU and the average time for a single invocation (corresponding to one call of `clEnqueueNDRangeKernel`), working group sizes, as well as averaged GPU hardware metrics collected for a kernel. Hover over a metric column header to read the metric description. If a metric value for a computing task exceeds a threshold set up by Intel architects for the metric, this value is highlighted in pink, which signals a performance issue. Hover over such a value to read the issue description.

In the example below, the Accelerator_Intersect kernel took the most time to execute (53.398s). The GPU metrics collected for this workload show high L3 Bandwidth usage spent in stalls when executing this kernel. For compute bound code it indicates that the performance might be limited by cache usage.

Analysis Configuration		Collection Log		Summary		Graphics		Platform		Bottom-up					
Grouping: Computing Task / Instance															
Computing Task / Instance	Work Size		Computing Task					EU Array			L3 Shader Bandwidth, [ns]				
	Global	Local	Total Time ▼	Average...	Instanc...	SIMD ...	SV...	Active	Stalled	Idle					
▶ Accelerator_Intersect_R	1048576	64	53.398s	0.080s	669	16		85.6%	14.3%	0.1%	90.368				
▶ AdvancePaths_MK_DL_I	1048576	64	10.280s	0.015s	669	16		69.4%	30.5%	0.1%	71.860				
▶ AdvancePaths_MK_SPL_I	1048576	64	10.243s	0.015s	669	16		30.1%	69.7%	0.2%	63.976				
▶ AdvancePaths_MK_GEN_I	1048576	64	9.236s	0.014s	669	16		35.2%	64.7%	0.1%	65.490				
▶ AdvancePaths_MK_GEN_I	1048576	64	8.226s	0.012s	669	16		24.8%	75.0%	0.2%	33.103				
▶ AdvancePaths_MK_HIT_I	1048576	64	7.804s	0.012s	669	16		32.5%	67.3%	0.1%	47.085				
▶ AdvancePaths_MK_RT_I	1048576	64	7.160s	0.011s	669	16		43.8%	55.2%	1.0%	113.391				
▶ AdvancePaths_MK_DL_I	1048576	64	5.164s	0.008s	669	16		39.3%	60.5%	0.2%	58.673				
▶ AdvancePaths_MK_NEX_I	1048576	64	4.812s	0.007s	669	32		17.2%	82.5%	0.3%	31.891				
▶ AdvancePaths_MK_HIT_I	1048576	64	2.991s	0.004s	669	32		11.8%	87.9%	0.3%	28.215				
▶ AdvancePaths_MK_RT_I	1048576	64	2.614s	0.004s	669	32		16.9%	82.8%	0.3%	28.784				

Analyze and optimize hot kernels with the longest Total Time values first. These include kernels characterized by long average time values and kernels whose average time values are not long, but they are invoked more frequently than the others. Both groups deserve attention.

If a kernel instance used the [OpenCL 2.0 Shared Virtual Memory \(SVM\)](#), the VTune Profiler detects it and, depending on your hardware, displays the SVM usage type as follows:

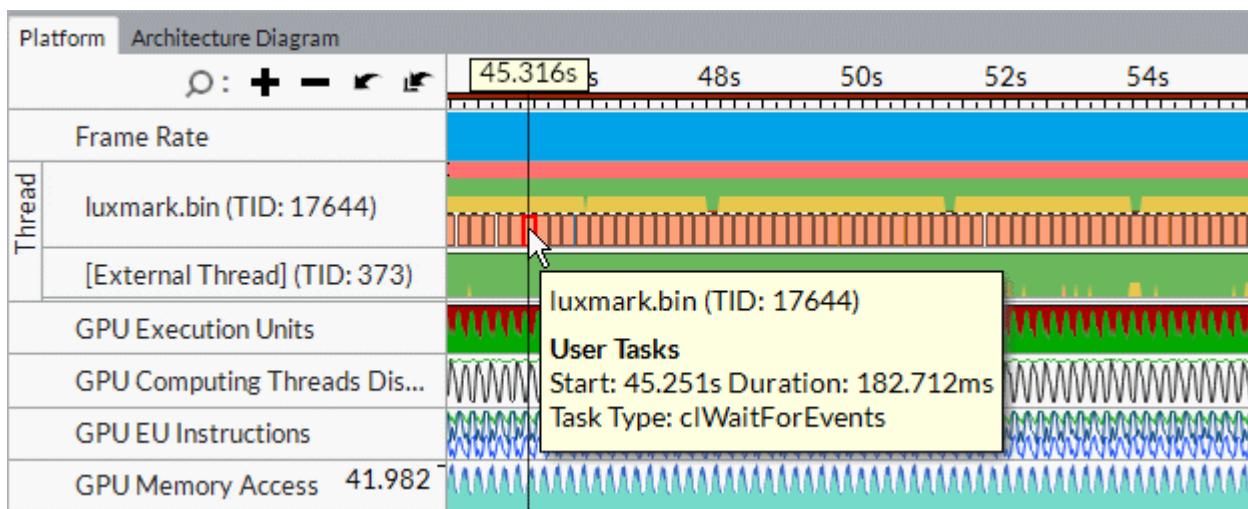
- **Coarse-Grained Buffer SVM:** Sharing occurs at the granularity of regions of OpenCL buffer memory objects. Cross-device atomics are not supported.
- **Fine-Grained Buffer SVM:** Sharing occurs at the granularity of individual loads and stores within OpenCL buffer memory objects. Cross-device atomics are optional.
- **Fine-Grained System SVM:** Sharing occurs at the granularity of individual loads/stores occurring anywhere within the host memory. Cross-device atomics are optional.

Every `clCreateKernel` results in a line in the **Compute** category. If two different kernels with the same name (even from the same source) were created with two `clCreateKernel` calls (and then invoked through two or more `clEnqueueNDRangeKernel`), two lines with the same kernel name appear in the table. If they are enqueued twice with a different global or local size or different sets of SVM arguments, they are also listed separately in the grid.

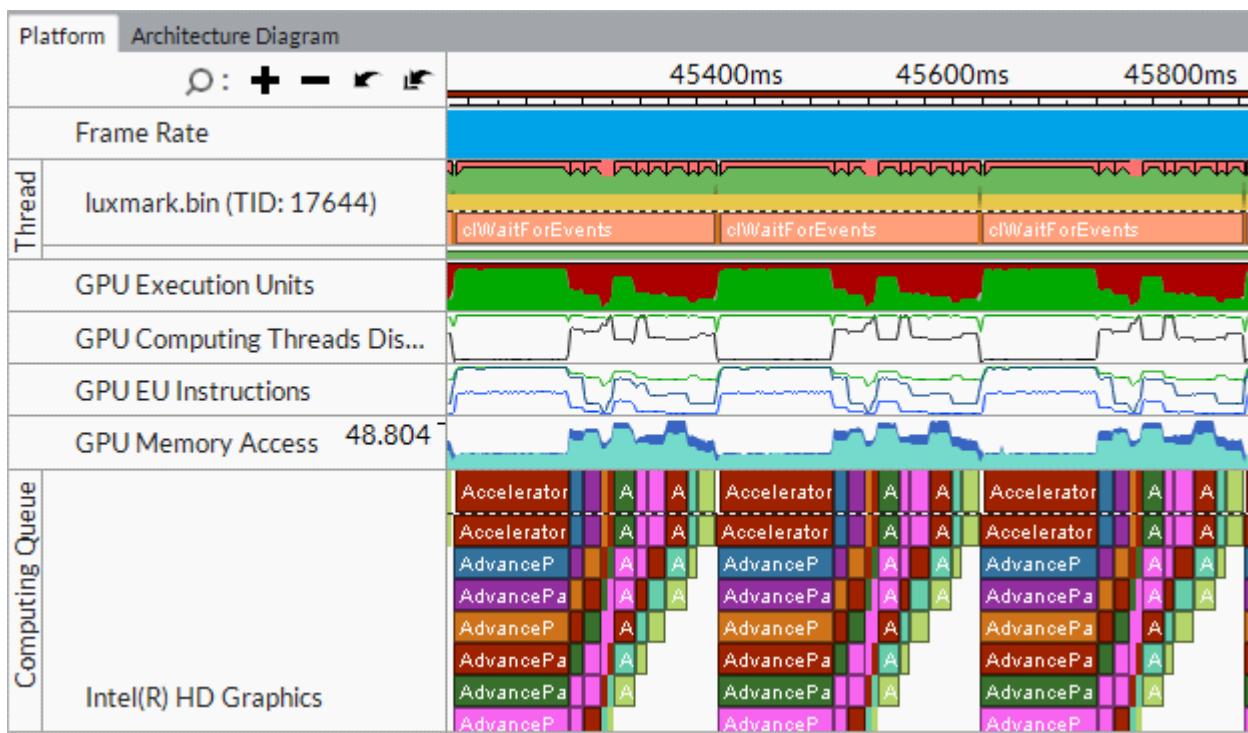
Correlate OpenCL Kernels Data with GPU Metrics

In the **Graphics** window, explore the **Timeline** pane > **Platform** tab to analyze OpenCL kernels execution over time.

OpenCL APIs (for example, `clWaitForEvents`) show up on the **Thread** area as tasks:



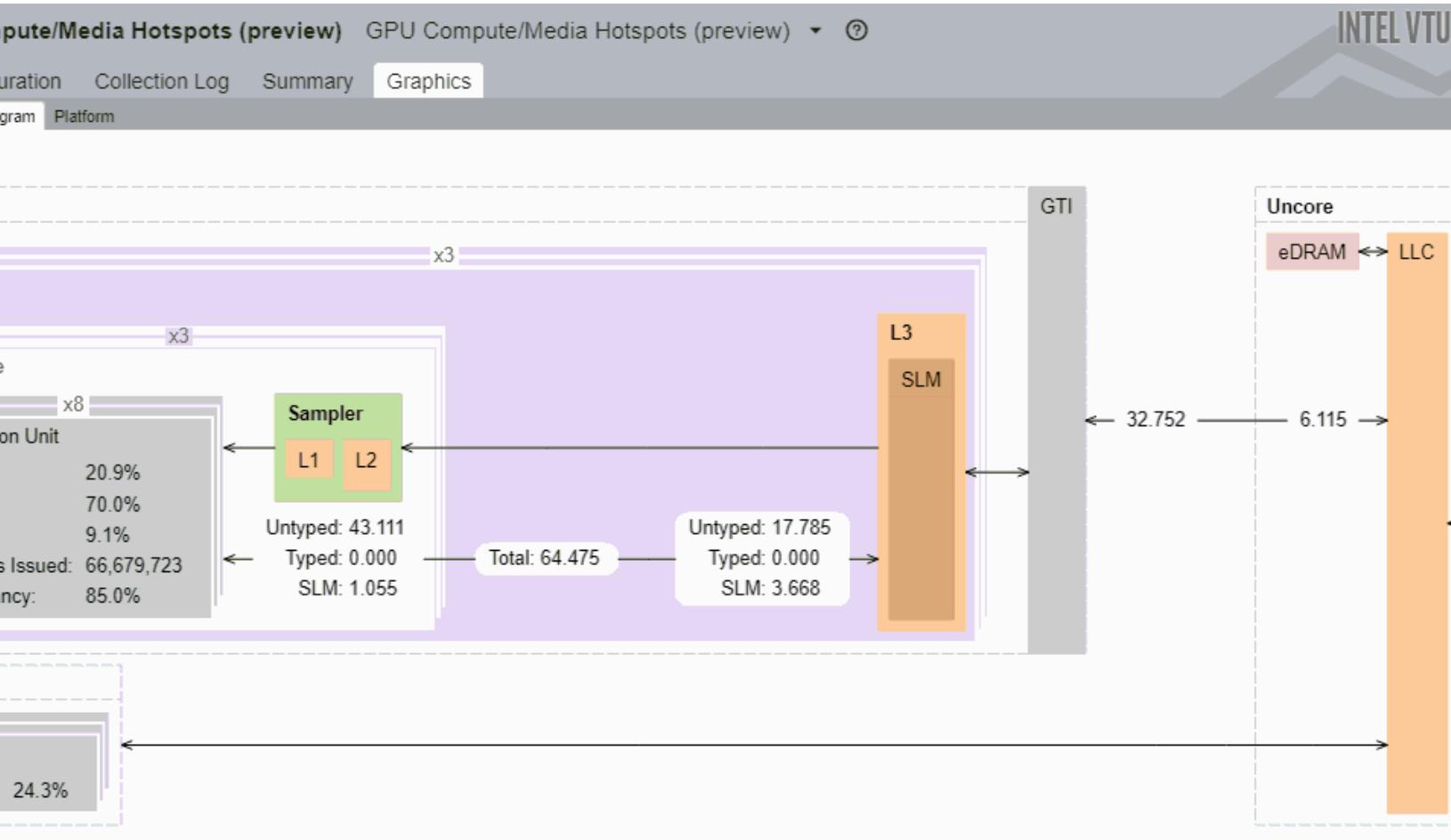
Correlate GPU metrics and OpenCL kernels data:



NOTE

GPU hardware metrics are available if you enabled the **Analyze Processor Graphics events** option for Intel® HD Graphics or Intel® Iris® Graphics. To collect these metrics, make sure to [set up your system for GPU analysis](#).

You may find it easier to analyze your OpenCL application by exploring the GPU hardware metrics per GPU architecture blocks. To do this, choose the **Computing Task** grouping level in the **Graphics** window, select an OpenCL kernel of interest and click the **Memory Hierarchy Diagram** tab in the **Timeline** pane. VTune Profiler updates the architecture diagram for your platform with performance data per GPU hardware metrics for the time range the selected kernel was executed.



Computing Task

Task	EU Array			L3 Bandwidth, GB/sec	Work Size		Computing Task			
	Active	Stalled	Idle		Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width
	13.0%	86.9%	0.2%	30.961	65536	64	0.004s	0.004s	1	32
	36.2%	56.0%	7.8%	69.586	65536	64	13.361s	0.001s	18,103	32
	50.8%	46.6%	2.6%	46.166	65536	64	75.644s	0.004s	18,103	8
	20.9%	70.0%	9.1%	64.475	65536	64	29.858s	0.002s	18,103	16
-	0.0%	0.0%	100.0%	0.000	362432	64	0.000s	0.000s	1	32
dBuffer	13.8%	73.7%	12.6%	107.978			0.274s	0.000s	2,266	
sk]	33.4%	46.1%	20.5%	11.521			0s			

Currently this feature is available starting with the 4th generation Intel® Core™ processors and the Intel® Core™ M processor, with a wider scope of metrics presented for the latter one.

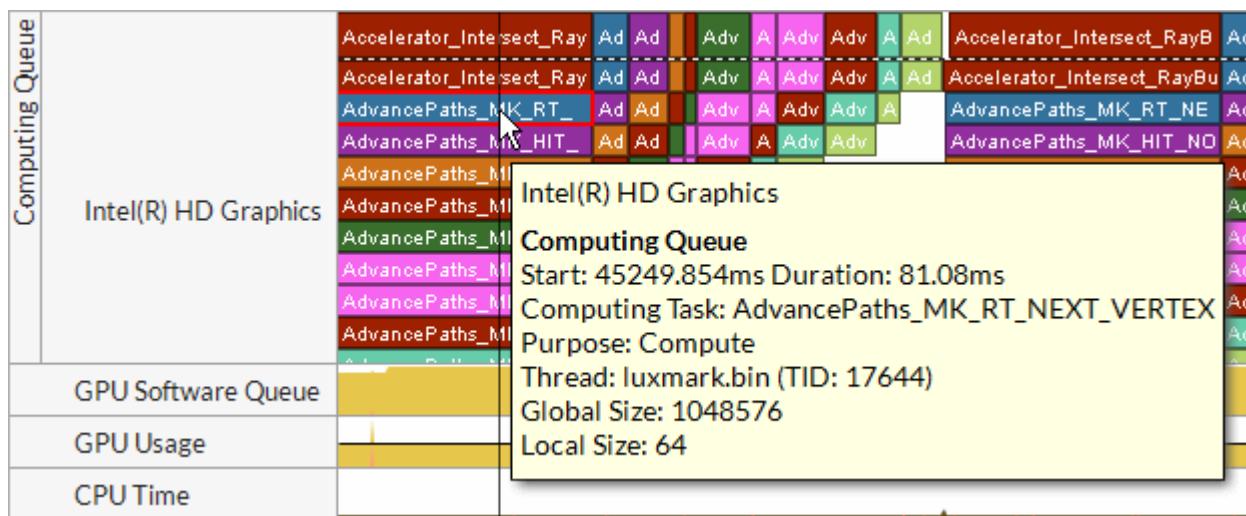
NOTE

You can right-click the **Memory Hierarchy Diagram**, select **Show Data As** and choose a format of metric data representation:

- Total Size
- Bandwidth (default)
- Percent of Bandwidth Maximum Value

Explore the Computing Queue

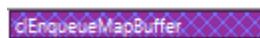
To view details on OpenCL kernels submission, in particular distinguish the order of submission and execution, and analyze the time spent in the queue, [zoom in](#) and explore the **Computing Queue** data in the **Timeline** pane. You can click a kernel task to highlight the whole queue to the execution displayed at the top layer. Kernels with the same name and size show up in the same color.



VTune Profiler displays kernels with the same name and size in the same color. Synchronization tasks are marked with vertical hatching



. Data transfers, OpenCL routines responsible for transferring data from the host system to a GPU, are marked with cross-diagonal hatching

**NOTE**

In the Attach mode if you attached to a process when the computing queue is already created, VTune Profiler will not display data for the OpenCL kernels in this queue.

Analyze Source and Assembly Code

You may select a computing task of interest in the grid view, double-click it to open the **Source/Assembly** window and analyze the code for the selected kernel (with source files available).

Source	Assembly	Time	Addr.	Blocks	Assembly
S. ▲	Source				
1 #define LOCAL_SIZE 1			0	3	mov (line) null,r24; mov 0x24A7000d 0x800000000000004c
2			0x10	3	mov (line) r10,r24; mov 0x1000000000000000d 0x8000000000000000
3			0x20	3	[W] or (1 M0) rnh,0x1000000000000000d; mov 0x1000000000000000d,0x8000000000000000
4 l			0x30	10	(W) nov (1 M0) 22,0<clifug z3,0>clif
5 #if LOCAL_SIZE > 0			0x38	10	(W) nov (1 M0) r4,0<clifif 0x0if
6 const int size = LOCAL_SIZE + 1024 / aise;			0x48	12	(W) cmp (1 M0) (ptr)f1,0 null,loc3;
7 _local int buffer[size];			0x58	12	(W) add (1 M0) r10,r10; mov 0x1000000000000000d,0x8000000000000000
8 #endif			0x68	10	(W) sende (1 M0) nullif r2 24 0x4c
9			0x78	12	_loc(f1,0) long; bb_11
10 *result = 0;0;			0x88		Block 2:

Analyze the assembler code provided by your compiler for the OpenCL kernel, estimate its complexity, identify issues, match the critical assembly lines with the affected source code, and optimize, if possible. For example, if you see that some code lines were compiled into a high number of assembly instructions, consider simplifying the source code to decrease the number of assembly lines and make the code more cache-friendly.

Explore GPU metrics data per computing task in the **Graphics** window and drill down to the **Source/Assembly** view to explore instructions that may have contributed to the detected issues. For example, if you identified the Sampler Busy or Stalls issues in the **Graphics** window, you may search for the `send` instructions in the **Assembly** pane and analyze their usage since these instructions often cause frequent stalls and overload the sampler. Each `send/sends` instruction is annotated with comments in square brackets that show a purpose of the instruction, such as data reads/writes (for example, `Typed/Untyped Surface Read`), accesses to various architecture units (`Sampler`, `Video Motion Estimation`), end of a thread (`Thread Spawner`), and so on. For example, this `sends` instruction is used to access the Sampler unit:

```
0x408 260 sends (8|M0) r10:d r100 r8 0x82 0x24A7000 [Sampler, msg-length:1, resp-length:4, header:yes, func-control:27000]
```

NOTE

- Source/Assembly support is available for OpenCL programs with sources and for kernels created with IL (intermediate language), if the intermediate SPIR-V binary was built with the `-gline-tables-only -s <cl_source_file_name>` option.
- The Source/Assembly analysis is not supported for the source code using the `#line` directive.
- If your OpenCL kernels use inline functions, you can enable the [Inline Mode filter bar option](#) to view inline functions in the grid and analyze them in the Source view.

See Also

[GPU Compute/Media Hotspots Analysis \(Preview\)](#)

[OpenCL™ Kernel Analysis Metrics Reference](#)

[GPU Metrics Reference](#)

Intel® Media SDK Program Analysis

Use Intel® VTune™ Profiler to enable analysis of Intel® Media SDK tasks execution over time.

Prerequisites:

To analyze the Intel Media SDK tasks execution, make sure to do the following:

- **Windows* OS:** Install the latest Intel Graphics driver from <https://www.intel.com/content/www/us/en/download-center/home.html>
- **Linux* OS:** Install the Intel® Media SDK and check that your system is [configured for GPU analysis](#). For remote collection, [configure your target Linux system](#).
- If you are running the analysis on a Windows machine, register your **GPU Event Trace for Windows (ETW)** so that you can see packet details of the execution of the MediaSDK program. At the command line, type:

```
<vtune>\bin64\amplxe-gpuetwreg.exe -s
```

To configure the Intel Media SDK program analysis, do the following:

1. Configure your target for analysis.
 - For the **Attach to Process** and **Profile System** target types, enable MFX tracing.
2. Enable tracing Intel Media SDK programs and run the analysis.

Configure Target

Launch the VTune Profiler with root privileges and configure analysis for your Intel Media SDK target.

For the **Launch Application** mode, follow the standard [project setup](#) and [analysis target setup](#) process and specify your application or a script as a target. VTune Profiler automatically sets environment variables and, on Linux, creates an `.mfx_trace` configuration file for Intel Media SDK program analysis.

For the **Attach To Process** and **Profile System** modes, the `.mfx_trace` is not created by the VTune Profiler automatically, which makes the Intel Media SDK program analysis incomplete. You need to manually enable MFX tracing as follows:

1. Configure the system to include ITT traces to the result.

For Linux:

```
export INTEL_LIBITNOTIFY32=/opt/intel/oneapi/vtune/latest/lib32/runtime/
libittnotify_collector.so
export INTEL_LIBITNOTIFY64=/opt/intel/oneapi/vtune/latest/lib64/runtime/
libittnotify_collector.so
```

For Windows:

```
set INTEL_LIBITNOTIFY32=C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin32\runtime
\ittnotify_collector.dll
set INTEL_LIBITNOTIFY64=C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\runtime
\ittnotify_collector.dll
```

2. On Linux, before running the analysis, generate the `.mfx_trace` file:

```
echo "Output=0x30" > $HOME/.mfx_trace
chmod +r $HOME/.mfx_trace
```

If, for some reason, settings in this file are different from the settings specified in the VTune Profiler project, the `.mfx_trace` settings will prevail and re-write the VTune Profiler project settings.

Run Analysis

1. Click the



Configure Analysis button on the VTune Profiler toolbar.

2. In the **HOW** pane, select an analysis type for Intel Media SDK program profiling, for example: [GPU Compute/Media Hotspots analysis](#), [GPU Offload analysis](#), or a custom analysis.
3. Make sure the **Trace GPU Programming APIs** option is selected.
4. Optionally: For custom analysis, select the **GPU Utilization** option.
For the GPU Compute/Media Hotspots and GPU Offload analysis types, this option is enabled by default.
5. Click **Start** to launch the analysis.

When the data collection completes, the VTune Profiler opens the result in the default viewpoint. Start with the **Graphics** window to analyze the CPU workload during the execution of the Intel Media SDK tasks.

See Also

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

Configure GPU Analysis from Command Line

`knob enable-gpu-runtimes`
to enable Intel Media SDK program analysis from command line

Frame Data Analysis

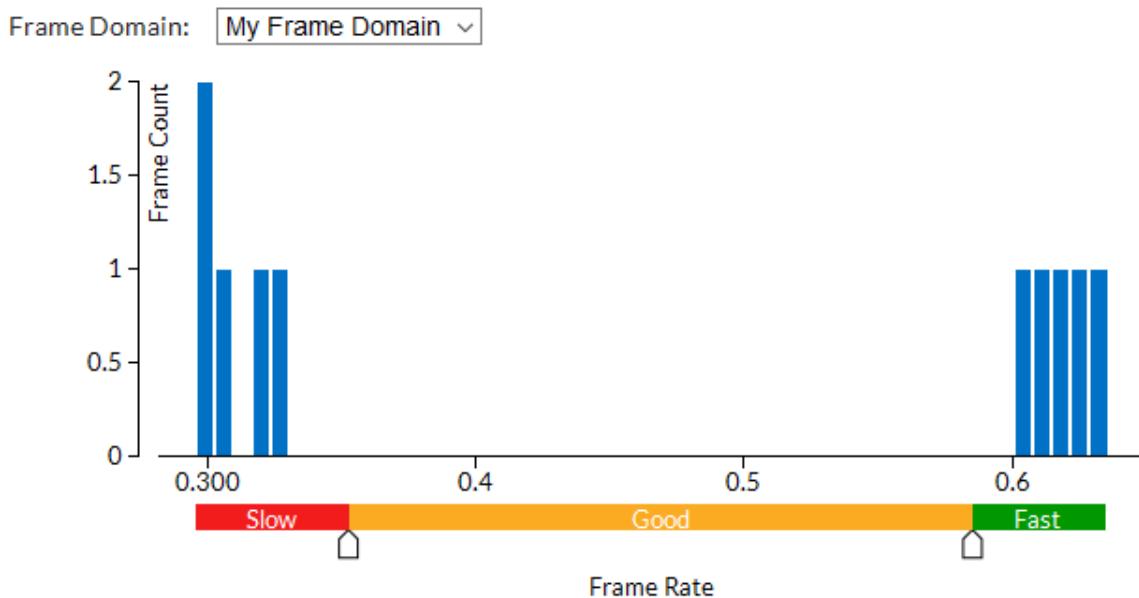
Explore frame analysis options provided by the Intel® VTune™ Profiler, which are especially useful for identifying a long latency activity.

You may use the [Frame API](#) to mark the start and finish of the code regions executed repeatedly (*frames*) in such applications as simulators with a time step loop, computations with a convergence loop, game applications computing next graphics frame, and so on. VTune Profiler analyzes the marked code regions and identifies bottlenecks in your application caused by slow or fast frame rate. To interpret the performance data provided during the frames analysis, you may follow the steps below:

1. Analyze summary frames statistics.
2. Analyze the timeline.
3. Identify the hotspot code sections.

Analyze Summary Frames Statistics

Click the **Summary** tab to open the **Summary** window and analyze the **Frames Rate Histogram**. Hover over a bar to see the total number of frames in your application executed with a specific frame rate. High number of slow or fast frames signals a performance bottleneck.



VTune Profiler automatically sets up thresholds for slow and fast frame rate. But you may change them, if needed, by dragging the slider at the bottom of the histogram. The thresholds you set will be automatically applied to all subsequent results for this project.

Switch to the **Bottom-up** window and group the data in the grid by **Frame Domain/Frame Duration Type/Function/Call Stack**:

The screenshot shows the 'Hotspots' tab in the 'Hotspots by CPU Utilization' section of the Intel VTune Profiler. The 'Bottom-up' tab is selected. The 'Grouping' dropdown is set to 'Frame Domain / Frame Duration Type / Function / Call Stack'. The table displays the following data:

Frame Domain / Fra...	CPU Time ▾	Frame Time	Frame Count	Module	Function (Full)
▼ My Frame Domain	1133.089s	24.254s	10		
▼ Slow	752.778s	16.154s	5		
▶ multiply3	752.690s			matrix.icc	multiply3
▶ ParallelMultipl	0.040s			matrix.icc	ParallelMulti...
▶ __pthread_crea	0.030s			libpthread...	__pthread_cre...
▶ OS_SyscallDo	0.010s			libc-dyna...	OS_SyscallDo
▶ ThreadFunction	0.008s			matrix.icc	ThreadFuncti...
▶ Fast	380.311s	8.100s	5		
▶ [No frame domain -]	0.130s				

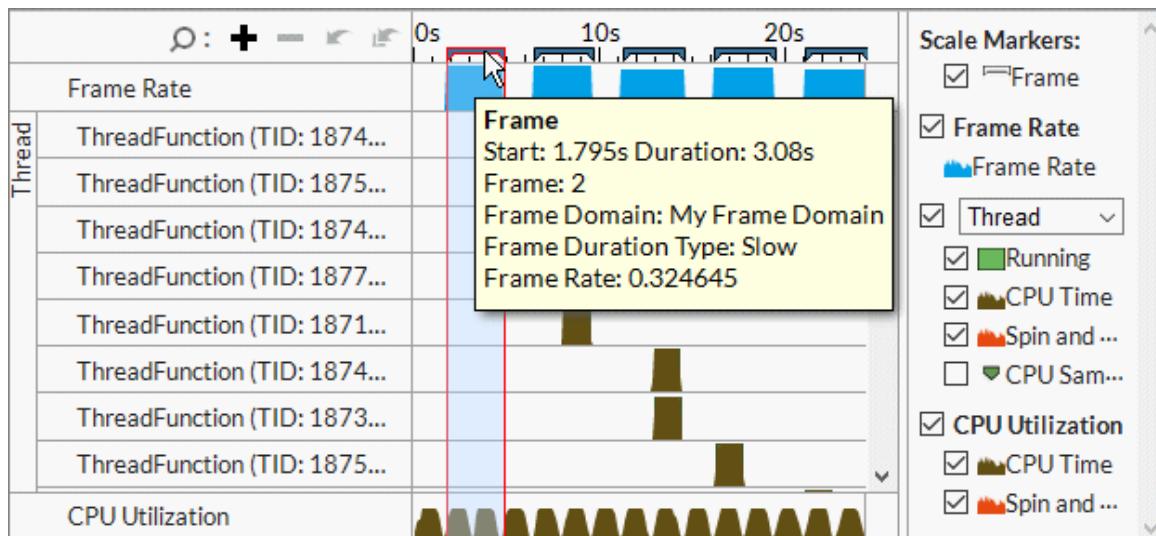
This grouping displays frame analysis metrics including the Frame Time that is the wall time during which frames were active. Focus on the frames with the highest Frame Time values. Expand a frame domain node to see frames grouped by frame duration. You may select slow frames, right-click and select **Filter In by Selection** to filter out all the data other than slow frames in this domain. Then you may group the data back by **Function/Call Stack** to see the functions that took most of the time in these slow frames:

The screenshot shows the 'Hotspots' tab in the 'Hotspots by CPU Utilization' section of the Intel VTune Profiler. The 'Bottom-up' tab is selected. The 'Grouping' dropdown is set to 'Function / Call Stack'. The table displays the following data:

Function / Call Stack	CPU Time ▾		Spin Time	Overhead Time
	Effective Time by Utilization	Idle Poor Ok Ideal Over		
▶ multiply3	752.690s	Idle	0s	0s
▶ ParallelMultiply	0.040s	Poor	0s	0s
▶ __pthread_create_2	0.030s	Ok	0s	0s
▶ OS_SyscallDo	0.010s	Ideal	0s	0s
▶ ThreadFunction	0.008s	Over	0s	0s

Analyze the Timeline

In the **Bottom-up** window, analyze the frame data represented in the **Timeline** pane. If you filtered the grid by slow frames, the Timeline data is also automatically filtered to display data for the selected frames:



The scale area displays frame markers. Hovering over a marker opens a tooltip with details on frame duration, frame rate and so on.

The **Frame Rate** band displays how the frame rate is changing over time. To understand the cause of the bottleneck, identify sections with the Slow or Fast frame types and analyze the **CPU Utilization** data. For example, you may detect the Slow frame rate for the section with the poor CPU utilization or thread contention. In this case, you may parallelize the code to utilize CPU resources more effectively or optimize the thread management.

To identify a hotspot function containing the critical frame from the Timeline view, select the range with the Slow or Fast frame rate. VTune Profiler highlights the selected frame in the **Bottom-up** grid.

Identify the Hotspot Code Sections

Double-click a critical function executing a slow/fast frame to view its source code. By default, the VTune Profiler highlights the code line in this function that took the most CPU time to execute.

Task Analysis

Focus your performance analysis on a task - program functionality performed by a particular code section.

Use the Intel® VTune™ Profiler to analyze the following types of tasks:

- **ITT API tasks:** Analyze performance of particular code regions (*tasks*) if your target uses the [Task API](#) to mark task regions and you enabled the **Analyze user tasks, events and counters** option during the analysis type configuration
- **Platform tasks:** Analyze tasks enabled for analysis of Ftrace* events, Atrace* events, Intel Media SDK programs, OpenCL™ kernels, and so on.

Enabling Task Analysis

Prerequisites:

- Use the ITT Task API to insert calls in your code and define the tasks.
- [Configure your analysis target](#).

1. Click the



(standalone GUI)/



(Visual Studio IDE) **Configure Analysis** button on the VTune Profiler toolbar.

2. Choose the analysis type from the **HOW** pane.
3. Select the **Analyze user tasks, events, and counters** option.
4. Click the **Start** button to run the analysis.

VTune Profiler collects data detecting the marked tasks.

Analyze the collected results to identify the task regions and task duration versus application performance over time.

To interpret the data provided during the user task analysis, you may use the following options:

- [Identify most critical tasks](#).
- [Analyze slow tasks per function](#).
- [Analyze tasks per threads](#).

Identify Most Critical Tasks

Start exploring the collected data with the **Summary** window where the **Top Tasks** section provides a list of tasks that took most of the time to execute.

Top Tasks

This section lists the most active tasks in your application.

Task Type	Task Time ^②	Task Count ^②	Average Task Time ^②
Task1	7.042s	1	7.042s
Task2	5.031s	1	5.031s
Task3	3.011s	1	3.011s
MediaSDK	1.504s	1	1.504s
Task4	1.002s	1	1.002s
[Others]	0.501s	1	0.501s

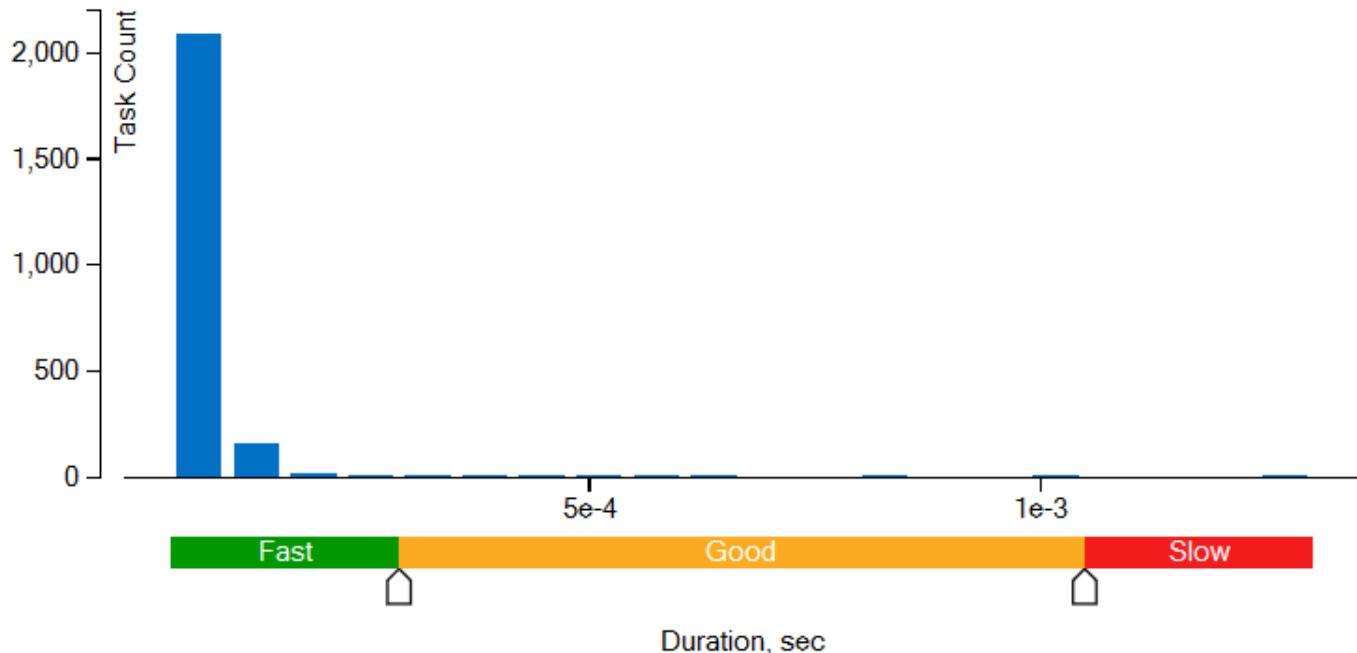
*N/A is applied to non-summable metrics.

If you collected data for Ftrace/Atrace tasks using the System Overview or a custom analysis with Ftrace/Atrace events selected, the **Summary** window also provides the **Task Duration Histogram** that helps you identify slow tasks:

Task Duration Histogram

This histogram shows the total number of task instances executed with a specific duration. High number of slow instances may signal a performance bottleneck.

Task Type: **heci_bus_event_work**



Use the **Task Type** drop-down list to switch between different tasks and analyze their duration. Based on the thresholds set up for the task duration, you can understand whether the duration of the selected task is acceptable or slow.

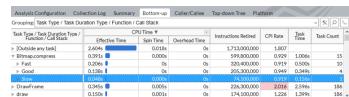
Analyze Slow Tasks per Function

Click a task type in the **Top Tasks** section to switch to the grid view (for example, **Bottom-up** or **Event Count**) grouped by the **Task Type** granularity. The task selected in the **Summary** window is highlighted. For example, for ITT API tasks collected during the Threading analysis the **Bottom-up** grid view is grouped by **Task Type/Function/Call Stack**:

Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform						
Grouping: Task Type / Function / Call Stack		CPU Time ▾			Wait Time by Utilization ▾	
Task Type / Function / Call Stack	Effective Time by Utilization	Spin Time	Overhead Time	Idle	Poor	Ok
func4_task	2.923s	0s	0s	0.000s		3.001s
func6_task	2.916s	0s	0s	0.000s		3.001s
func2_task	2.903s	0s	0s	0.000s		3.002s
[Outside any task]	0s	0.002s	0s	9.024s		

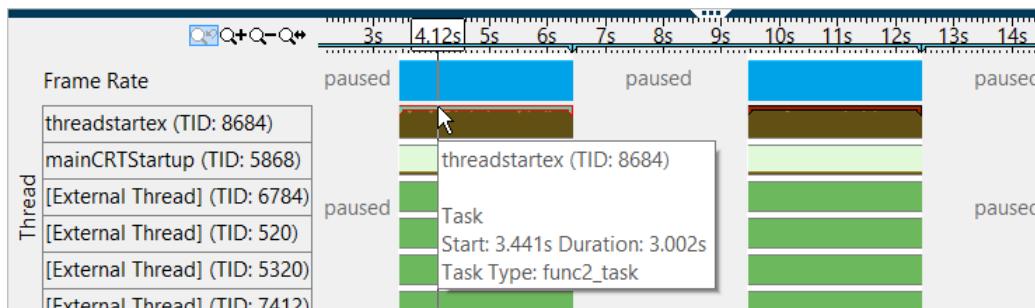
In the example above, the `func4_task` task has the longest duration - 2.923 seconds. You may expand the node to see the function this task belongs to. Double-click the function to analyze the source code in the Source view.

For Ftrace/Atrace tasks collected during the System Overview analysis, you may select the **Task Type/Task Duration Type/Function/Call Stack** granularity and explore functions executed while a slow task instance was running. You may double-click the function to open its source code and analyze the most time-consuming source lines.



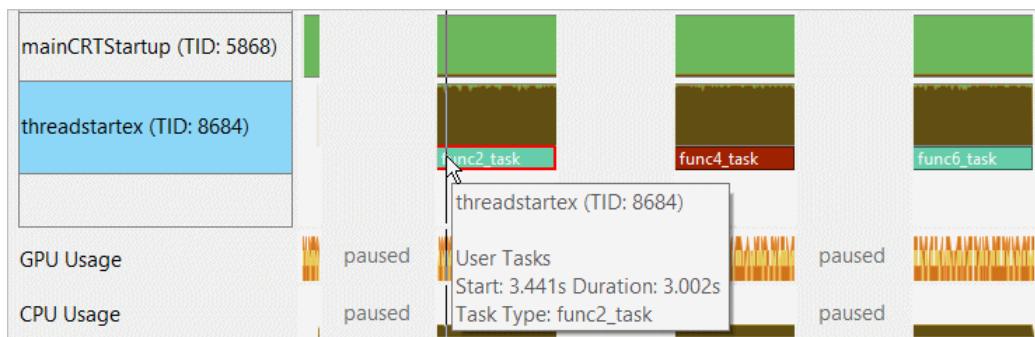
Analyze Tasks per Threads

To analyze a duration of each task instance, explore the **Timeline** view:

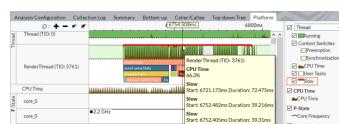


User tasks are shown on the timeline with yellow markers. Hover over a task marker for task execution details. In the example above, the `func2_task` started at the 3.4th second of the application execution on the thread `threadstartex` (TID: 8684) and lasted for 3.002 seconds.

If you collected platform-wide metrics, you may switch to the **Platform** window and identify threads responsible for particular tasks. Each task shows up in the **Thread** section as a separate layer.



For Ftrace/Atrace tasks, the **Platform** view provides an option to enable **Slow Tasks** markers and explore the CPU utilization, GPU usage and power consumption at the moment of slow tasks execution:



If several tasks were executed on a thread in parallel, a stack of tasks is displayed.

See Also

Pane: Timeline

[Switch Viewpoints](#)

[Linux* and Android* Kernel Analysis](#)
to configure Systrace*/FTrace* tasks

[Intel® Media SDK Program Analysis](#)
to see Intel Media SDK tasks on the timeline

[Examples of CSV Format and Imported Data](#)
displayed as tasks

Instrumentation and Tracing Technology APIs

View Instrumentation and Tracing Technology (ITT) API Task Data in Intel® VTune™ Profiler

Control Data Collection

Explore options to run, stop, cancel, or pause your performance analysis with Intel® VTune™ Profiler.

Run Analysis from Standalone Interface

To run an analysis:

1. Create/open a VTune Profiler project.

From the **Configure Analysis** window:

- Specify an analysis system from the **WHERE** pane.
- Specify an analysis target from the **WHAT** pane.
- Select an analysis type from the **HOW** pane.

2. At the bottom of the **Configure Analysis** window, click the



Start button to run the analysis.

To pause the analysis at the application start and then manually resume it when required, click the



Start Paused button.

NOTE

The **Start** button may be disabled if you either did not specify the analysis target or selected the analysis type that is not supported by your processor.

Run Analysis from the Microsoft Visual Studio* IDE

1. Open your target in Visual Studio.

2. Build your target in the Release mode in the development environment of your choice.

3. Click the



Configure Analysis button on the VTune Profiler toolbar to choose and configure an analysis type in the **Configure Analysis** window.

4. At the bottom of the **Configure Analysis** window, click the



Start button to run the analysis.

To pause the analysis at the application start and then manually resume it when required, click the



Start Paused button.

Stop/Cancel the Analysis

When you run the analysis, the command toolbar at the bottom of the **Configure Analysis** window is updated with a set of buttons for managing the data collection.

- To stop the analysis, click the



Stop button or press **Ctrl-C**.

VTune Profiler stops collecting data and opens the analysis result.

- To cancel the analysis, click the



Cancel button.

VTune Profiler stops collecting data and displays the warning message: **Collection was cancelled by the user. The data cannot be displayed.**

Open Analysis Results

VTune Profiler analyzes the target, finalizes the result, and opens the collected data in the default viewpoint. The data collection results (*.vtune file) show up in the Solution Explorer (for the VTune Profiler integrated into Visual Studio)/Project Navigator (standalone) under the project folder, in the alphabetical order. For executable files imported to the Visual Studio project, the data result node appears at the solution level. Double-click a result to open the collected data in the default viewpoint.

NOTE

- You can provide a meaningful name for the result (for example, application name) for better identification. To do this, select the result, right-click and choose **Rename**. The file extension *.vtune cannot be changed.
 - To change the result name template or the default directory for result location, go to **Tools > Options** (or **Options...** in the standalone interface menu) and select **Intel VTune Profiler version > Result Location** from the left pane of the **Options** dialog box.
 - You may program hot keys to start/stop a particular analysis. For more details, see <http://software.intel.com/en-us/articles/using-hot-keys-in-vtune-amplifier-xe/>.
-

See Also

[Pause Data Collection](#)

[Generate Command Line Configuration from GUI](#)

[VTune Profiler Filenames and Locations](#)

[Finalization](#)

Finalization

Finalization is the process by which Intel® VTune™ Profiler converts the collected data to a database, resolving symbol information, and pre-computes data to make further analysis more efficient and responsive. VTune Profiler finalizes data automatically when data collection completes.

VTune Profiler provides three basic finalization modes:

- Full** mode is used to perform the finalization on unchanged sampling data on the target system. This mode takes the most time and resources to complete, but produces the most accurate results.
- Fast** (default) mode is used to perform the finalization on the target system using algorithmically reduced sampling data. This greatly reduces the finalization time with a negligible impact on accuracy in most cases.
- Deferred** mode is used to collect the sampling data and calculate the binary checksums to perform the finalization on another machine. After data collection completes, you can finalize and open the analysis result on the host system. This mode may be useful for profiling applications on targets with limited computational resources, such as IoT devices, and finalizing the result later on the host machine.

- **None** option is used to skip finalization entirely and to not calculate the binary checksums. You can also finalize this result later, however, you may encounter certain limitations. For example, if the binaries on the target system have changed or have become unavailable since the sampling data collection, binary resolution may produce an inaccurate or missing result for the affected binary.

Modify the Finalization Mode

By default, the **Fast** finalization mode is used for any analysis configuration. If you need to change it, do the following:

1. Click the



Configure Analysis button.

2. From the **WHERE** pane, click the



Browse button, choose a target system and specify required details.

3. From the **WHAT** pane, click the



Browse to choose an appropriate target type.

4. Expand the **Advanced** section on the **WHAT** pane and scroll down to select the required finalization mode, for example: **Deferred to use another system**.

NOTE

When the analysis result is collected and open, you can always check the used finalization mode in the **Summary** view > **Collection and Platform Info** section.

Re-Finalize Results

You may want to re-finalize a result to:

- update symbol information after changes in the search directories settings
- resolve the number of **[Unknown]**-s in the results

Beware that re-finalization can lead to wrong results if you do not have the original binaries for your target on the machine performing the re-finalization; for example, if you recompiled the target. The re-finalization deletes the old database and then picks up the newer versions of the binaries. Since the collector raw data does not contain a binary checksum, the VTune Profiler does not know when a binary has changed and attempts to resolve the symbols matching the old addresses against the new binary. As a result, the VTune Profiler may unwind stacks incorrectly and resolve samples to the wrong functions. To avoid this, make sure you configured the search directories to use the correct files.

By default, the VTune Profiler saves the raw collector data after finalization. You may choose to remove these data to reduce the size of the result file if you do not plan to re-finalize this result in the future. To remove the raw collector data, from the Microsoft Visual Studio* menu go to **Tools > Options > Intel VTune Profiler <version> > General** pane and select the **Remove raw collector data after result finalization** option. To remove the raw collector data in the standalone interface, click the



menu button and select **Options... > General**.

To re-finalize a result in the Microsoft Visual Studio* IDE, select the result in the Solution Explorer, right-click and select **Re-resolve and Open**.

To re-finalize a result in the standalone VTune Profiler interface:

1. Click the



menu button and select **Open > Result....**

The **Select Result** dialog box opens.

2. Navigate to the required result *.vtune file you want to re-finalize and click **OK**.

The selected result opens in the default **viewpoint**.

3. Click the **Analysis Configuration** tab.
4. Click the



Re-resolve button on the command bar.

Intel® VTune™ Profiler repeats result finalization. If you updated the list of search directories in the **Binary/Symbol Search** or **Source Search** dialog boxes, the VTune Profiler uses the latest version of these directories to search for supporting binary/source/symbol files.

See Also

[finalization-mode](#)

vtune option

[Search Directories](#)

Pause Data Collection

You can configure the analysis run to launch the application but start collecting data after some delay or pause the data collection in the middle of the application execution. This is useful if you do not want to include all the warm-up activities in the analysis results or you want the data collection to start when a specific event occurs (for example, message box or mouse click). Intel® VTune™ Profiler provides several options to pause and resume your analysis:

- Start running an application with the [data collection paused](#), and then manually resume the data collection when required.
- Use the **Pause/Resume** button to pause the data collection at any time of application execution.
- Use the [Pause/Resume API](#) to insert calls into your code to start and stop the analysis.

Start Data Collection Paused, Then Manually Resume

To manually start and resume the analysis, do the following:

1. [Create/open a project](#).
2. Click the



Configure Analysis button on the toolbar.

The **New Amplifier Result** result tab opens.

3. [Specify and configure your analysis target](#) on the **WHAT** pane.
4. Switch to the **HOW** pane and click the [Browse](#) button to select and configure, if required, an [analysis type](#).
5. Click the



Start Paused button on the command bar.

VTune Profiler runs the application. The **Start Paused** button is replaced with the **Resume** button.

- Click the **Resume** button on the command bar to start data collection.

Use the Pause/Resume Button to Pause at Any Time of Application Execution

- Click the **Start** button on the command bar to run the selected analysis.

When analysis starts running, the command bar is updated with a set of analysis management buttons.

- When you need to pause the collection, click the **Pause** button on the command bar.

VTune Profiler collects no data but your application keeps running. The **Start** button on the command bar is replaced with the **Resume** button.

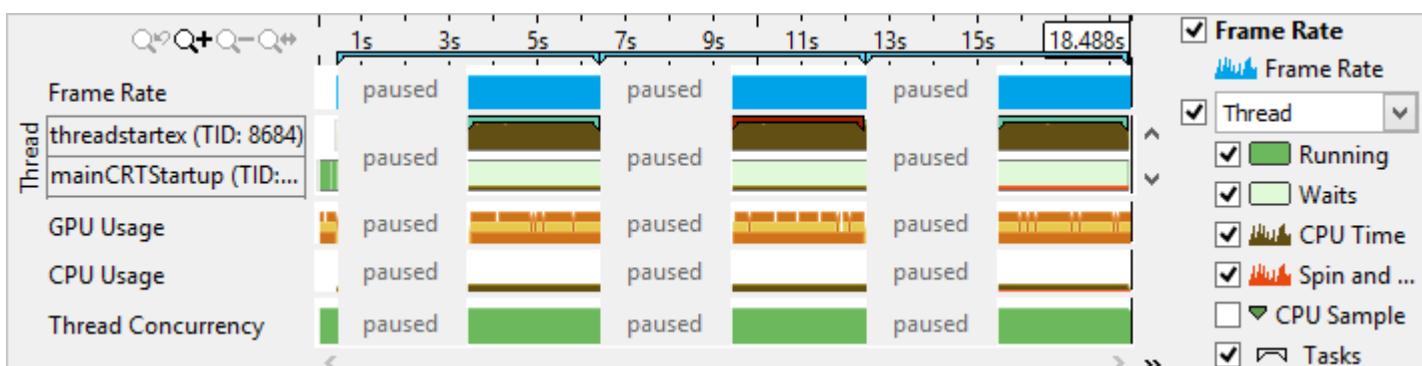
- When you need to resume the data collection, click the **Resume** button on the command bar.

VTune Profiler continues collecting data.

Use the Pause/Resume API to Insert Calls into Your Code to Start and Stop the Analysis

To get details on using the Pause/Resume API, see the [Collection Control API](#) topic.

When the data collection is complete, the VTune Profiler displays paused regions in the Timeline pane as follows:



See Also

[start-paused vtune option](#)

[Problem: Unexpected Paused Time](#)

[Toolbar: Configure Analysis](#)

Limit Data Collection

Specify a predefined amount of data to collect by setting up the expected result size or collection time.

This prevents from collecting a large amount of data that may slow down the data processing. For example, it may happen when running Threading Analysis on frequently contended applications or when analyzing long profiles.

Typically, the default maximum amount of raw data used by the Intel® VTune™ Profiler for the result file is enough to identify a problem.

When the data size limit is reached and the data collection is suspended, click the



Stop button on the [command toolbar](#) at the bottom of the **Configure Analysis** window. VTune Analyzer proceeds with the analysis of the collected data. If you want to extend the data collection for your target application for future analysis runs, you may modify the default size limit for collected data as follows:

- Click the



Configure Analysis button on the VTune Profiler toolbar.

2. Select a required target system from the **WHERE** pane and a target type from the **WHAT** pane.
3. From the **Advanced** section of the **WHAT** pane, use the **Limit collected data by** group of options and choose any of the following mechanisms:
 - **Result size from collection start, MB:** Set the maximum possible result size (in MB) to collect. VTune Profiler will start collecting data from the beginning of the target execution and suspend data collection when the specified limit for the result size is reached. For unlimited data size, specify 0.
 - **Time from collection end, sec:** Set the timer enabling the analysis only for the last seconds before the target run or collection is terminated. For example, if you specified 2 seconds as a time limit, the VTune Profiler starts the data collection from the very beginning but saves the collected data only for the last 2 seconds before you terminate the collection.

Limiting data collection to the beginning or end of the target execution reduces the size of the raw data gathered by the VTune Profiler and enables you to quickly start analyzing collection results. If you want to keep the default data size limit but continue collecting data on the next portion of the target execution, run the analysis after a delay using the [Start Paused](#) option.

See Also

[Set Up Analysis Target](#)

[data-limit](#)

vtune option

[ring-buffer](#)

vtune option

[Manage Analysis Duration from Command Line](#)

Generate Command Line Configuration from GUI

Use the Intel® VTune™ Profiler to automatically generate a command line for an analysis configuration and copy this line to the buffer for running from a terminal window. You can use this approach to run the generated command line configuration on a different system.

To generate and apply a command line configuration:

Prerequisites: Set up your project.

1. Run the VTune Profiler graphical interface.
2. Click the



(standalone GUI)/

(Visual Studio IDE)**Configure Analysis** toolbar button to choose and configure your analysis.

The **Configure Analysis** window opens.

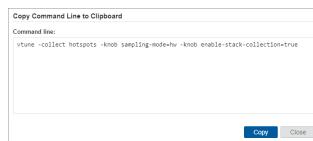
3. From the **HOW** pane, choose a predefined or custom analysis type and configure the required settings.
4. Click the



Command Line button at the bottom of the window.

The **Copy Command Line to Clipboard** dialog box opens providing the command line required to launch the selected analysis type configuration. Options with default values are hidden.

For predefined analysis types, the `-collect <analysis-type>` option is applied:



For [custom analysis](#) types, the `-collect-with <collector-type>` option is applied:

Copy Command Line to Clipboard

Command line:

```
vtune -collect-with runsa -knob enable-stack-collection=true -knob enable-trip-
counts=true -knob enable-user-tasks=true -knob event-
config=CPU_CLK_UNHALTED.THREAD:sa=200000,CPU_CLK_UNHALTED.REF_TSC:sample:sa=200000
0,INST_RETIRED.ANY:sample:sa=200000,CPU_CLK_UNHALTED.REF_XCLK:sa=100003,CPU_CLK_UN
HALTED.ONE_THREAD_ACTIVE:sa=100003,UOPS_RETIRIED.RETIRE_SLOTS:sample:sa=2000003,FP_A
RITH_INST_RETIRIED.SCALAR_SINGLE:sa=2000003,FP_ARITH_INST_RETIRIED.128B_PACKED_SINGLE
:sa=2000003,FP_ARITH_INST_RETIRIED.256B_PACKED_SINGLE:sa=2000003,FP_ARITH_INST_RETIR
ED.SCALAR_DOUBLE:sa=2000003,FP_ARITH_INST_RETIRIED.128B_PACKED_DOUBLE:sa=2000003,FP_
ARITH_INST_RETIRIED.256B_PACKED_DOUBLE:sa=2000003,UOPS_EXECUTED.X87:sa=2000003,UOPS_
RETIRIED.RETIRE_SLOTS:sa=2000003,UOPS_EXECUTED.THEAD:sa=2000003
```

Copy **Close**

5. Click the **Copy** button to copy the command line to the clipboard.
6. Paste the copied command line to the shell.
7. Optionally, edit the application data in the command line as required.

If you analyze a remote application from the local host, make sure to:

- Set up your remote [Linux](#) or [Android](#) target system for data collection.
- Specify the correct path to the remote application in the command line.
- Use the `-target-system=<system_details>` option to specify your remote target address (for Linux) or device name (for Android). For example:

```
host>./vtune -target-system=ssh:user@hostName -collect hotspots -- myapp
```

8. Press **Enter** to launch the analysis from the command line.

Vtune Profiler collects the data and [saves the result](#) to the analysis result directory under your working directory.

9. Open your data collection result file in the GUI or as a text-based command line report.

NOTE

To enable analyzing the source code, make sure to copy the required symbol/source files from your remote machine and update the [search directories](#) in the **Binary/Symbol Search** or **Source Search** dialog boxes.

See Also

[Collect Data on Remote Linux* Systems from Command Line](#)

[target-system](#)

[vtune option](#)

[Intel® VTune™ Profiler Command Line Interface](#)

[Manage Data Views](#)

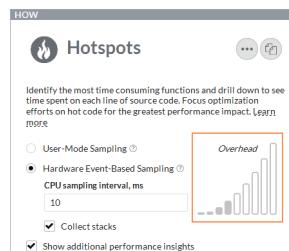
Minimize Collection Overhead

Explore configuration options provided by the Intel® VTune™ Profiler that incur collection overhead and increase the result size.

If required, consider disabling or modifying these options either by editing the predefined analysis configuration or by creating a new custom analysis type:

Hotspots Sampling Mode

When you select the Hotspots analysis, you can choose between the User-Mode Sampling (higher overhead) and Hardware Event-Based Sampling (lower overhead). The **Overhead** diagram on the right adjusts to your settings and shows how each of them impacts on the collection overhead:



Collect Context Switches

This option enables collection of thread context switches for [hardware event-based sampling collection](#) and is available in a custom hardware event-based sampling analysis configuration.

To disable/modify this option for custom analysis:

From GUI:

1. In the **Configure Analysis** window > **HOW** pane, click the Browse button and select the **Custom Analysis > your_custom_analysis** type.
2. In the custom analysis configuration, de-select the **Collect context switches** option.

From CLI:

Use the `-knob enable-stack-collection=false` option. For example:

```
vtune -collect-with runsa -knob enable-stack-collection=false -knob event-
config=CPU_CLK_UNHALTED.REF_TSC:sa=1800000,CPU_CLK_UNHALTED /home/test/sample
```

Sampling Interval

This option configures the amount of wall-clock time the VTune Profiler waits before collecting each sample. The smaller the **Sampling Interval**, the larger the number of samples collected and written to the disk. The minimal value of the sampling interval depends on the system:

- 10 milliseconds for systems with a single CPU
- 15 milliseconds for systems with multi-core CPUs

To disable/modify the sampling interval value:

From GUI:

1. In the **Configure Analysis** window > **HOW** pane, click the Browse button and select an analysis type, for example, **Hotspots** and use the **Hardware Event-based Sampling** mode.
2. For the **CPU sampling interval, ms** option, specify a required value.

From CLI:

Use the `-knob sampling-interval=<value>` option. For example:

```
vtune -collect-with runss -knob sampling-interval=100 -knob cpu-samples-mode=stack -knob signals-mode=stack -knob waits-mode=stack -knob io-mode=stack /home/test/sample
```

Stack Size

This option is used to specify the size of a raw stack (in bytes) to process during hardware event-based sampling collection. Zero value means unlimited size. Possible values are numbers between 0 and 2147483647.

To disable/modify this option:

From GUI:

1. In the **Configure Analysis** window > **HOW** pane, click the Browse button and select the **Custom Analysis > your_custom_analysis** type.
2. In the custom configuration, decrease the **Stack size, in bytes** value.

From CLI:

Use the `-stack-size` option, for example:

```
vtune -collect-with runsa -knob enable-stack-collection=true -knob stack-size=8192 -knob enable-call-counts=true -app-working-dir /home/samples/nqueens_fortran -- /home/samples/nqueens_fortran/nqueens_parallel
```

See Also

[Custom Analysis](#)

knob

vtune option

stack-size

vtune option

Import External Data

Correlate interval or discrete data provided by an external collector with the regular data provided by the Intel® VTune™ Profiler.

For example, you can see how the data captured from SoCs or peripheral devices (camera, touch screen, sensors, and so on) correlate with VTune Profiler metrics collected for your analysis target.

VTune Profiler can load and process the following data types:

- Interval data with start time and end time
- Samples with a set of counters

Data may be optionally bound to process and thread ID.

To add external performance statistics to a VTune Profiler result:

1. Launch a custom data collector in parallel with the selected VTune Profiler analysis type.
2. Convert the collected data to the CSV format and import it to the VTune Profiler.

Launch a Custom Data Collector

Collect custom performance data using one of the following modes:

- **Application mode:** You can leverage the statistics collected by your target application to enhance the VTune Profiler analysis. For example, a part of your application has many instances executed many times in one run and some of these instances exhibit a performance problem. You can retrieve time frames where problems occur from your application log file and supply this data to the VTune Profiler.

- **Custom collector mode:** If you cannot/do not want to collect statistics directly by your application during the VTune Profiler analysis, you may either create a custom collector or use an existing external collector (for example, ftrace, ETW, logcatthat) and launch it from the VTune Profiler. To enable this mode, configure a VTune Profiler analysis type to use the [Custom collector](#) option and specify a command starting your external collector.

Convert Custom Data to the CSV Format and Import It to VTune Profiler

To import the externally collected data to the VTune Profiler:

1. Convert the collected custom data to a `csv` file with a [predefined structure](#).

To do this for the custom collector mode, you need to configure the collector to output the data in the required CSV format using the `VTUNE_HOSTNAME` environment variable that identifies the name of the current host required for the `csv` file format. For the application mode, you may identify the hostname from the **Computer name** field provided in the **Summary** window for your result, or from the `summary` command line report.

2. Import the `csv` file to the VTune Profiler result using any of the following options:

in GUI:

- a. Open the VTune Profiler result that was launched in parallel with the external data collection.
 - b. Open the **Analysis Target** tab, or **Analysis Type** tab.
 - c. Click the **Import from CSV** button on the command toolbar on the left.
- The **Choose a File to Import** dialog box opens.
- d. Navigate to the required `csv` file and click **Open**. You may import several `csv` files at a time.

NOTE

Importing a `csv` file to the VTune Profiler result does not affect symbol resolution in the result. For example, you can safely import a `csv` file to a result located on a system where module and debug information is not available.

in CLI: use the `import` option as follows:

```
vtune -r <existing result dir> -import <path to csv file>
```

VTune Profiler processes the data gathered by its own collectors and the external application and provides an integrated picture of your code performance in its standard data views, such as the **Timeline** pane, **Bottom-up** pane and others.

NOTE

If you develop a custom collector yourself, you may use the `VTUNE_DATA_DIR` environment variable to make your collector identify the VTune Profiler result directory and automatically save the custom collection result (in the CSV format) to this directory. In this case, external statistics will be imported to the VTune Profiler result automatically.

See Also

[Use a Custom Collector](#)

[Create a CSV File with External Data](#)

[custom-collector](#)

`vtune` option

Use a Custom Collector

Extend a standard Intel® VTune™ Profiler performance analysis and launch a custom data collector directly from the VTune Profiler.

Your custom collector can be an application you analyze with the VTune Profiler or a collector that can be launched with the VTune Profiler.

To use a custom collector with the VTune Profiler and correlate the collected data:

1. Configure the custom collector .
2. Launch the custom collector .

Configure the Custom Collector

VTune Profiler sets several environment variables that can be used by a custom collector to manage the data collection and collected results:

Environment Variable Provided by VTune Profiler	Enables Custom Collector To Do This								
AMPLXE_DATA_DIR	Identify a path to the VTune Profiler analysis result. The custom collector uses this path to save the output <code>csv</code> file and make it accessible for the VTune Profiler that adds the <code>csv</code> data to the native VTune Profiler result.								
AMPLXE_HOSTNAME	Identify the full hostname of the machine where data was collected. The hostname is a mandatory part of the csv file name .								
AMPLXE_COLLECT_CMD	Manage a custom data collection. The custom collector may receive the values listed below. After any of these commands the custom collector should exit immediately and return control to the VTune Profiler.								
NOTE									
For each command, the custom collector will be re-launched.									
	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">start</td><td style="padding: 5px;">Start custom data collection. If required, the custom collector may create a background process.</td></tr> <tr> <td style="padding: 5px;">stop</td><td style="padding: 5px;">Stop data collection (background process), convert data to a <code>csv</code> file, copy it to the result directory (specified by <code>AMPLXE_DATA_DIR</code>) and return control to the VTune Profiler.</td></tr> <tr> <td style="padding: 5px;">pause</td><td style="padding: 5px;">Temporarily pause data collection. This value is optional.</td></tr> <tr> <td style="padding: 5px;">resume</td><td style="padding: 5px;">Resume data collection after pause. This value is optional.</td></tr> </table>	start	Start custom data collection. If required, the custom collector may create a background process.	stop	Stop data collection (background process), convert data to a <code>csv</code> file, copy it to the result directory (specified by <code>AMPLXE_DATA_DIR</code>) and return control to the VTune Profiler.	pause	Temporarily pause data collection. This value is optional.	resume	Resume data collection after pause. This value is optional.
start	Start custom data collection. If required, the custom collector may create a background process.								
stop	Stop data collection (background process), convert data to a <code>csv</code> file, copy it to the result directory (specified by <code>AMPLXE_DATA_DIR</code>) and return control to the VTune Profiler.								
pause	Temporarily pause data collection. This value is optional.								
resume	Resume data collection after pause. This value is optional.								
AMPLXE_COLLECT_PID	Identify a Process ID of the application to analyze. VTune Profiler sets this environment variable to the PID of the root target process. The custom collector may use it, for example, to filter the data. VTune Profiler sets this variable to the process only when profiling in the Launch Application or Attach to Process mode. For system-wide profiling, the value is empty. When your profiled application spawns a tree of processes, the <code>AMPLXE_COLLECT_PID</code> variable points to the PID of the launched or attached								

Environment Variable Provided by VTune Profiler	Enables Custom Collector To Do This
	process. This is important to know in case of using a script to launch a workload since you may need to use your own means to pass the child process PID to the custom collector.

The templates below demonstrate an interaction between the VTune Profiler and a custom collector:

Example in Python:

```
import os

def main():
    cmd = os.environ['AMPLXE_COLLECT_CMD']
    if cmd == "start":
        path = os.environ['AMPLXE_DATA_DIR']
        #starting collection of data to the given directory
    elif cmd == "stop":
        pass #stopping the collection and making transformation of own data to CSV if necessary

main()
```

Example in Windows CMD shell:

```
if '%AMPLXE_COLLECT_CMD%' == 'start' goto start
if '%AMPLXE_COLLECT_CMD%' == 'stop' goto stop
echo Invalid command
exit 1

:start
rem Start command in non-blocking mode
start <my collector command to start the collection> '%AMPLXE_DATA_DIR%\data_file.csv
exit 0

:stop
<my collector command to stop the collection>
exit 0
```

Launch the Custom Collector

To launch a custom collector from the VTune Profiler GUI:

1. Click the



Configure Analysis button on the toolbar.

The [Configure Analysis](#) window opens.

2. Make sure the correct target system and target type are selected in the **WHERE** and **WHAT** panes.
3. In the **Advanced** section of the **WHAT** pane, edit the **Custom collector** field to add a command launching your external collector, for example:
 - on Windows*: python.exe C:\work\custom_collector.py
 - on Linux*: python home/my_collectors/custom_collector.py
4. From the **HOW** pane, select the required analysis type, for example, **Hotspots**.
5. Configure available analysis options as you need.

6. Click the **Start** button to launch the VTune Profiler analysis and collect custom data in parallel.

VTune Profiler does the following:

- a. Launches the target application, if any, in the suspended mode.
- b. Launches the custom collector in the attach (or system-wide) mode.
- c. Switches the application to the active mode and starts profiling.

If your custom collector cannot be launched in the attach mode, the collection may produce incomplete data.

To launch a custom collector from the command line:

Use the `-custom-collector=<string>` option.

Command Line Examples:

This example runs Hotspots analysis in the default user-mode sampling mode and also launches an external script collecting custom statistics for the specified application:

Windows:

```
vtune -collect hotspots -custom-collector="python.exe C:\work\custom_collector.py" -- notepad.exe
```

Linux:

```
vtune -collect hotspots -custom-collector="python /home/my_collectors/custom_collector.py" -- my_app
```

This example runs VTune Profiler event-based sampling collector and also uses an external system collector to identify product environment variables:

Windows:

```
vtune -collect-with runsa -custom-collector="set | find \"AMPLXE\"'" -- notepad.exe
```

Linux:

```
vtune -collect-with runsa -custom-collector="set | find \"AMPLXE\"'" -- my_app
```

NOTE

If you [use your target application as a custom collector](#), you do not need to apply the **Custom collector** option but make sure your application uses the following variables:

- `AMPLXE_DATA_DIR` environment variable to identify a path to the VTune Profiler result directory and save the output `csv` file in this location.
 - `AMPLXE_HOSTNAME` environment variable to identify the name of the current host and use it for the `csv` file name.
-

See Also

[Import External Data](#)

[Create a CSV File with External Data](#)

[Cookbook: Core Utilization in DPDK Apps](#) tracing with the custom collector
tracing with the custom collector

[Intel® VTune™ Profiler Command Line Interface](#)

Create a CSV File with External Data

Intel® VTune™ Profiler can process and integrate performance statistics collected externally with a [custom collector](#) or with your target application in parallel with the native VTune Profiler analysis. To achieve this, provide the collected custom data as a `csv` file with a predefined structure and save this file to the VTune Profiler result directory.

VTune Profiler can load and process the following data types:

- [Interval data with start time and end time](#)
- [Samples with a set of counters](#)

To make the VTune Profiler interpret the custom statistics from the `csv` file, make sure the file format meets the following requirements:

File Name

`csv` filename should specify the hostname where your custom collector gathered the data, following these format requirements:

Filename format: `[user-defined]-hostname-<hostname-of-system>.csv`

Where:

- `[user-defined]` is an option string, for example, describing the type of data collected
- `-hostname-` is a **required** text that must be specified verbatim
- `<hostname-of-system>` is the name of the system where the data is collected. If you use a custom collector you can retrieve the hostname by using the `VTUNE_HOSTNAME` environment variable. If you create a CSV file to import into an existing result, you can either refer to the [Summary](#) window that provides the required hostname in the **Collection and Platform Info** section > **Computer name**, or check the corresponding `vtunesummary` report: `vtune -r <result> -R summary`.

Example: `phases-hostname-octagon53.csv`

NOTE

If the hostname in the `csv` file name is not specified or specified incorrectly, the VTune Profiler displays the imported data with the following limitations:

- Event timestamps are represented in the UTC format.
 - Only global data (not attributed to specific threads/processes) are displayed.
-

Format for Interval Values

Interval data may be optionally bound to a thread ID. VTune Profiler represents data not bound to a particular thread (there are no TID values in the `csv` file) as *frames*. Data bound to a thread (there are TID values in the `csv` file) is represented as *tasks*.

For imported *interval* values, use 5 columns, where the order of columns is important:

`name,start_tsc.[QPC|CLOCK_MONOTONIC_RAW|RDTSC|UTC],end_tsc,[pid],[tid]`

Column Name	Description
<code>name</code>	Name of an event.
<code>start_tsc.[QPC CLOCK_MONOTONIC_RAW RDTSC UTC]</code>	Event start timestamp. This column name has a <code>QPC CLOCK_MONOTONIC_RAW</code> , <code>RDTSC</code> or <code>UTC</code> suffix that indicates the type of a timestamp counter:

Column Name	Description
	<ul style="list-style-type: none"> Specify QPC (QueryPerformanceCounter) on Windows* OS if the performance counter is used and specify CLOCK_MONOTONIC_RAW on Linux* OS if clock_gettime(CLOCK_MONOTONIC_RAW) is used. Specify RDTSC if the RDTSC counter is used. To obtain RDTSC: <ul style="list-style-type: none"> For Microsoft* Compiler and Intel® Compiler, use _rdtsc() intrinsic For GCC* compiler, copy the following function to your code and call it where necessary: <pre>#include <stdint.h> int64_t rdtsc() { int64_t tstamp; #if defined(__x86_64__) asm("rdtsc\n\t" "shlq \$32,%rdx\n\t" "or %%rax,%%rdx\n\t" "movq %%rdx,%0\n\t" : "=g"(tstamp) : : "rax", "rdx"); #elif defined(__i386__) asm("rdtsc\n": "=A"(tstamp)); #else #error NYI #endif return tstamp; }</pre> Specify UTC if date and time is used. Expected format is YYYY-MM-DD hh:mm:ss.sssss, where the number of decimal digits is arbitrary.
end_tsc	Event end timestamp.
pid	<p>Process ID, provided optionally. Absence of a value in this field does not affect how a result is imported except for extremely rare cases when the following conditions are all met:</p> <ul style="list-style-type: none"> Thread ID is reused by the operating system within the collection time frame. Different threads with the same thread ID generate records for the CSV file. Timestamps are inaccurate and data may be attributed to more than one thread with the same thread ID. <p>You may specify this field as an empty value within the data, or skip it from both file header and data entirely.</p>
tid	<p>Thread ID, provided optionally. If a value is specified in this field, the interval will be interpreted as a Task; otherwise, interval will be interpreted and shown as a Frame.</p> <p>You may specify this field as an empty value within the data, or skip it from both file header and data entirely.</p>

Examples

Format for Discrete Values

You can import two types of discrete values:

- Cumulative data type (for example, distance, hardware event count), specified with the .COUNT suffix in the csv file
- Instantaneous data type (for example, power consumption, temperature), specified with the .INST suffix in the csv file

The following format is required:

tsc.[QPC|CLOCK_MONOTONIC_RAW|RDTSC|UTC],CounterName1.COUNT|INST[,CounterName2.COUNT|INST],[pid],[tid]

Column Name	Description
tsc.[QPC CLOCK_MONOTONIC_RAW RDTSC UTC]	<p>Event start timestamp. This column has a QPC CLOCK_MONOTONIC_RAW, RDTSC, or UTC suffix that indicates the type of a timestamp counter:</p> <ul style="list-style-type: none"> • Specify QPC (QueryPerformanceCounter) on Windows* OS if the performance counter is used and specify CLOCK_MONOTONIC_RAW on Linux* OS if clock_gettime(CLOCK_MONOTONIC_RAW) is used. • Specify RDTSC if the RDTSC counter is used. Use __rdtsc() intrinsic to obtain RDTSC on Windows. To obtain RDTSC on Linux, copy the following function to your code and call it where necessary: <pre>#include <stdint.h> int64_t rdtsc() { int64_t tstamp; #if defined(__x86_64__) asm("rdtsc\n\t" "shlq \$32,%rdx\n\t" "or %%rax,%%rdx\n\t" "movq %%rdx,%0\n\t" : "=g"(tstamp) : : "rax", "rdx"); #elif defined(__i386__) asm("rdtsc\n", "=A"(tstamp)); #else #error NYI #endif return tstamp; }</pre> <ul style="list-style-type: none"> • Specify UTC if date and time is used. Expected format is YYYY-MM-DD hh:mm:ss.sssss, where the number of decimal digits is arbitrary.
CounterName1	Name of the event. Each counter has a separate column. COUNT suffix is used to specify a cumulative counter value. INST suffix is used to specify instantaneous counter values.
pid	<p>Process ID, provided optionally. Absence of a value in this field does not affect how a result is imported except for extremely rare cases when the following conditions are all met:</p> <ul style="list-style-type: none"> • Thread ID is reused by the operating system within the collection time frame. • Different threads with the same thread ID generate records for the csv file. • Timestamps are inaccurate and data may be attributed to more than one thread with the same thread ID.

Column Name	Description
	You may specify this field as an empty value within the data, or skip it from both file header and data entirely.
tid	Thread ID, provided optionally. If a value is specified in this field, the interval will be interpreted as a Task; otherwise, interval will be interpreted and shown as a Frame. You may specify this field as an empty value within the data, or skip it from both file header and data entirely.

Examples

Additional Requirements

- Make sure each `csv` file contains only one table. If you need to load several tables, create several `csv` files with one table per file.
- Use commas as value separators.
- Use RDTSC, UTC or performance counter (`QueryPerformanceCounter` on Windows OS and `CLOCK_MONOTONIC_RAW` on Linux OS) to specify events timestamp.

See Also

[Import External Data](#)

[Use a Custom Collector](#)

[Examples of CSV Format and Imported Data](#)

```
import
vtune option
```

Import Linux Perf* Trace with VTune Profiler Metrics

If you have your own performance monitoring system based on Linux Perf (for example, as part of your data center infrastructure) and cannot collect data with the Intel® VTune™ Profiler, you can still use the VTune Profiler for data analysis as follows:

1. Select a VTune Profiler analysis type that is of interest to you.
2. Use VTune Profiler to get a set of Linux Perf options and apply them to a Perf collection on your target system.
3. Import the generated Linux Perf trace into a VTune Profiler project and start analysis.

Select a VTune Profiler Analysis Type

VTune Profiler provides a rich set of [predefined analysis types](#) targeting particular performance problems. Each analysis type contains a selected list of low-level performance events and high-level metrics based on them. For example, Microarchitecture Exploration analysis collects all required PMU (Performance Monitoring Unit) events from CPU cores needed for TMA methodology. The Memory Access analysis has a set of both core and uncore PMU events needed for memory-related performance metrics (like DRAM bandwidth).

Using a native Linux Perf interface to collect all needed low-level PMU events may be complicated, so consider reusing the VTune Profiler configuration targeted for Perf collection (driverless mode).

Run VTune Profiler to Get Linux Perf Options for Analysis

When the VTune Profiler runs a performance data collection in the [driverless mode](#), it uses a Linux Perf command line and logs it inside the result folder in the `<result-folder>/data.0/perfcmd` file. To get a correct set of Perf options, do the following:

1. Install the VTune Profiler on any Linux system with a similar hardware configuration (the same CPU family) as the system where real performance profiling is planned to be run.
2. Run a VTune Profiler analysis of your interest to generate perfcmd file with Perf options:

```
$ vtune-cl -r <result-folder> -collect <analysis-type> -finalization-mode=none -d 1
```

For example, for the Microarchitecture Exploration run:

```
vtune-cl -r bogus_result -collect uarch-exploration -finalization-mode=none -d 1
```

The <result-folder>/data.0/perfcmd file with all necessary Linux Perf options is generated.

NOTE

- You do not run any real workload here. The only purpose of this run is to generate the perfcmd file.
 - VTune Profiler license is not required for this step since you only collect data without opening it.
-

3. Open the perfcmd file and copy-paste its content to a Linux Perf command invocation on your real target system.

NOTE

Your Perf tool should contain a patch from <https://github.com/torvalds/linux/commit/f92da71280fb8da3a7c489e08a096f0b8715f939#diff-809984534aa420619413fdf4c260605d>. In Linux kernel version >= 4.19, this patch is applied out of the box, in earlier versions you need to manually apply it and recompile the Perf tool.

4. Run the Linux Perf configuration on your target system.

Import the Linux Perf Trace into a VTune Profiler Project

1. Create a VTune Profiler project or open an existing one.
2. Click the



Import Result toolbar button.

The **Import File and Create a Result** window opens.

3. Select **Import a single file** option and navigate to the Linux Perf trace file.

VTune Profiler imports the trace and opens the result in the default viewpoint. You may [switch between viewpoints](#) to apply the most relevant. For example, use the Microarchitecture Exploration viewpoint for the Microarchitecture Exploration analysis.

See Also

[Set Up Project](#)

[Import Results and Traces into VTune Profiler GUI](#)

Examples of CSV Format and Imported Data

Explore examples of the performance data gathered with an external collector and imported into an Intel® VTune™ Profiler project in the CSV format.

- Examples for importing interval data:

- CSV file with the performance counter timestamp
- CSV file with the system counter timestamp
- CSV file with interval data bound to a process
- Command line report for imported interval data bound to a process
- CSV file with interval data not bound to a particular process
- Command line report for imported interval data not bound to a process
- Examples for importing discrete data:
 - CSV file with the performance counter timestamp
 - CSV file with the system counter timestamp
 - CSV file with discrete data not bound to a particular process
 - Command line report for imported discrete data

Examples for Importing Interval Data

Example 1: CSV File with the Performance Counter Timestamp

```
name,start_tsc.QPC,end_tsc.pid,tid
frame1,2,30,,
frame1,33,59,,
taskType1,3,43,1,1
taskType2,5,33,1,1
taskType1,46,59,1,1
taskType2,45,54,1,1
```

VTune Profiler will process data with missing PID and TID as frames. Data with the PID and TID specified will be processed as tasks.

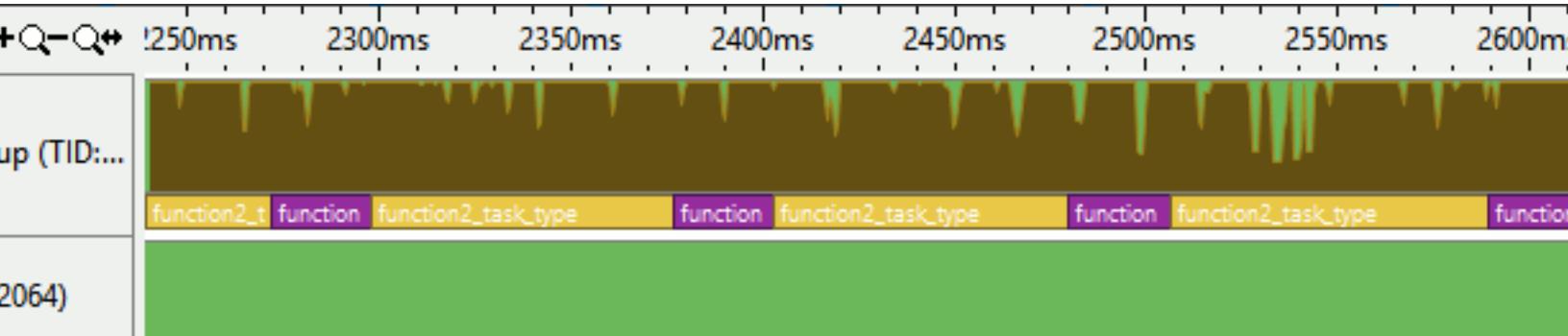
Example 2: CSV File with the System Counter Timestamp

```
name,start_tsc.UTC,end_tsc.pid,tid
Frame1,2013-08-28 01:02:03.0004,2013-08-28 01:02:03.0005,,
Task,2013-08-28 01:02:03.0004,2013-08-28 01:02:03.0005,1234,1235
```

Example 3: CSV File with Interval Data Bound to a Process

```
name,start_tsc.TSC,end_tsc.pid,tid
function1_task_type,419280823342846,419280876920231,12832,11644
function2_task_type,419280876920231,419281044717992,12832,11644
function1_task_type,419281044745822,419281102121452,12832,11644
function2_task_type,419281102121452,419281277898762,12832,11644
function1_task_type,419281277935812,419281342158661,12832,11644
function2_task_type,419281342158661,419281527040239,12832,11644
```

VTune Profiler processes this data as tasks (TID and PID values are specified) and displays the result in the **Platform** window as follows:



Example 4: Command Line Report for Imported Interval Data Bound to a Process

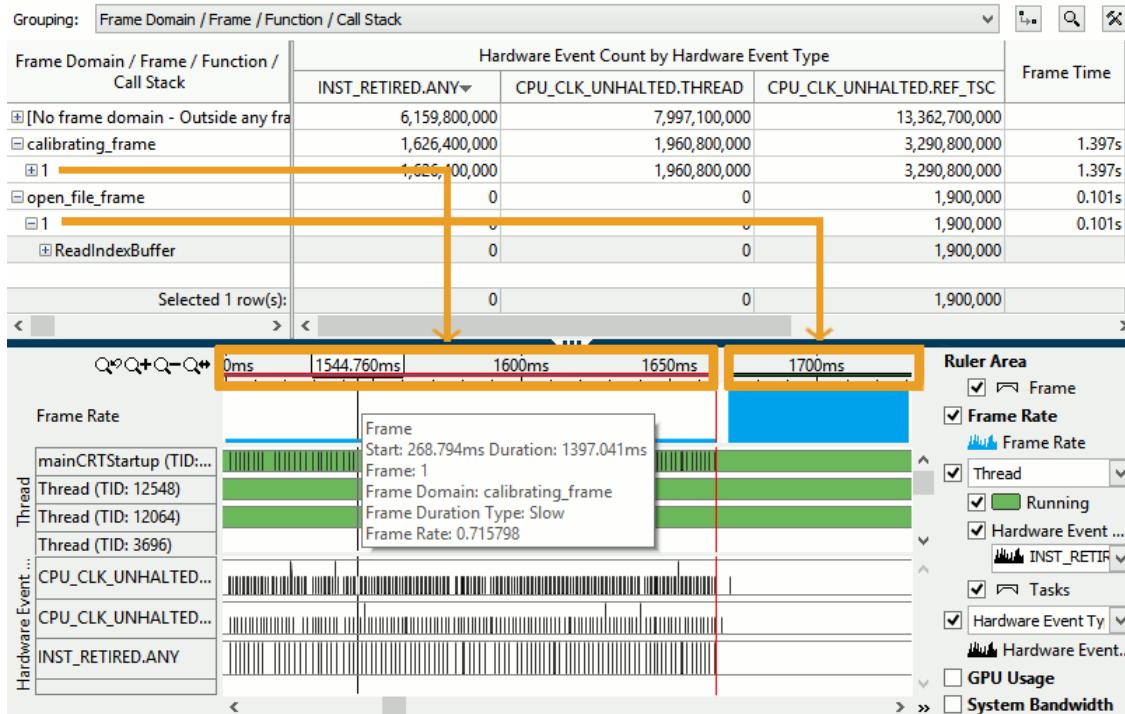
In this example, the `hotspots` report shows counters bound to a specific process/thread grouped by tasks:

```
vtune -R hotspots -group-by=task -r my_result
vtune: Using result path 'my_result'
vtune: Executing actions 50 % Generating a report
Task Type      CPU Time:Self Task Time:Self Overhead Time:Self Spin Time:Self Thread
Counter:victim_counter:Self Thread Counter:victim_counter_x2:Self
-----
[Outside any task]          0           0           0
0                          0           2           0
ITT Task                  0           0           0.009
0                          2           6           0
victim_task                0           0           0.000
0                          0           0           0
vtune: Executing actions 100 % done
```

Example 5: Interval Data Not Bound to a Particular Process

```
name,start_tsc.TSC,end_tsc.pid,tid
calibrating_frame,419743756747826,419747241283878,,,
open_file_frame,419747251423510,419747504506086,,
```

VTune Profiler processes this data as frames (there are no TID and PID values specified) and displays the result as follows:



With the VTune Profiler, you can easily correlate the frame data in the **Timeline** pane and grid view.

Example 6: Command Line Report for Imported Interval Data Not Bound to a Process

In this example, the `hotspots` report shows counters not bound to a specific thread/process grouped by frame domain:

```
vtune -R hotspots -group-by=frame-domain -r my_result
vtune: Using result path 'my_result'
vtune: Executing actions 50 % Generating a report
Frame Domain      Frame Time:Self Counter:global_counter:Self Counter:global_counter_x2:Self
```

```
-----  
cuscol_frame    0.126      4          8  
cuscol_utc_frame 0.126      4          8  
vtune: Executing actions 100 % done
```

Examples for Importing Discrete Data

Example 1: CSV File with the Performance Counter Timestamp

```
tsc.QPC,MyCounter1.COUNT,MyCounter2.INST,pid,tid  
2,1,3,1,1  
5,2,5,1,1  
10,3,3,1,1  
23,10,7,1,1
```

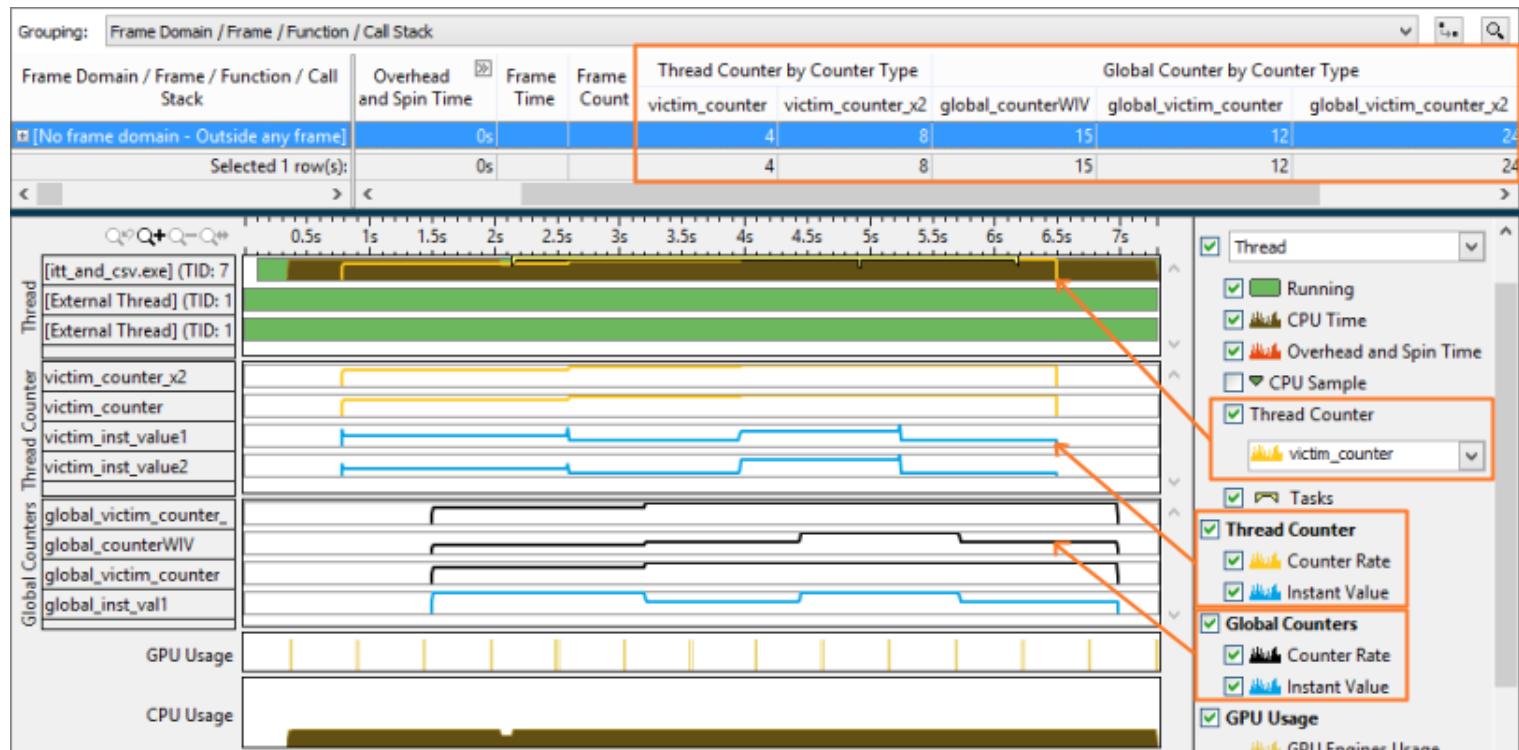
Example 2: CSV File with the System Counter Timestamp

```
tsc.UTC,MyCounter1.COUNT,MyCounter2.COUNT,pid,tid  
2013-08-28 01:02:03.0004,1234,,1234,1235  
2013-08-28 01:02:03.0005,1234,,1234,1235  
2013-08-28 01:02:03.0006,,1000234,,
```

Example 3: CSV File with Discrete Data Not Bound to a Particular Process

```
tsc.TSC,global_inst_val1.INST,global_counterWIV.COUNT,pid,tid  
78912463824135,3,6,,  
78916553573577,6,9,,  
78919519641325,3,12,,  
78922574591880,6,18,,  
78925599513489,3,21,,
```

VTune Profiler processes this data and displays the result as follows:



Discrete cumulative counter values, both thread-specific and global (not thread-specific), are provided in the grid view and in the **Timeline** pane in yellow. Instantaneous counter values, thread-specific and global, are displayed in blue in the **Timeline** pane only.

NOTE

To view global counter values in the grid, make sure to select a generic (not thread specific) grouping level like **Frame Domain/Frame/Function/Call Stack**.

Example 4: Command Line Report for Imported Discrete Data

This example provides the `hw-events` report with external discrete data (counters) integrated into a VTune Profiler hardware event-based sampling analysis result `cl_result.vtune`:

```
vtune -R hw-events -group-by=process -r my_result
vtune: Using result path 'my_result'
vtune: Executing actions 50 % Generating a report
Process          Counter:victim_counter:Self  Counter:victim_counter_x2:Self
-----
itt_and_csv.exe           2                   4
vtune: Executing actions 100 % done
```

See Also

[Import External Data](#)

[Create a CSV File with External Data](#)

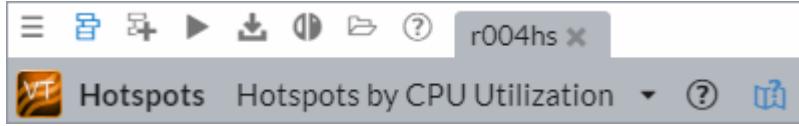
[Use a Custom Collector](#)

`import`
option

Manage Data Views

When the analysis run is complete, the Intel® VTune™ Profiler generates a result that is automatically opened in the default viewpoint. The location of the result files is specified in the **Configure Analysis** window.

A viewpoint typically contains the following elements:

Format	Description
Result Tab	<p>This is a container of all other viewpoint elements. This tab has the same name as the VTune Profiler result file. The result tab name uses the <code>r@@@{@at}</code> format, where <code>@@@</code> is an incremented result number starting with 000 and <code>at</code> is the analysis type.</p> <p>For example:</p>  <p>r004hs is the fifth result run in this project and provides data for the Hotspots (hs) analysis. The Hotspots is the analysis type name. Hotspots by CPU Utilization is the name of the viewpoint selected via the down arrow. Use this arrow to switch to other viewpoints available for this analysis result.</p>

Format	Description
Windows	<p>Each result tab includes a number of windows presenting collected data from different perspectives. Each window has a corresponding tab. To ease your navigation, some windows are synchronized: when you select an element in a window, the same element is automatically selected in other windows of the same viewpoint. The list of windows depends on the selected viewpoint.</p> <p>Each window has a corresponding context help topic available via F1 button or  icon.</p> <p>NOTE Context help as part of this product help is available on the web only. You may also download a copy of this help from the VTune Profiler documentation archive.</p>
Panes	Each window typically includes two or three panes, such as Call Stack pane , Timeline pane , and others.

NOTE

For a brief overview on a particular viewpoint, click the question mark icon at the viewpoint name.

All the data views make your analysis more convenient and manageable with the following options:

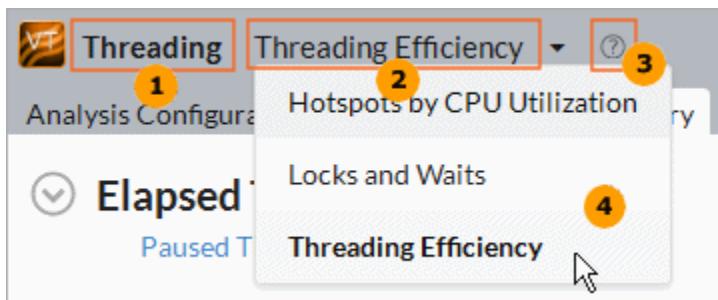
Switch Viewpoints

Use a viewpoint, a pre-set configuration of Intel® VTune™ Profiler's data views, to focus on specific performance problems.

NOTE

By default, VTune Profiler shows no viewpoints, or a managed selection of viewpoints that may be helpful for the specific analysis type. You can enable the display of all applicable viewpoints by enabling the **Show all applicable viewpoints** option in the [Options pane](#).

When you select a viewpoint, you select a set of performance metrics the Intel® VTune™ Profiler shows in the windows of the result tab. To select the required viewpoint, click the down arrow:



1 Name of the analysis type you ran.

2 Name of the current viewpoint. Click the down arrow next to the viewpoint name to open a drop-down menu with a choice of applicable viewpoints.

- 3** Context-sensitive help icon for the current viewpoint.
- 4** Viewpoint drop-down menu that displays a list of viewpoints available for the current analysis type.

Explore the table below to understand which viewpoints are available for each analysis type:

Viewpoint	Description
Hotspots by CPU Utilization	Helps identify <i>hotspots</i> - code regions in the application that consume a lot of CPU time. CPU time is broken down into CPU utilization states: idle, poor, fair, and good.
Threading Efficiency	Shows how your multi-threaded application is utilizing available CPU cores and helps identify the possible causes of ineffective utilization. Use this view to find threads waiting too long on synchronization objects (locks) or identify scheduling overhead.
Microarchitecture Exploration	Helps identify where the application is not making the best use of available hardware resources. This viewpoint displays metrics derived from hardware events. The Summary window reports overall metrics for the entire execution along with explanations of the metrics. From the Bottom-up and Top-down Tree windows you can locate the hardware issues in your application. Cells are highlighted when potential opportunities to improve performance are detected. Hover over the highlighted metrics in the grid to see explanations of the issues.
Hardware Events	Displays statistics of monitored hardware events: estimated count and/or the number of samples collected. Use this view to identify code regions (modules, functions, code lines, and so on) with the highest activity for an event of interest.
Memory Usage	Helps understand how effectively your application uses memory resources and identify potential memory access related issues like excessive access to remote memory on NUMA platforms, hitting DRAM or Interconnect bandwidth limit, and others. It provides various performance metrics for both the application code and memory objects arrays.
HPC Performance Characterization	Helps understand how effectively your application uses CPU, memory, and floating-point operation resources. Use this view to identify scalability issues for Intel OpenMP and MPI runtimes as well as next steps to increase memory and FPU efficiency.
Input and Output	Shows input/output data, CPU and bus utilization statistics correlated with the execution of your target. Use this view to identify long latency of I/O requests, explore call stacks for I/O functions, analyze slow I/O requests on the timeline and identify imbalance between I/O and compute operations.
GPU Compute/Media Hotspots	Helps identify GPU tasks with high GPU utilization and estimate its effectiveness. It is particularly useful for SYCL computing tasks, analysis of the OpenCL™ kernels and Intel Media SDK tasks . Use this view to identify the most time-consuming GPU computing tasks, analyze GPU tasks execution over time, explore the GPU hardware metrics per GPU architecture blocks, and so on.
FPGA Hotspots	Helps identify the FPGA and CPU tasks with high utilization. Use this view to assess FPGA time spent executing kernels, overall time for memory transfers between the CPU and FPGA, and how well a workload is balanced between the CPU and FPGA.

Viewpoint	Description
GPU Rendering	Provides platform-wide CPU/GPU utilization and efficiency statistics collected with GPU Rendering analysis (preview) including dedicated support for the Xen virtualization platform.
Platform Power Analysis	Helps identify where the application is generating idle and wake-up behavior that can lead to inefficient use of energy. Where possible, it provides data from both the OS and hardware perspective, such as the detailed C-state residency report that shows the OS requested time in deep sleep states compared to the actual residency the hardware indicated.

See Also

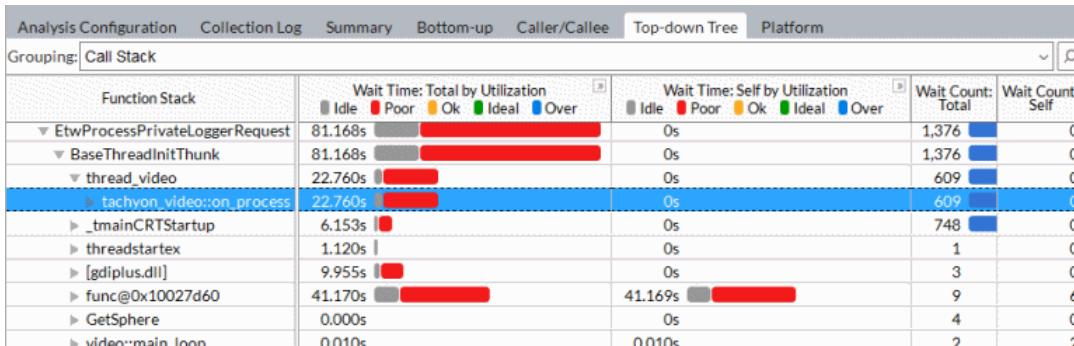
[Interpret Energy Analysis Data with Intel® VTune™ Profiler](#)

[Analyze Performance](#)

Control Window Synchronization

*Correlate the data displayed in the **Bottom-up** window for each program unit (bottom-up analysis) and in the **Top-down Tree** window for an overall impact of each element together with its callees (top-down analysis).*

The **Top-down Tree** window includes **Self Time** and **Total Time** columns for each data column in the **Bottom-up** window:



In the Threading Efficiency example above, columns in the **Top-down Tree** window match the columns in the **Bottom-up** window as follows:

Bottom-up Window	Top-down Tree Window
Wait Time by Thread Concurrency	Wait Time: Total by Thread Concurrency Wait Time: Self by Thread Concurrency
Wait Count	Wait Count: Total Wait Count: Self

The **Bottom-up** window provides only *Self* type of data (function without callees). In the grid, *Self time/Count* column headers do not have :*suffix*.

The *Total* type of data (function + all callees' *Self* data) is provided in the <*data*>:**Total** column and unique to the **Top-down Tree** window. In the example above, these are the **Wait Time:Total by Utilization** and **Wait Count:Total** columns.

Self time for a program unit in the **Bottom-up** window equals the sum of Self time values for the same program unit in different call sequences in the **Top-down Tree** window.

See Also

[Window: Bottom-up](#)

[Window: Top-down Tree](#)

[View Stacks](#)

View Stacks

Manage the Intel® VTune™ Profiler view to display call stacks for user and system functions and estimate an impact of each stack on the performance metrics.

Intel VTune Profiler provides call stack information in the **Call Stack** pane, **Bottom-up** pane, **Top-down Tree**, and **Caller/Callee** pane. You may use the following options to manage and analyze stacks in different views:

- [Change stack layout](#)
- [Navigate between stacks](#)
- [View stacks per metric](#)
- [View system functions in the stack](#)
- [View source for a stack function](#)

Change Stack Layout

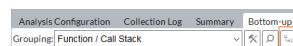
Manage the stack representation in the grid (**Bottom-up** or **Top-down Tree** pane) by using the



/



stack layout toolbar button.

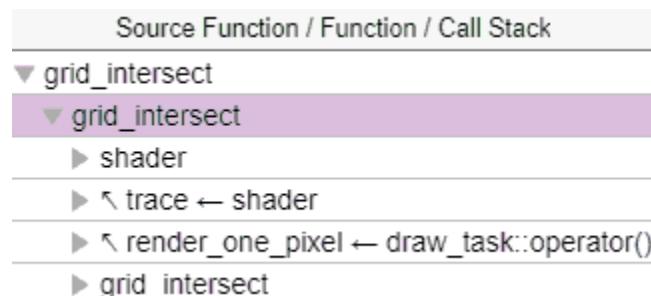


The button dynamically changes according to the selected layout. For example, if the chain layout is selected for the view, the button changes to show an option to choose a tree layout, and vice versa.

Chain layouts



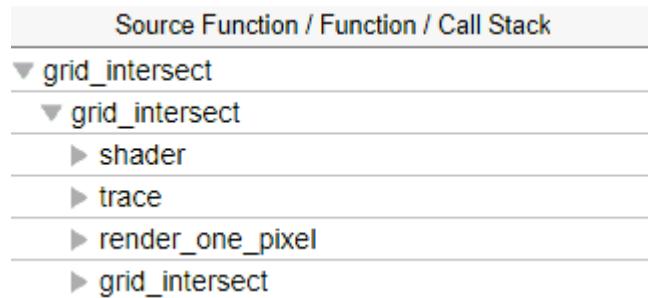
are typically more useful for the bottom-up view:



While tree layouts



are more natural for the top-down view:

**NOTE**

Chain layout in the **Top-down Tree** pane is possible only if there is no branching AND when all values of data columns are the same for the parent and for the child.

Navigate Between Stacks

To view stacks for the selected program unit, estimate stack contribution, and identify the most performance-critical stack, use the **Call Stack** pane and click the next/previous

/



arrows.

To view information on several stacks or program units, Ctrl-click to select these stacks or program units in the **Bottom-up** or **Top-down Tree** pane. The **Call Stack** pane shows the highest contributing stack from all the selected stacks, with the contribution calculated based on the sum of all selected stacks. All the stacks related to the selection are added to the tab and you can navigate to them using the next/previous

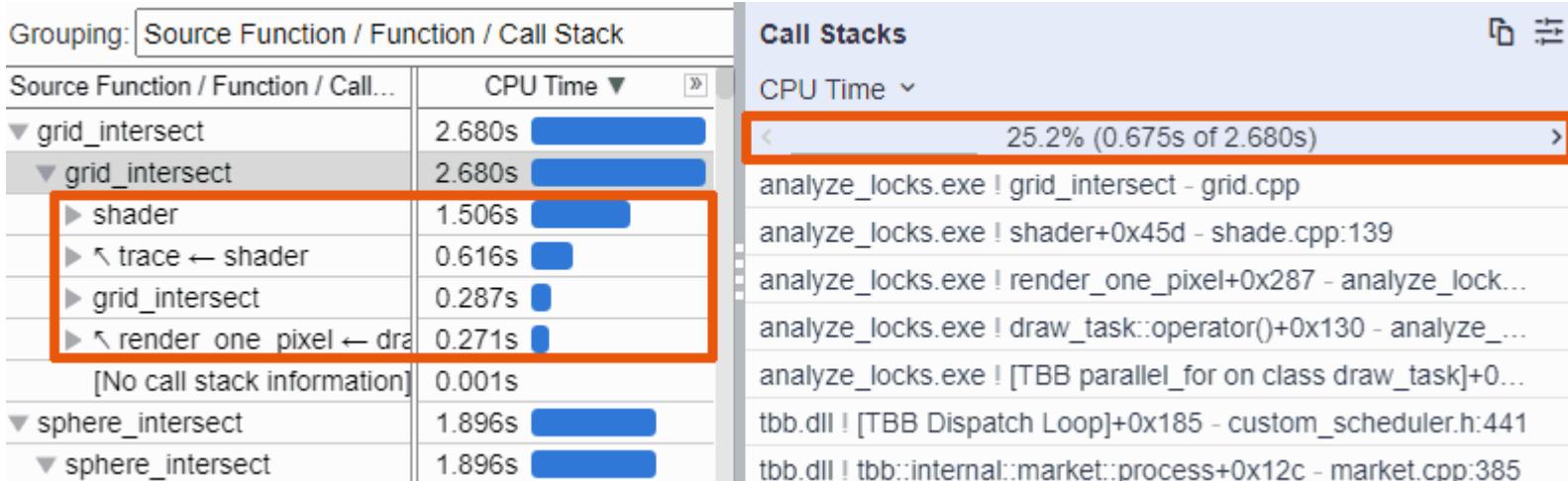
/



arrows.

Note that though each stack in the **Bottom-up** pane corresponds to a call stack provided in the **Call Stack** pane, the number of tree branches in the **Bottom-up** grid does not necessarily equal the number of stacks in the **Call Stack** pane. Since the stack in the **Bottom-up** pane is function-based and the stacks in the **Call Stack** pane are line-number-based, the number of stacks in these views may differ.

For example, in the screen capture below, the **Bottom-up** pane shows two stacks for the `grid_intersect` function whereas the **Call Stack** pane shows that 17 stacks exist.



View Stacks per Metric

Use the drop-down menu in the **Call Stack** pane, to choose the stack type for the selected program unit.

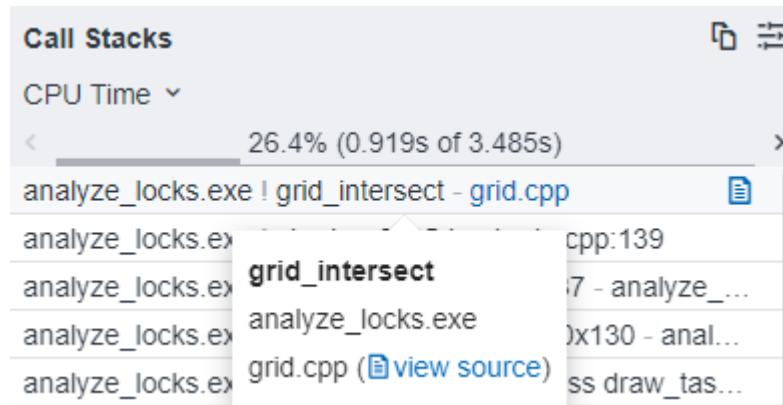
For example, when a synchronization object is selected in the Threading analysis result, you can set the **Call Stack** pane to show the stacks where that object was created, signaled or waited for.

View System Functions in the Stack

To control whether you need the system functions show up in the stacks in the grid and **Call Stack** pane, use the **Call Stack Mode** menu provided on the filter toolbar.

View Source for a Stack Function

Hover over any item in the **Call Stack** pane to get information on the related source file and code line. To go to that line, click the **View Source** hyperlink. The source file opens in the **Source/Assembly** window on the code that generated the item in the selected row.



For example, in a Threading analysis result, if you double-click the topmost item of the **Wait Time (Sync Object Creation)** stack, the related source file opens on the source line that created the corresponding synchronization object.

If the source code is not found, you can either locate it manually, or open the **Assembly** pane for this program unit.

If you select a system function, the **Source/Assembly** window opens the source file of the system function if it is available. If not, it shows the disassembly for the binary file containing this system function.

See Also

[Window: Bottom-up](#)

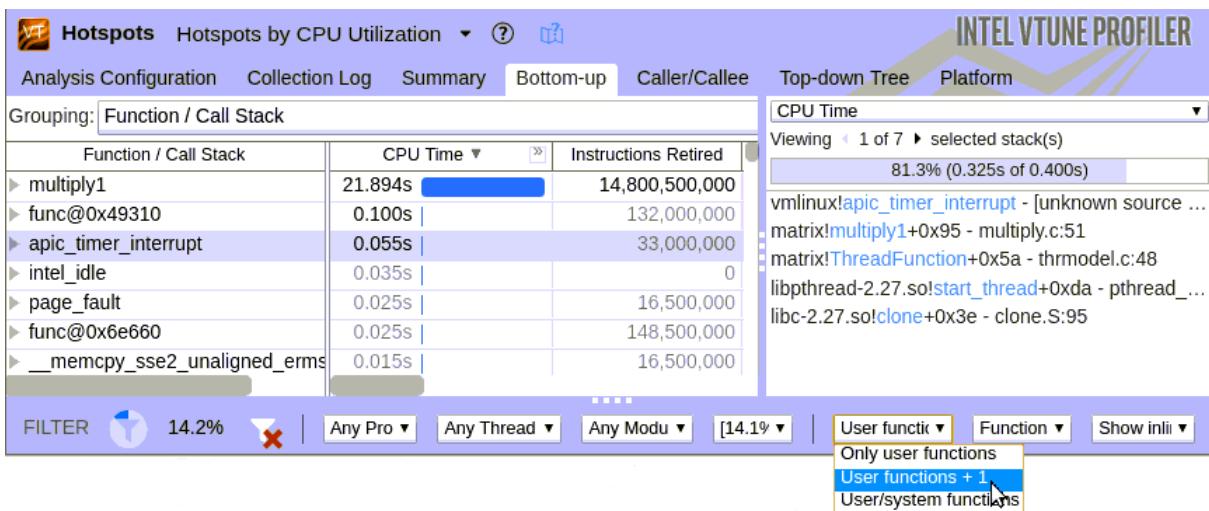
[Pane: Call Stack](#)

[Metrics Distribution Over Call Stacks](#)

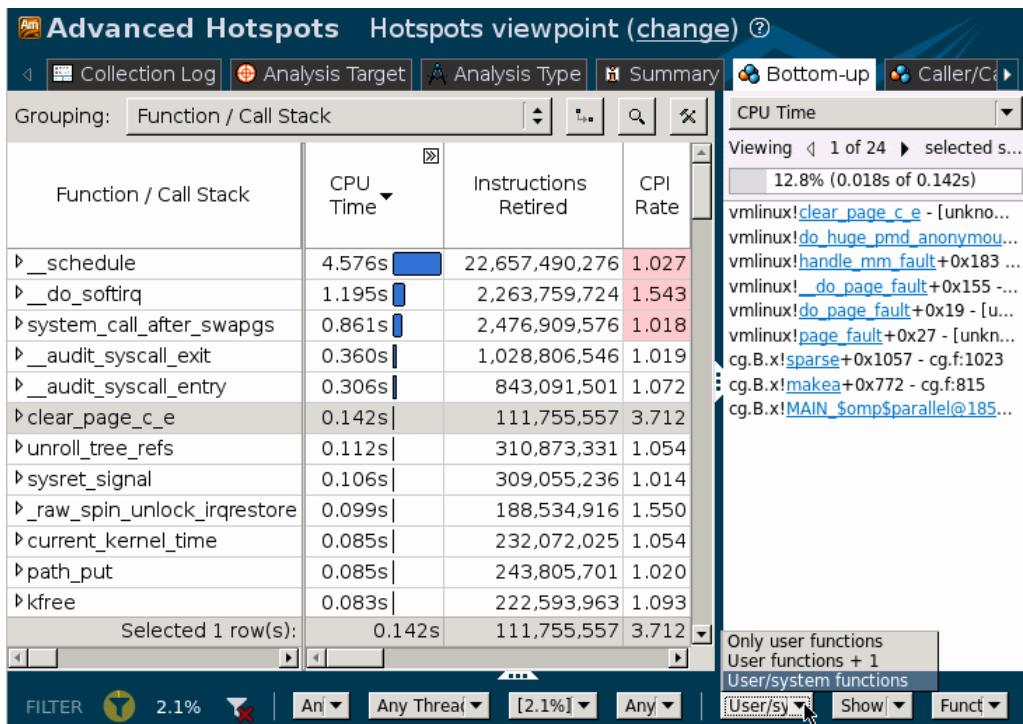
Call Stack Mode

Use the **Call Stack Mode** filter bar menu to choose how to show system functions in the stack.

By default, the Intel® VTune™ Profiler uses the **User function + 1** mode and filters out all system functions except for those directly called from user functions.



To view system functions (for example, kernel stacks) in the user function stacks, select the **User/system functions** call stack mode :



To locate the call of the kernel function in the assembly code, double click the function in the **Call Stack** pane.

NOTE

For more accurate kernel stack analysis on Linux targets, use the `CONFIG_FRAME_POINTER=y` option for kernel configuration.

See Also

[Enable Linux* Kernel Analysis](#)

[Debug Information for Windows* System Libraries](#)

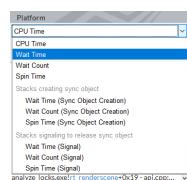
[Toolbar: Filter](#)

[call-stack-mode](#)

[vtune option](#)

[inline-mode vtune option](#)

Metrics Distribution Over Call Stacks



When interpreting the performance analysis results, you can select an object in the grid and select a performance metric in the drop-down menu of the **Call Stack** pane to:

- View stacks leading to the selected object
- Analyze the distribution of the selected performance metric per stacks of the selected object. For example, if the **CPU Time** metric is selected, the contribution

bar shows a share of CPU time spent executing the selected stack relative to the total CPU time spent executing the selected function.

You can also select an object in the **Timeline** pane. In this case, the **Call Stack** pane displays metric data for all objects with the same stacks.

Depending on your analysis configuration, the following metrics are available:

Use This Metric	To Analyze This
CPU Time	Time during which the CPU is actively executing your application on all cores.
Overhead and Spin Time	Combined Overhead and Spin time calculated as CPU Time where call site type is Overhead + CPU Time where call site type is Synchronization.
Wait Time	Distribution of time when one thread is waiting for a signal from another thread. For example, a thread that needs a lock that is currently held by another thread, is waiting for the other thread to release the lock.
Wait Count	Distribution of the number of times the corresponding system wait API was called.
Spin Time	Distribution of Wait Time during which the CPU was busy.
Task Time (Task)	Time spent within a task .
Context Switch Time	Distribution of software thread inactive time due to a context switch, regardless of its reason (Preemption or Synchronization), over different call stacks.
Context Switch Count	Distribution of the amount of context switches, regardless of their reason (Preemption or Synchronization), over different call stacks.
Preemption Context Switch Count	Distribution of the amount of context switches where the operating system task scheduler switched a thread off a processor to run another, higher-priority thread.
Synchronization Context Switch Count	Distribution of the amount of context switches where a thread was switched off because of making an explicit call to thread synchronization API or to I/O API.
Inactive Time	Distribution of time during which a thread remained preempted from execution.
Event metric such as Instructions Retired, Clockticks, LLC Miss , and others	Distribution of a hardware event. Use this metric to identify stacks with the highest contribution of the event count into the total event count collected for the target.
Wait Time (Signal)	Distribution of Wait Time by call stacks of a signaling thread that was releasing a lock where the thread was waiting. Use this metric to identify signaling stacks resulted in long waits to optimize algorithms of the signaling thread.
Wait Count (Signal)	Distribution of Wait Count by call stacks of a signaling thread that was releasing a lock where the thread was waiting. Use this metric to identify signaling stacks resulted in the high number of waits.

Use This Metric	To Analyze This
Spin Time (Signal)	Distribution of Spin Time by call stacks of a signaling thread that was releasing a lock where the thread was waiting. Use this metric to identify signaling stacks resulted in long waits while the CPU is busy.
Wait Time (Sync Object Creation)	Distribution of Wait Time by various object creations. For example, the currently selected row in the grid may contain wait operations on various objects created in different places of the program.
Wait Count (Sync Object Creation)	Distribution of Wait Count by various object creations.
Spin Time (Sync Object Creation)	Distribution of Spin Time by various object creations.
Loads (Memory Allocation)	Distribution of the total number of loads in the stacks allocating memory objects.
Execution (Computing Task (GPU))	Distribution of time spent in the stacks to execute computing tasks. Use this metric to identify most expensive operations for Offload.
Host-to-Device Transfer (Computing Task (GPU))	Distribution of time spent in the stacks to transfer data from host to device. Use this metric to identify most expensive operations for Offload.
Device-to-Host Transfer (Computing Task (GPU))	Distribution of time spent in the stacks to transfer data from device to host. Use this metric to identify most expensive operations for Offload.

NOTE

If a selected stack type is not applicable to a selected program unit, VTune Profiler uses the first applicable stack type from the stack type list instead.

See Also

[Pane: Call Stack](#)

[Group and Filter Data](#)

[Hardware Event-based Sampling Collection](#)

[Reference](#)

Manage Grid Views

Explore the mechanisms provided by the Intel® VTune™ Profiler to manage the data format, sort and filter data in the grid views.

These features are available in all grid views that display collected performance data:

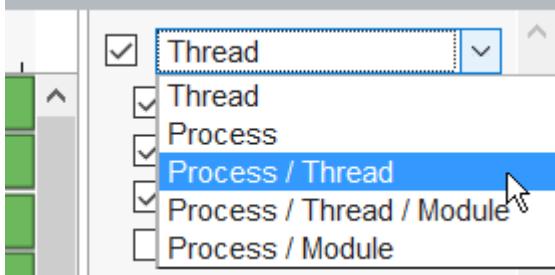
To Do This	Do This
Sort the table by values in a particular column	Click any column header  . You can only sort by one column, however, a previous sorting may be kept for rows with the same values on the current sorting.
Synchronize the selected data	Select a program unit of your interest in a grid or Timeline pane and the VTune Profiler highlights the same unit in other panes/windows.
Re-group the displayed data	Select the required granularity from the Grouping drop-down menu. The available groups depend on the analysis type.
Expand/Collapse data in the column	Click the expand/collapse 
Expand/Collapse row data	Click the expand/collapse 
Change the data representation format	Right-click the data column and select Show Data As > and select from the different data format options . The data format you configure is used in all the windows.
Select rows	Shift-click to select two or more consecutive rows. Ctrl-click to select two or more rows that are not consecutive.
Filter the content of the window	<ul style="list-style-type: none"> Use the drop-down controls in the Filter toolbar to filter data by the contribution of a selected program unit. The percentage contribution depends on the filtering metric selected by clicking the  <p>Filter icon. In the example below, the <code>analyze_locks</code> process contributes 53.4% of the Clockticks event count (default filtering metric for the hardware event-based analysis) to the overall application Clockticks event count, so filtering the collected data by this module causes the viewpoint to show 53.4 % of the overall Clockticks data.</p>  <ul style="list-style-type: none"> Use the Filter In/Out by Selection options of the context menu. VTune Profiler filters in/out the data based on the Total time of the selected item(s). <p>Filtering the data in one window applies the same filter to all the windows of that viewpoint.</p> <p>If you applied filters available on the Filter bar to the data already filtered with the Filter In/Out by Selection context menu options, all filters are combined and applied simultaneously.</p>

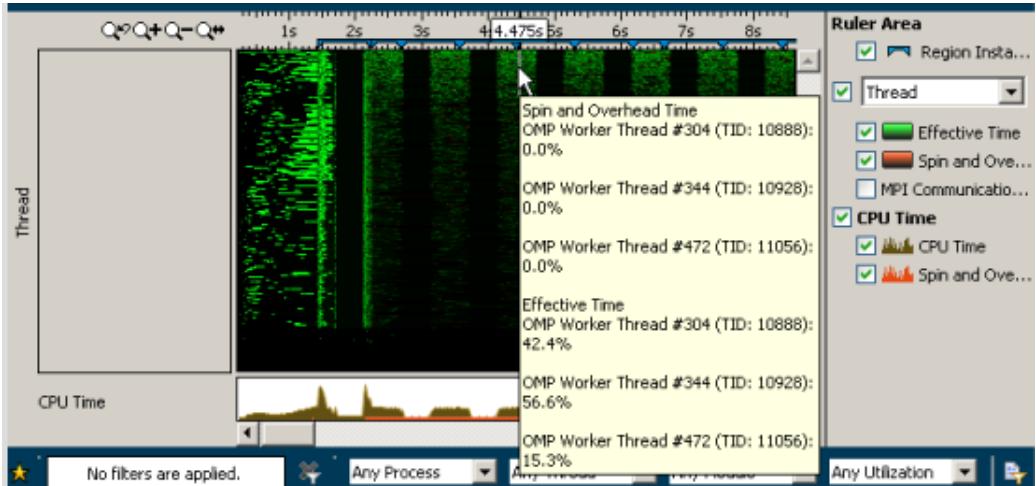
To Do This	Do This
View source/assembly code	Select a program unit you need and double-click. If the source file is not found, the assembly pane is displayed.

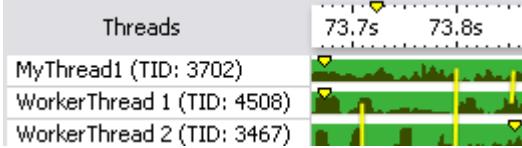
See Also[Window: Bottom-up](#)[Window: Top-down Tree](#)[Toolbar: Filter](#)[Context Menu: Grid](#)**Manage Timeline View**

Explore filtering, sorting and grouping mechanisms available in the Timeline pane in Intel® VTune™ Profiler.

To manage the **Timeline** pane, do the following:

To Do This	Do This
Group the data by program units	<p>Use the Timeline grouping menu to select a grouping level:</p>  <p>A grouping level depends on the analysis type. Selected grouping affects the metrics provided in the Timeline pane. If some of the metrics are not supported for the selected grouping, this data does not show up in the Timeline view and the legend is updated accordingly.</p>
Sort the data	<p>Right-click the list of threads/cores/CPU (depending on the analysis type) and select the required type of sorting from the Sort By content menu option:</p> <ul style="list-style-type: none"> • Row Start Time sorts the rows by the thread creation time. • Row Label sorts the rows alphabetically. • <Metric> sorts the rows by performance metric monitored for the selected viewpoint, for example, CPU Time, Hardware Event Count, and others. • Ascending sorts the program units in the ascending order by one of the categories selected above. • Descending sorts the program units in the descending order by one of the categories selected above.
Re-order the rows	Select the row you need, hold and drag it to the required position. Press SHIFT to select multiple adjacent rows. Press CTRL to select multiple disjointed rows.

To Do This	Do This
Filter data	Select the required program unit(s), right-click and choose from the context menu to filter in or filter out the data in the view by the selected items. To go back to the default view, select the Remove All Filters option.
Zoom in and focus on a particular graph section	<p>1. Drag and drop to select the range of interest. 2. Right-click and select Zoom In on Selection from the context menu.</p> <p>To restore the timeline to the previous state, right-click and select Undo Previous Zoom. To restore the timeline to the entire time interval (for example, after multiple zooming operations), right-click and select Reset Zoom from the context menu or click the  Reset Zoom button on the timeline toolbar.</p>
Zoom in/Zoom out the timeline	Click the  Zoom In/  Zoom Out buttons on the timeline toolbar.
Change the height of the row	<p>Right-click and select the Change Band Height option from the Timeline context menu and select the required mode:</p> <ul style="list-style-type: none"> • Super Tiny mode fits all available rows (corresponding to program units such as processes, threads, and so on) into the timeline area and display metric data in a gradient fill. This mode is especially useful for results with multiple processes/threads since it shows all the data in a compact way ("bird's-eye view") with no scroll bar. It helps observe large numbers that are typical for high-end parallel applications and easily recognize application phases and places of underutilization for further zooming/filtering.  <p>If there are more rows than pixels, then multiple rows can share a pixel, in which case the pixel shows the maximum value. If you hover over a chart object, the tooltip shows all of the rows assigned to a pixel separately. If you resize the window, the timeline view is re-drawn and pixels are re-shared.</p>

To Do This	Do This
	<p>If there is data, the active ranges are colored: the more data associated with a pixel, the more intense color is used for drawing. Otherwise, the band is shown in a black background color.</p> <p>For hierarchical data, the Super Tiny mode shows timeline data for the last level of hierarchy aggregated by the upper levels. For example, for the Process/Thread grouping you see threads data aggregated by process. Hover over a chart element to view the full hierarchy listed in the tooltip.</p> <hr/> <p>NOTE The Super Tiny display mode is available only for the HPC Performance Characterization viewpoint.</p> <ul style="list-style-type: none"> • Normal mode sets the normal row height (about 16-18px). This mode shows charts, time markers, row identification (threads), and transitions. Rows can be reordered.  <ul style="list-style-type: none"> • Rich mode sets the maximum row height (35-50px). This mode shows charts, charts for nested tasks, time markers, row identification (threads), and transitions. Rows can be reordered and their height can be manually adjusted. 
Change the measurement units on the time scale	Right-click, select the Show Time Scale As context menu option, and choose from the following values: <ul style="list-style-type: none"> • Elapsed Time (default) • OS Timestamp • CPU Timestamp
Open the source view	Double-click the required transition/wait. VTune Profiler opens the Source or, if symbol information is not available, Assembly pane and highlights the corresponding waiting/signaling call site.
Synchronize the selection with other panes	Select a thread/module of your interest. VTune Profiler automatically highlights the program units (for example, functions) corresponding to the selected item in the Bottom-up and Top-down Tree panes.

See Also

Pane: Timeline

Source Code Analysis

Group and Filter Data

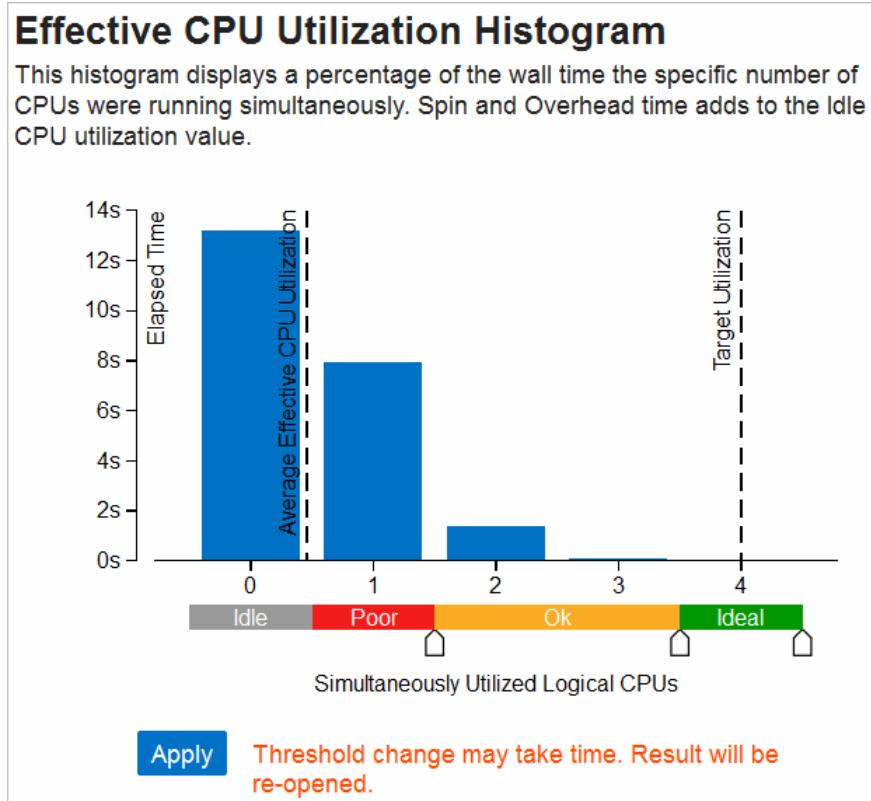
Change Threshold Values

If required, modify default thresholds (for example, for CPU Utilization, concurrency, and frame rate histograms) set up by the Intel® VTune™ Profiler based on your system data.

These thresholds define Poor, OK, Ideal, and Over utilization categories for CPU usage and concurrency metric data and Good, Slow, Fast frame quality categories for frame rate.

To change the default threshold settings:

1. Open the collected result in the **Summary** window.
 2. Drag a slider in a histogram to change the ranges of the required category.
- The **Apply** button shows up at the bottom.
3. Click the **Apply** button to apply your changes.



VTune Profiler applies these threshold changes to the data provided in all viewpoints/windows of the current and subsequent results in this project.

See Also

[Window: Summary](#)

[Set Up Analysis Target](#)

Choose Data Format

Configure the format of presenting the performance data in the grid views.

To configure the data format, right-click the column, select **Show Data As >** from the context menu and choose between available options:

Use This Format	To Do This
Bar	Display a graphical indicator of the amount of CPU time spent on this row item (blue bar) or the processor utilization during CPU or Wait time (composed bar). The longer the bar, the higher the value. Composed bar is available for the Threading analysis only.
Percent	Display the amount of time calculated as the percentage of the cell value to the sum of values in this column for the whole result (or to the non-filtered-out items if a filter is applied). For the nested columns (for example, CPU Time > Idle), the sum of values used in the formula is based on the top-level column values (for example, CPU Time). In the compare mode , the same formula is used for per-result columns (for example: CPU Time:<result 1 name> , CPU Time:<result 2 name>). But for the Difference column (for example: CPU Time:Difference), the percent value is calculated as the cell value / sum of values in this column for the first result (or for non-filtered-out items if a filter is applied).
Percent and Bar	Display both the percent and a bar.
Time	Display the amount of time the processor spent on this row item. The unit size (m, s, ms) is added to each cell value.
Time and Bar	Display both the amount of time and a bar.
Counts	For the Threading Efficiency viewpoint, display the number of times the corresponding system wait API was called. For the event-based sampling results, display event count based on the number of samples collected. Event Counts = Samples x Sample After value.
Counts and Bar	Display both the counts and a bar.
Scientific	Display performance values in the scientific notation. Typically this format is recommended if a value is < 0.001.
Scientific and Bar	Display both scientific notation and a bar.
Double	For some viewpoints available for the hardware event-based sampling analysis types, display the percentage of CPU cycles used by a program unit. For example, 1.533 means that 153% of CPU cycles were used to handle a particular hardware issue during the execution of the selected program unit.
Double and Bar	For some viewpoints available for the hardware event-based sampling analysis types, display the percentage of CPU cycles used by a program unit and corresponding graphical indicator.
Percent of Collection Time	For some metrics (for example, OpenMP* and MPI metrics), display the Time value as percent of Collection Time, which is the wall time from the beginning to the end of collection, excluding Paused Time.

NOTE

The values in the data columns are rounded. For items that are sums of several other items, such as a function with several stacks, the rounded sums may differ slightly from the sum of rounded summands.

For example:

Module / Function	Time (exact)	Time (rounded)
foo.dll	0.2468	0.247
foo()	0.1234	0.123
bar()	0.1234	0.123

The rounded values in the grid do not sum up exactly as $(0.123 + 0.123) \neq 0.247$.

See Also

[CPU Metrics Reference](#)

Group and Filter Data

Analyze the data collected with the Intel® VTune™ Profiler by filtering in areas of interest and grouping the data by specific program units (modules, functions, frame domains, and so on).

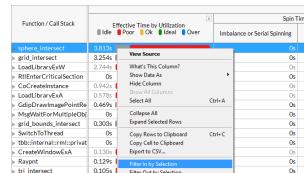
VTune Profiler provides powerful filtering mechanisms that enable you to focus on specific objects or time regions. This helps you focus only on the areas of interest and at the same time speeds up the GUI response when a smaller data set is processed.

Filter by Objects

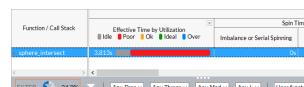
To filter by particular program units (functions, modules, and so on), use any of the following options:

- **Context menu options:** Select objects of interest in the grid, right-click and choose the **Filter In by Selection** context menu option to exclude all objects from the view other than the objects you selected. And conversely, choosing the **Filter Out by Selection** hides the selected data. The filter bar at the bottom is updated to show the percentage of the displayed data by a certain metric.

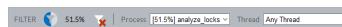
For example, you want to filter in the grid by the most time-consuming function `sphere_intersect`:



When the filter is applied, the filter bar shows that you see only 24.9% of the collected CPU Time data.

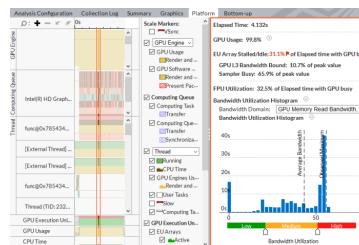


- **Filter toolbar options:** Select a program unit in the filtering drop-down menu (process, module, thread) to filter out your grid and Timeline view for displaying the data for this particular program unit. For example, if you select the `analyze_locks` process introducing 51.5% of the CPU Time, the result data will display statistics for this module only and the filter bar provides an indicator that only 51.5% of the CPU Time data is currently displayed:



Filter by Time Regions

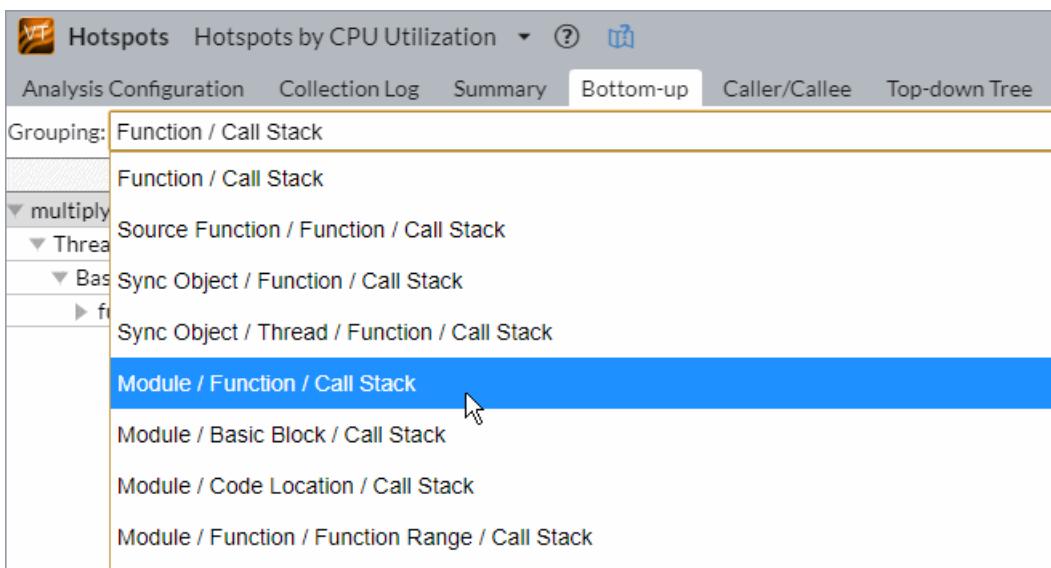
You can narrow down your analysis to particular regions on the timeline. For example, you may select an area of interest on the Timeline pane in the GPU Compute/Media Hotspots viewpoint, right-click and select the **Zoom In by Selection** or **Zoom In and Filter In by Selection** context menu option:



The context summary on the right will be updated for the selected time range and the filter toolbar will show the percentage of the data (per the default metric for this viewpoint) displayed.

Group Data

You can organize a view to focus on the sequence of data you need using the **Grouping** menu. The available groups depend on the analysis type and **viewpoint**:



For example, if you want to view the collected data for the modules you develop, you may select the **Module/Function/Call Stack** granularity, identify the hottest functions in your modules, and then switch to the **Function/Thread/Logical Core/Call Stack** granularity to see which CPUs your hot functions were running on.

VTune Profiler provides a set of pre-configured granularities that could be semantically divided into the following groups:

Groups targeted for analysis	Description
Basic	Identify function hotspots and distinguish problem call stacks.

Groups targeted for analysis	Description
	<p>For most viewpoints, Function level is the default. If application modules have debug information, you can rely on functions shown as hotspots. When debug information is incomplete or missing, you may see a number of <unknown> functions, or samples collected on internal functions of a module might be attributed to adjacent exported functions.</p>
	<p>Examples:</p>
	<p>Function/Call Stack</p>
	<p>Source Function/Function/Call Stack for analyzing all instances of the inline and JITed functions</p>
	<p>Multi-threading analysis Analyze hotspots in multi-threaded applications from the function, OS (Threads) or HW (Packages, Core, Threads) perspectives.</p>
	<p>Examples:</p>
	<p>Function/Thread/Logical Core/Call Stack for detecting anomalies of the function execution on different threads</p>
	<p>Function/Package/Logical Core/Thread/Call Stack for identifying Interconnect/NUMA issues on multi-processor systems</p>
	<p>Physical Core/Logical Core/Function/Call Stack for identifying specific hyper-threading issues</p>
	<p>Physical Core/Thread/Function/Call Stack and Thread/Physical Core/Function for identifying issues caused by thread migration between cores</p>
	<p>Frame analysis Identify slow and fast frames.</p>
	<p>Examples:</p>
	<p>Frame Domain/Frame Duration Type/Function/Call Stack</p>
	<p>Frame Domain/Frame Duration Type/Frame/Function/Call Stack</p>
	<p>OpenMP* analysis Identify hotspots called from OpenMP regions.</p>
	<p>Examples:</p>
	<p>OpenMP Region/OpenMP Barrier-to-Barrier Segment/Function/Call Stack for identifying load imbalance between different segments</p>
	<p>OpenMP Region/OpenMP Region Duration Type/Function/Call Stack for analyzing fast/slow OpenMP region instances</p>
	<p>GPU analysis Analyze the CPU activity while the GPU was either idle or executing some code</p>
	<p>Examples:</p>
	<p>Render and GPGPU Packet Stage / Function / Call Stack</p>
	<p>Render and GPGPU Packet Stage / Thread / Function / Call Stack</p>

Typically, you start your analysis with the **Summary** window where clicking an object of interest opens the grid pre-grouped in the most convenient way for analysis.

If the pre-configured grouping levels do not suit your analysis purposes, you can create your own grouping levels by clicking the



Customize Grouping button and configuring the [Custom Grouping](#) dialog box.

See Also

[Filter and Group Command Line Reports](#)

from command line

[Cookbook: OpenMP* Code Analysis Method](#)

View Data on Inline Functions

Configure the Intel® VTune™ Profiler data view to display the performance data per inline functions for applications in the Release configuration.

Requirements

This option is supported if you compile your code using:

- Linux*:
 - GCC* compiler 4.1 (or higher)
 - Intel® oneAPI DPC++/C++ Compiler. The `-debug:inline-debug-info` option is enabled by default if you compile with optimization level `-O2` and higher, and if debugging is enabled with the `-g` option.
- Windows*:
 - Intel® C++ Compiler Classic, with `/debug:inline-debug-info` option.
 - Intel® oneAPI DPC++/C++ Compiler and Microsoft* Visual C++, with the `/Zo` option. The `/Zo` option is enabled by default when you generate debug information with `/Zi` or `/Z7`.
- Any other compiler that can generate debug information for inline functions in the DWARF format on Linux or Microsoft PDB format on Windows.
- JIT Profiling API is used for inline functions of JIT-compiled code.

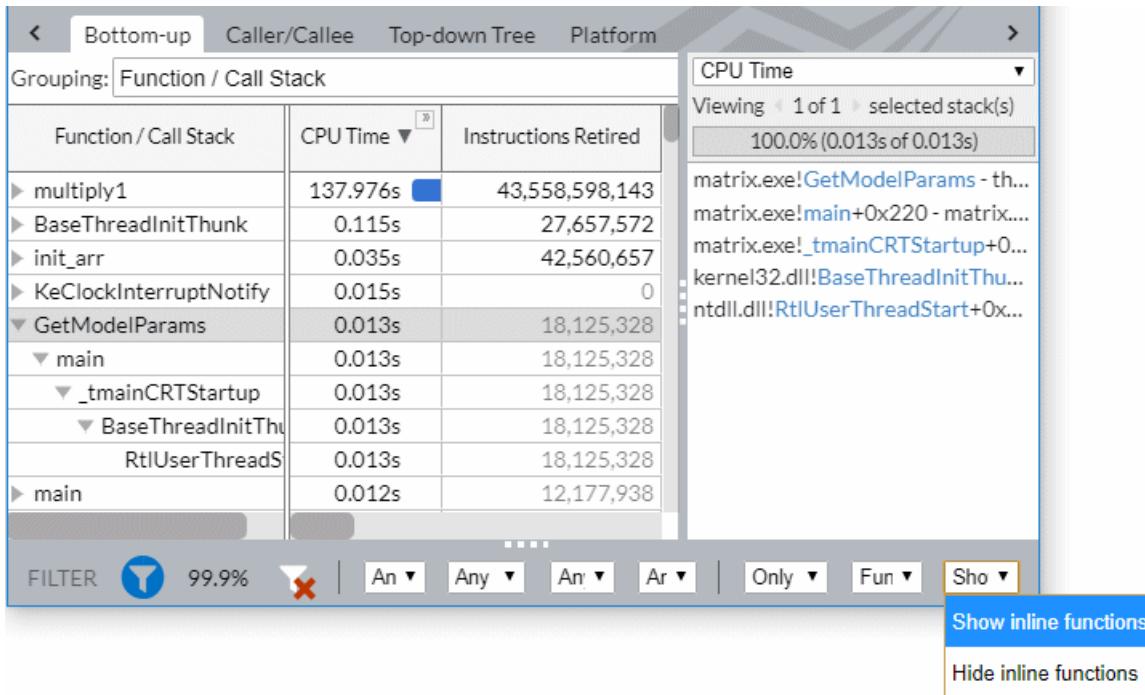
View Inline Functions

To view data on inline functions, in the analysis result window, set the **Inline Mode** filer bar option to **Show inline functions**. VTune Profiler will display inline functions (virtual frames) as regular functions.

To disable displaying inline functions, select **Hide inline functions**.

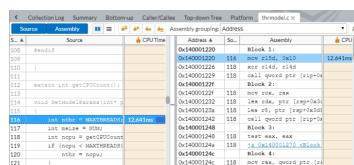
Example 1: Inline Mode for Hotspots Analysis

In this example, you enable the **Show inline functions** option for the Hotspots analysis. This mode shows a full stack for the `GetModelParams` inline function:



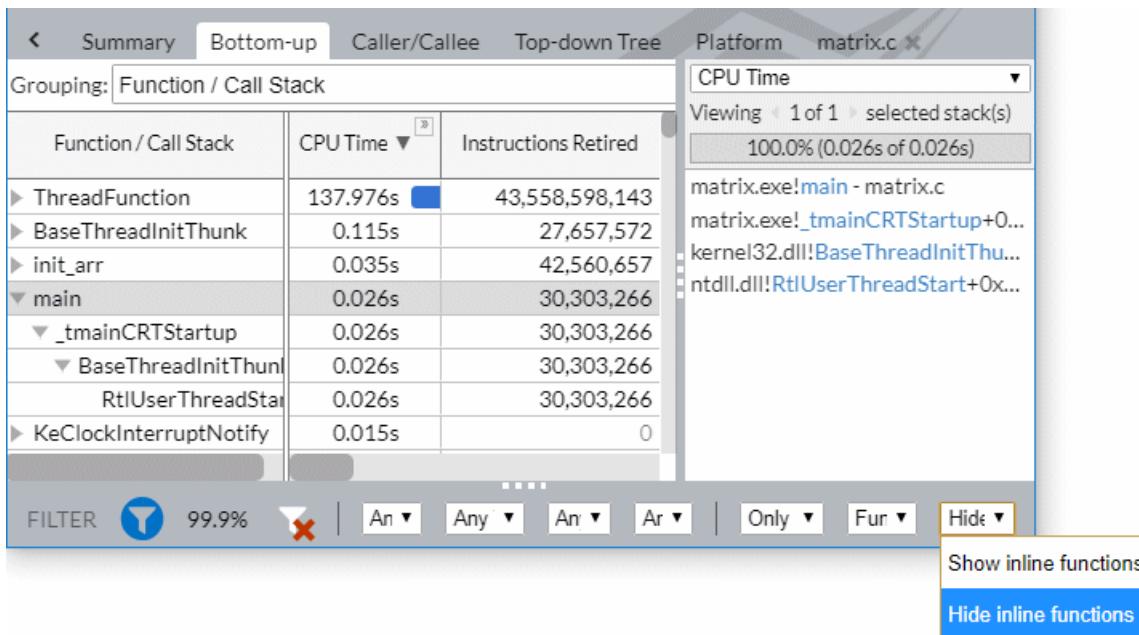
You can select the **Source Function/Function/Call Stack** level in the **Grouping** menu to view all instances of the inline function in one row.

If you double-click the GetModelParams inline function, you can identify the code line that took the most CPU time and analyze the corresponding assembly code:

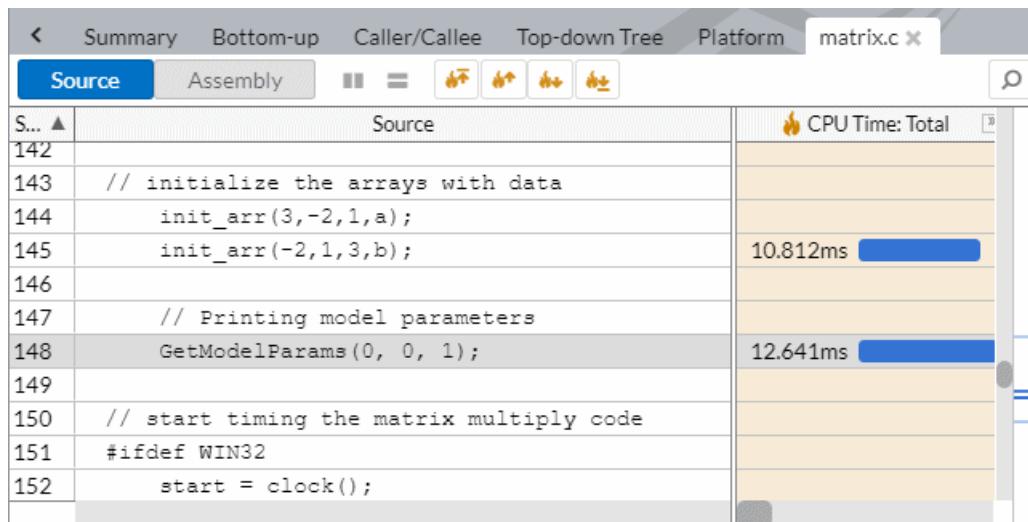


Example 2: Inline Mode for Hotspots analysis Disabled

When you select the **Hide inline functions** option on the filter bar for the same sample, the VTune Profiler does not show the GetModelParams function in the Bottom-up view:

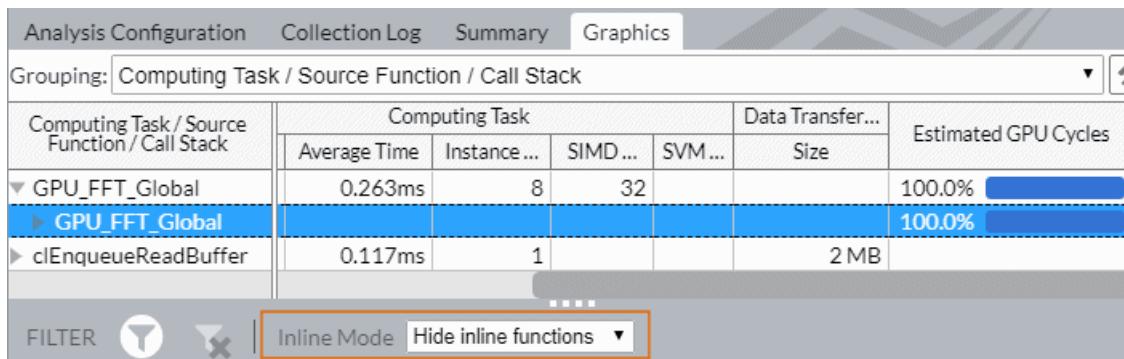


But if you double-click the `main` function entry and explore the source, you can see that all CPU time is attributed to the code line where the `GetModelParams` inline function is called:

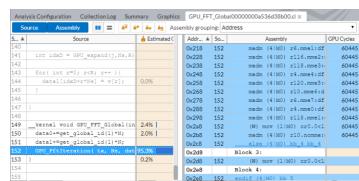


Example 3: Inline Mode for GPU Compute/Media Hotspots

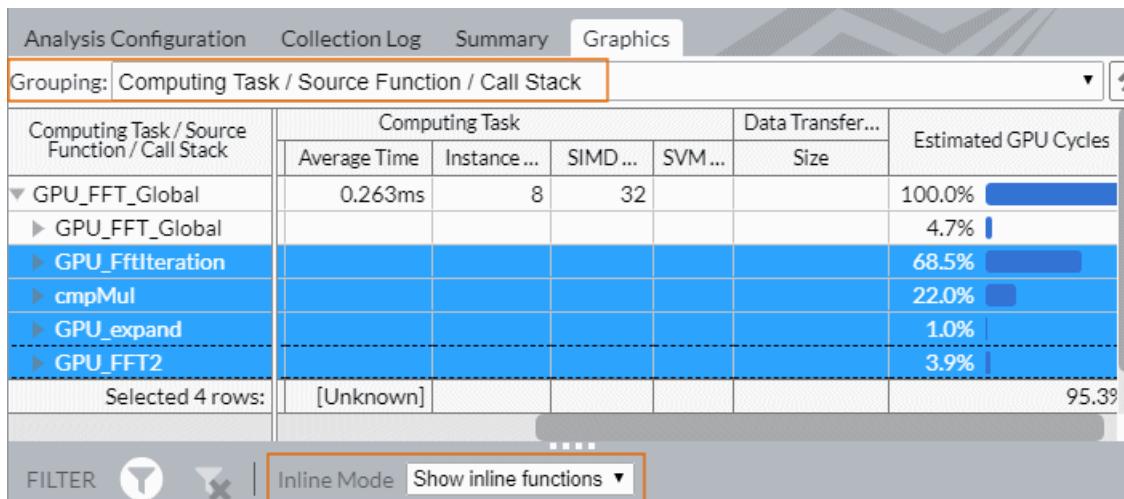
By default, the **Inline Mode** for GPU Compute/Media Hotspots analysis is disabled. In this example, 100% of GPU Cycles are attributed to the `GPU_FFT_Global` function:



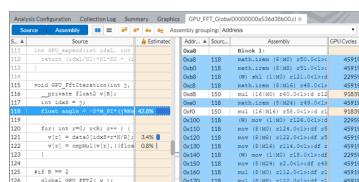
Double-clicking the `GPU_FFT_Global` source function opens the source view positioned on the code line invoking this function with 95.3% of Estimated GPU Cycles attributed to it:



But if you select the **Computing Task/Function/Call Stack** or **Computing Task/Source Function/Call Stack** grouping level and enable the Inline Mode for this view, you see that the `GPU_FFT_Global` function took only 4.7% of the GPU Cycles, while four inline functions took the rest of cycles:



Double-click the hottest `GPU_FftIteration` function to analyze its source and assembly code:



See Also

Toolbar: Filter

[inline-mode vtune option](#)

Analyze Loops

Use the Intel® VTune™ Profiler to view a hierarchy of the loops in your application call tree and identify code sections for optimization.

To view and analyze loops in your application:

1. Create a custom analysis (for example, Loop Analysis) based on hardware event-based collection and select the **Analyze loops**, **Estimate call counts**, and **Estimate trip counts** options.
2. Select the required filtering level from the **Loop Mode** drop-down menu on the Filter toolbar.
 - **Loops only**: Display loops as regular nodes in the tree. Loop name consists of:
 - start address of the loop
 - number of the code line where this loop is created
 - name of the function where this loop is created
 - **Loops and functions**: Display both loops and functions as separate nodes.
 - **Functions only** (default): Display data by function with no loop information.

VTune Profiler updates the grid according to the selected filtering level.

3. Analyze Self and Total metrics in the **Bottom-up** and **Top-down Tree** windows and identify the most time-consuming loops.
4. Double-click a loop of interest to view the source code.

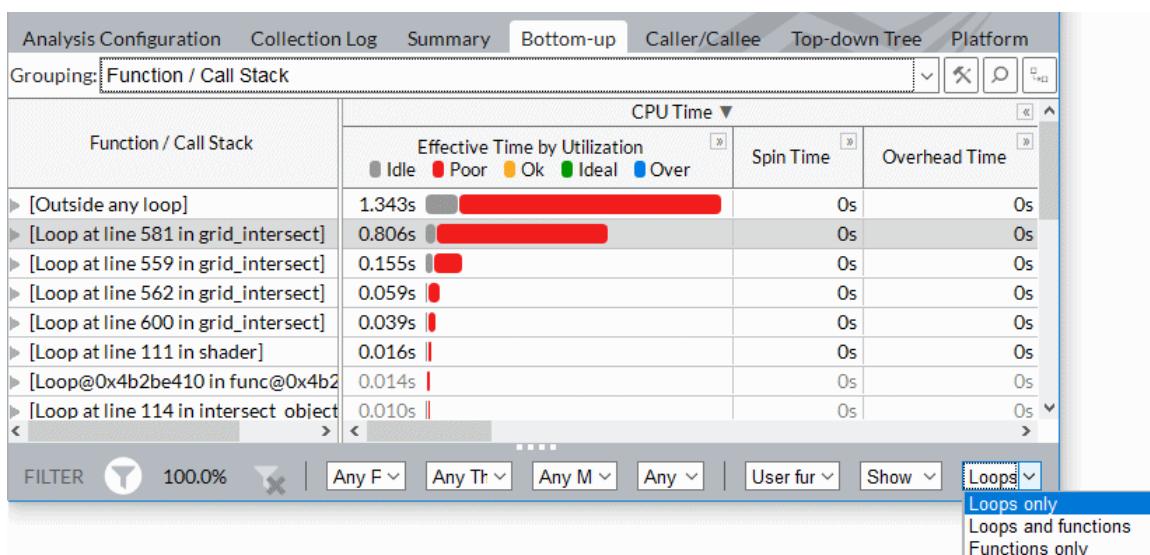
VTune Profiler opens a source file for the function with the selected loop. The code line creating the loop is highlighted.

NOTE

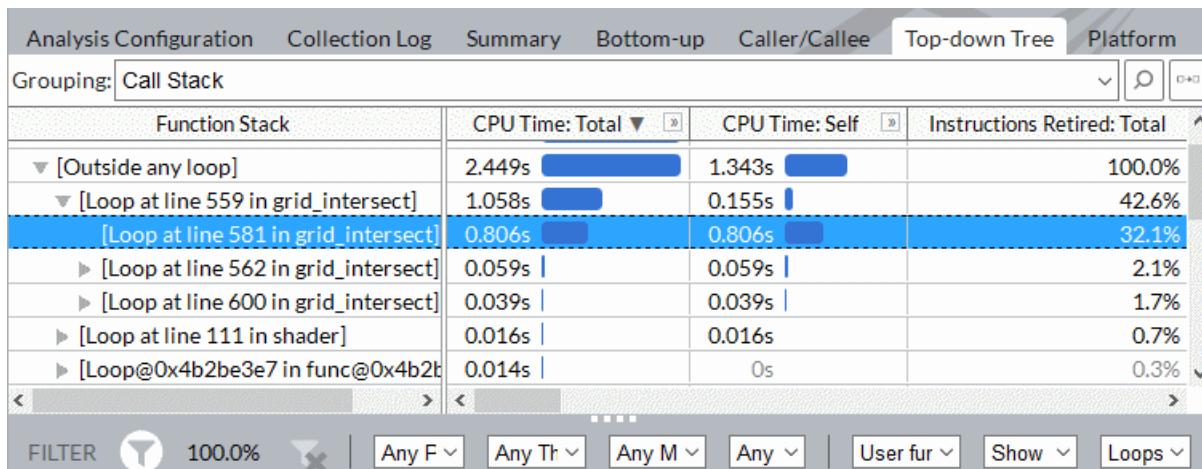
You can see the code line information only if debug information for your function is available.

Examples

To identify the most time-consuming loop, select the **Loops only** mode in the **Bottom-up** window. By default, loops with the highest CPU Time values show up at the top of the grid.



To identify the heaviest top-level loops, switch to the **Top-down Tree** window. The data in the grid is sorted by the Total time metric displaying the hottest top-level loops first:



See Also

[Custom Analysis](#)

[loop-mode](#)

vtune option

[Toolbar: Filter](#)

Stitch Stacks for Intel® oneAPI Threading Building Blocks or OpenMP* Analysis

Use the **Stitch stacks** option to restore a logical call tree for Intel® oneAPI Threading Building Blocks(oneTBB) or OpenMP* applications by catching notifications from the runtime and attach stacks to a point introducing a parallel workload.

Typically the real execution flow in the applications based on Intel® oneAPI Threading Building Blocks(oneTBB) or OpenMP is very different from the code flow. During the [user-mode sampling and tracing](#) analysis of an oneTBB -based application or an OpenMP application using Intel runtime libraries, the Intel® VTune™ Profiler automatically enables the **Stitch stacks** option. To view the OpenMP or oneTBB objects hierarchy, explore the data provided in the **Top-down Tree** pane.

NOTE

- To analyze a logically structured OpenMP call flow, make sure to compile and run your code with the Intel® Compiler 13.1 Update 3 or higher (part of the Intel Composer XE 2013 Update 3).
- Stack stitching is available when you run the application from the VTune Profiler (the **Launch Application** target type). It does not work when attaching to the application (the **Attach to Process** target type).

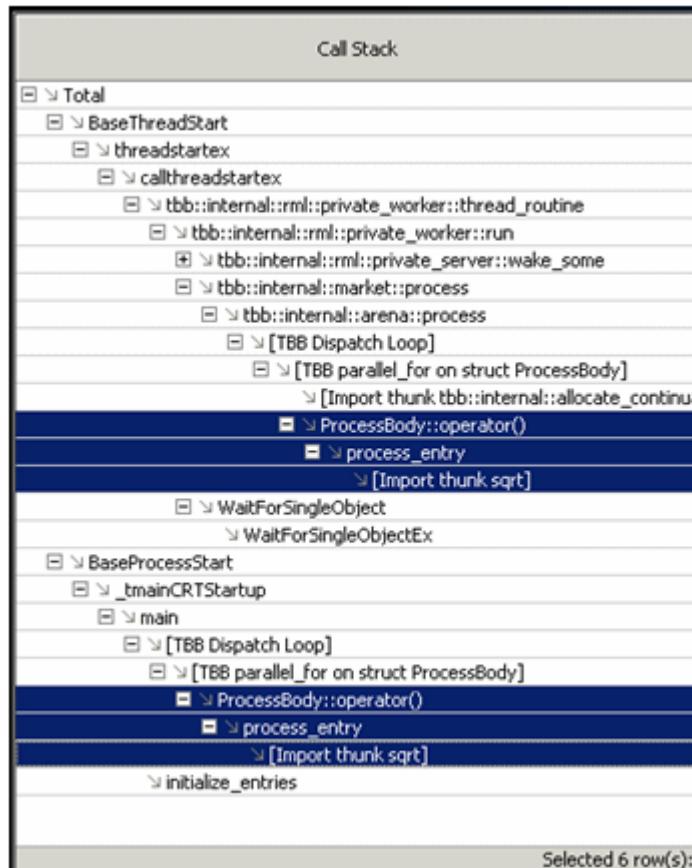
You may want to disable stack stitching, for example, to minimize the collection overhead. To do this for your predefined user-mode sampling and tracing analysis type (for example, Hotspots or Threading), you need to create a new custom analysis configuration and deselect the **Stitch stacks** option in the Custom Analysis configuration. You may use the same modified GUI analysis configuration for command line analysis. For this, just click the **Command Line...** button in the **Configure Analysis** window and [copy the generated command line](#) to run it from the terminal window. Alternatively, you can manually configure the command line for a custom runss analysis using the `knob stack-stitching=false` option like this:

```
> vtune -collect-with runss -knob cpu-samples-mode=stack -knob stack-stitching=false -
knob mrte-type=java,dotnet,python -app-working-dir <path> -- <application>
```

In this case, the **Top-down Tree** pane (or [top-down](#) report) displays separate entries for OpenMP worker threads.

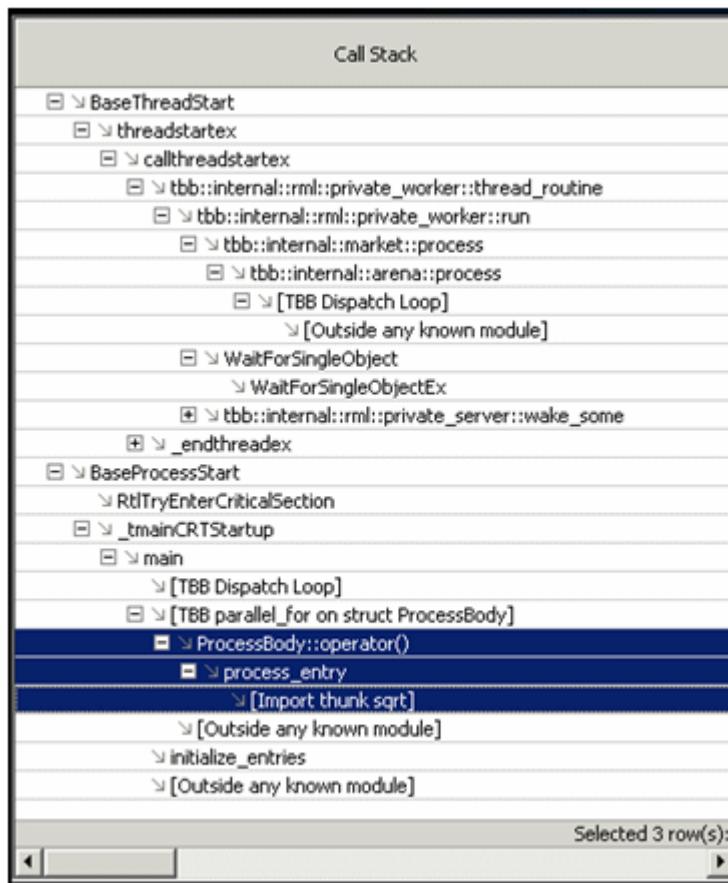
Examples

Call stack in the **Top-down Tree** pane with the **Stitch stacks** option disabled:



Selected 6 row(s):

Call stack in the **Top-down Tree** pane with the **Stitch stacks** option enabled (default behavior):



See Also

[Window: Top-down Tree](#)

[Cookbook: OpenMP* Code Analysis Method](#)

[Intel® Threading Building Blocks Code Analysis](#)

knob

stack-stitching=true

Search for Data

Use the **Find** button to search for data in the **Bottom-up**, **Top-down Tree**, **Source**, or **Assembly** panes.

1. Do one of the following:

- a. Click the



Find button on the toolbar.

- b. Press **CTRL-F** keyword combination.
- c. Right-click and select the **Find** option from the context menu.

The search bar opens.

2. Type in a search string.

All corresponding strings are highlighted in the grid.

3. Press **Enter** or click the Up/Down buttons to navigate between matches. Pressing **SHIFT-Enter** or the Up button goes to the previous match.

See Also

[Context Menu: Grid](#)

[Context Menus: Source/Assembly Window](#)

Manage Result Files

Customize settings for the results you collect with Intel® VTune™ Profiler.

Configure Result Names

You can change the name of an existing result or configure the template of the result name, determining the format of future analysis results.

To change the name of an existing result file in Microsoft Visual Studio * IDE:

1. Right-click the result in the Solution Explorer to open the result context menu.
2. Select **Rename** and edit the name in the Solution Explorer. Make sure to keep the .vtune extension.

You can do the same in the standalone version of the product, using the **Rename Result** context menu option in the [Project Navigator](#).

To change the default result name template:

1. Open the **Result Location** pane as follows:

- Visual Studio IDE: Go to **Tools > Options > Intel VTune Profiler Version > Result Location** pane.
- Standalone interface: click the



menu button and select **Options... > Intel VTune Profiler Version > Result Location** pane.

2. In the **Result name template** text box, edit the text to configure the naming scheme for new analysis results. By default, `r@00{at}` scheme is used, where `{at}` is an analysis type (for example, `hs` for Hotspots).

NOTE

Do not remove the `@@@` part from the template. This is a placeholder enabling multiple runs of the same analysis configuration.

Configure Result Location

For the product version integrated with the Microsoft Visual Studio*, analysis results, by default, are stored in the Visual Studio project default location. For the standalone interface, the analysis result is located in a subdirectory under the project directory. You can view the default location in the **Advanced** section of the **WHAT** configuration pane.

To change the result location:

1. Click the



Configure Analysis toolbar button.

- The **Configure Analysis** window opens.
2. Choose the required target system and target type in the **WHERE** and **WHAT** panes.
 3. Expand the **Advanced** options section and edit the **Store result in (and create link file to) another directory** field to specify a directory of your choice.

All subsequent analysis results will be located under the folder you defined in this tab.

VTune Profiler Filenames and Locations

Intel® VTune™ Profiler generates the following file types:

- [Analysis result files](#)
- [Analysis configuration files](#)
- [Project file](#)

Installation Information

Whether you downloaded Intel® VTune™ Profiler as a standalone component or with the Intel® oneAPI Base Toolkit, the default path for your <install-dir> is:

Operating System	Path to <install-dir>
Windows* OS	<ul style="list-style-type: none"> • C:\Program Files (x86)\Intel\oneAPI\ • C:\Program Files\Intel\oneAPI\ <p>(in certain systems)</p>
Linux* OS	<ul style="list-style-type: none"> • /opt/intel/oneapi/ for root users • \$HOME/intel/oneapi/ for non-root users
macOS*	/opt/intel/oneapi/

For OS-specific installation instructions, refer to the [VTune Profiler Installation Guide](#).

Analysis Result Files

File Type	Default Location
Result (*.vtune) produced with preset analysis type	<p>The location of the result files is controlled by the user. The default location for VTune Profiler is:</p> <ul style="list-style-type: none"> • On Linux: /root/intel/vtune/projects/[project directory]/r@@@{at} • On Windows: <ul style="list-style-type: none"> • VTune Profiler Results\[project name]\r@@@{at} directory in the solution directory (Visual Studio* IDE) • %USERPROFILE%\My Documents\Profiler XE\Projects\[project directory]\r@@@{at} directory (Standalone VTune Profiler GUI)
Result (*.vtune) produced with a custom analysis type	<p>The location of the result files is controlled by the user. The default location for the VTune Profiler is:</p> <ul style="list-style-type: none"> • On Linux: /root/intel/vtune/projects/[project directory]/r@@@ • On Windows: <ul style="list-style-type: none"> • VTune Profiler Results\[project name]\r@@@ directory in the solution directory (Visual Studio* IDE)

File Type	Default Location
	<ul style="list-style-type: none"> • %USERPROFILE%\My Documents\Profiler XE\Projects\[project directory]\r@@@ directory (Standalone VTune Profiler GUI)

To open a result from the standalone GUI, select **Open > Result...** from the menu and browse to the result file. To open a result from Visual Studio, double-click the node in the Solution Explorer.

Analysis Configuration Files

File Type	File Location
Preset analysis type (for example, hotspots.cfg)	config/analysis_type in the product installation directory.
Custom analysis type (for example, Hardware Event-based Sampling Analysis @@@.cfg, where @@@ is the next available number)	Windows: %APPDATA%\intel\Profiler XE\analysis_type Linux: /root/.intel/vtune/analysis_type

Project File

File Type	File Location
Project (for example, *.vtuneproj)	The filename is controlled by the system. However, the file location is controlled by the user. The default location is: <ul style="list-style-type: none"> • On Linux: /root/intel/vtune/projects/[project directory] • On Windows: <ul style="list-style-type: none"> • VTune Profiler Results\[project name] directory in the solution directory (Visual Studio* IDE) • Profiler XE\Projects\[project directory] directory (Standalone Intel VTune Profiler GUI)

Examples

Run the Hotspots analysis and then run the Threading analysis. If you use the default naming convention and result location, the VTune Profiler names and saves the results in the following manner:

- Standalone GUI Linux:
 - /root/intel/vtune/projects/r000hs/r000hs.vtune
 - /root/intel/vtune/projects/r001tr/r001tr.vtune
- Standalone GUI Windows:
 - %USERPROFILE%\My Documents\Profiler XE\Projects\[project directory]\r000hs\r000hs.vtune
 - %USERPROFILE%\My Documents\Profiler XE\Projects\[project directory]\r001tr\r001tr.vtune
- Visual Studio IDE:
 - VTune Profiler Results\[project name]\r000hs\r000hs.vtune
 - VTune Profiler Results\[project name]\r001tr\r001tr.vtune

where

- hs is the Hotspots analysis type
- tr is the Threading analysis type

See Also

Pane: Options - Result Location

Import Results and Traces into VTune Profiler GUI

If you collect performance data either remotely with the Intel® VTune™ Profiler command line interface or with standalone collectors (such as SEP collector, Intel SoC Watch collector, or Linux Perf* collector), import this data (result or trace) to the VTune Profiler project to analyze it in the graphical interface.*

To get ready for the import:

1. Create a VTune Profiler project for the data to be imported.
2. In the **Configure Analysis** window, click the



Search Sources/Binaries button at the bottom to specify [search directories](#) for the data to be imported. When you open the Source/Assembly view for the collected data, the VTune Profiler automatically applies binary/source search paths for proper symbol resolution.

NOTE

Make sure the search directories are accessible to the VTune Profiler. For example, if you are to import the data collected remotely, you need to copy the sources and binaries to the host system where the VTune Profiler is installed or make them available via a shared drive.

3. Select the **Import** option using any of the following options:

- From Microsoft Visual Studio* IDE: Open a project where you want to locate the imported result and go to **Tools > Intel® VTune™ Profiler version > Import Result...**.
- From standalone VTune Profiler interface: Open a project where you want to locate the imported result, click the



menu button and select **Import Result...**, or click the



Import Result button on the toolbar.

The **Import** window opens.

4. Choose between two options:

- [import an *.vtune result](#) (a marker file with associated result directories) collected remotely with the VTune Profiler command line interface;
- [import a raw trace file](#) collected by standalone collector tools.

Import Results

You can perform multiple collections on a remote system (with or without result finalization) with a full-fledged VTune Profiler command line interface, copy the result directories to the host, and import the result(s) into a VTune Profiler project.

To import result directories into a VTune Profiler project:

1. In the **Import** window, select the **Import a result into the current project** option.
2. Click the

browse button to navigate to the required directory.

3. If required, click the **Search Sources/Binaries** button on the right to view/modify the search directories.
4. Click the **Import** button on the right.

VTune Profiler copies the result directory to the current project folder and result name appears in the Project Navigator as a node of the current project.

NOTE

If you do not need to copy a result, select the **Import via a link instead of a result copy** option. VTune Profiler will import the result via this link.

Import Raw Trace Data

You can also import performance trace files collected using:

- SEP Collector
- Soc Watch Collector
- Perf Collector

View the collected data in the VTune Profiler GUI.

You can import these data formats:

- *.tb6/*.tb7 (sampling raw data files collected with the low-level SEP collector)
- *.perf (Linux* Perf data files)
- *.csv ([external data collection](#) files in the predefined format)
- *.pwr (processed Intel SoC Watch files with [energy analysis](#) data)
- *.json (FPGA performance data collected with the [Profiler Runtime Wrapper](#))

NOTE

For FPGA data collected with the Profiler Runtime Wrapper, you must import a folder with the `profile.json` file. Use the **Import multiple trace files from a directory** option in the **Import** window. See the section below on importing trace files into a VTune Profiler project.

Prerequisites for importing a *.perf file with event-based sampling data:

Run the Perf collection with the predefined command line options:

- For application analysis:

```
perf record -o <trace_file_name>.perf --call-graph dwarf -e cpu-cycles,instructions
<application_to_launch>
```

- For process analysis:

```
perf record -o <trace_file_name>.perf --call-graph dwarf -e cpu-cycles,instructions
<application_to_launch> -p <PID> sleep 15
```

where the `-e` option is used to specify a list of events to collect as `-e <list of events>`; `--call-graph` option (optional) configures samples to be collected together with the thread call stack at the moment a sample is taken. See Linux Perf documentation on possible call stack collection options (for example, `dwarf`) and its availability in different OS kernel versions.

NOTE

The Linux kernel exposes Perf API to the Perf tool starting from version 2.6.31. Any attempts to run the Perf tool on kernels prior to this version lead to undefined results or even crashes. See Linux Perf documentation for more details.

To import trace files into a VTune Profiler project:

1. In the **Import** window, select the **Import raw trace data** option.
2. Click the

browse button to navigate to the required file.

To import multiple files, select the **Import multiple trace files from a directory** option.

NOTE

For FPGA data collected with the Profiler Runtime Wrapper, you need to use this option to import a *folder* with the `profile.json` file. See the FPGA Optimization Guide for Intel® oneAPI Toolkits for details on generating the profiling data.

3. If required, click the **Search Sources/Binaries** button on the right to view/modify the search directories.
4. Click the **Import** button on the right.

VTune Profiler copies the trace file (or a directory with multiple traces) to the project directory, creates an `*.vtune` result directory, finalizes the trace(s) in the directory, and imports it to the current project. When you open the result in the VTune Profiler, it uses all applicable viewpoints to represent the data.

NOTE

- To reduce the size of the imported data, consider removing the copy of the trace file in the project directory using the **Remove raw collector data after resolving the result** option available from **Options... > Intel VTune Profilerversion > General** tab in the standalone interface menu



or from **Tools > Options... > Intel VTune Profilerversion > General** tab in Microsoft Visual Studio* IDE. This option makes the result smaller but prevents future re-finalization.

- You can run a custom data collection (with a third-party collector or your own collection utility) in parallel with the VTune Profiler analysis run, convert the collected data to a `*.csv` file and import this file to the VTune Profiler project using the **Import from CSV** GUI option or `-import` CLI option. You may also choose to use the **Custom collector** option of the VTune Profiler to run your custom collection directly from the VTune Profiler.
-

See Also

[import](#)
vtune option

[Dialog Box: Binary/Symbol Search](#)

[Dialog Box: Source Search](#)

Compare Results

Compare your analysis results before and after optimization and identify a performance gain.

Use this feature on a regular basis for regression testing to quickly see where each version of your target has better performance.

You can compare any results that have common performance metrics. Intel® VTune™ Profiler provides comparison data for these common metrics only.

To compare two analysis results:

- Click the



Compare Results button from the VTune Profiler toolbar.

The **Compare Results** window opens.

Option	Description
Result 1 / Result 2 drop-down menu	Specify the results you want to compare. Choose the result of the current project from the drop-down menu, or click the Browse button to choose a result from a different project.
Swap Results button	Click this button to change the order of the result files you want to compare. Result 1 always serves as the basis for comparison.
Compare button	Click this button to view the difference between the specified result files. This button is only active if the selected results can be compared. Otherwise, an error message is displayed.

- Specify two results that you want to compare and click the **Compare** button.

A new result tab opens providing difference between the two results per performance metric.

The tab name combines the identifiers of two results. For example, the comparison of the Microarchitecture Exploration analysis results **r001ue** and **r005ue** appears as **r001ue-r005ue**. The data views in the comparison mode provide calculation of the difference between the two results in the order you originally defined in the **Compare Results** window and as specified in the tab title.

You can compare performance statistics in the following views:

Use this view:	To do this:
Summary window	Analyze the difference in the overall application performance between two results and the system/platform difference, if any. Start exploring the changes from the Summary window and then move to the Bottom-up analysis to identify the changes per program unit.
Bottom-up window	Analyze the data columns of the two results and a new column with the difference between these results for a function and its callers.
Event Count window	Compare results and identify the difference in event count and performance per hardware event-based metrics collected during event-based sampling analysis.
Top-Down Tree window	Explore the performance difference between two collection runs for a function and its callees.
Caller/Callee window	Get a holistic picture of the performance changes before and after optimization by comparing data for a function, its callers and callees.
Source/Assembly window	Understand how differently input values, command line parameters, or compilation options affect the performance when you are optimizing your target. Double-click a program unit of your interest and compare the performance data for each line of the source/assembly code.

See Also

[Difference Report](#)
from command line

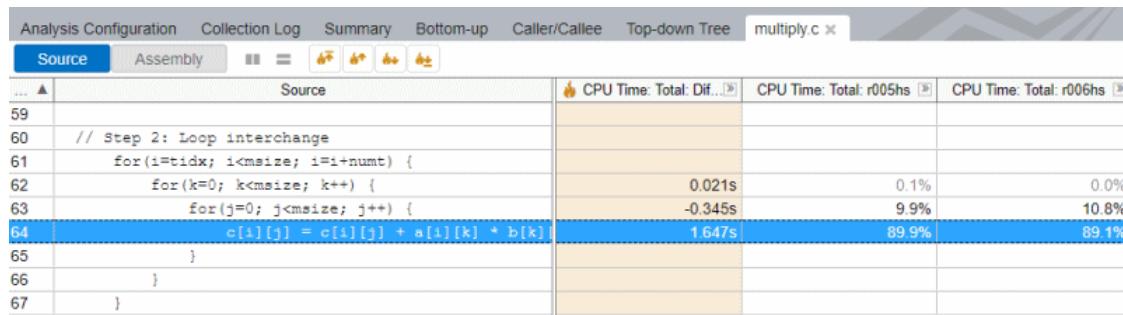
Compare Source Code

To view the performance difference for the source/assembly code, open the results in the Bottom-up compare mode and double-click the function you are interested in to view the performance values for each line of its source/assembly code of both results and the difference between them.

If	Then
The source and binary files are not modified and the debug information is available	Compare performance for each source/assembly code line.
The source and binary files are not modified but the binaries are compiled without the debug information	Compare performance for each assembly instruction.
The source files are not modified but the binary files are re-compiled with different options	Compare performance for each source code line.
NOTE When comparing the source code for binary files with different checksum, only the Source pane is available.	
The source and binary files are modified	Intel® VTune™ Profiler cannot compare performance for source/assembly code and displays an error message.

Example

When you click the hotspot function in the **Bottom-up** window, the VTune Profiler opens the **Source** pane that displays the CPU time data per each result and the difference between the results.



You see that the execution of the hottest line 64 took less CPU time in result r006hs.

See Also

[Window: Compare Results](#)

[Compare Results](#)

[Source Code Analysis](#)

View Comparison Data

Intel® VTune™ Profiler compares analysis results and displays difference in a separate result tab **<result1>-<result2>** in the following windows:

- **Summary** window provides top-level difference for the analysis run.

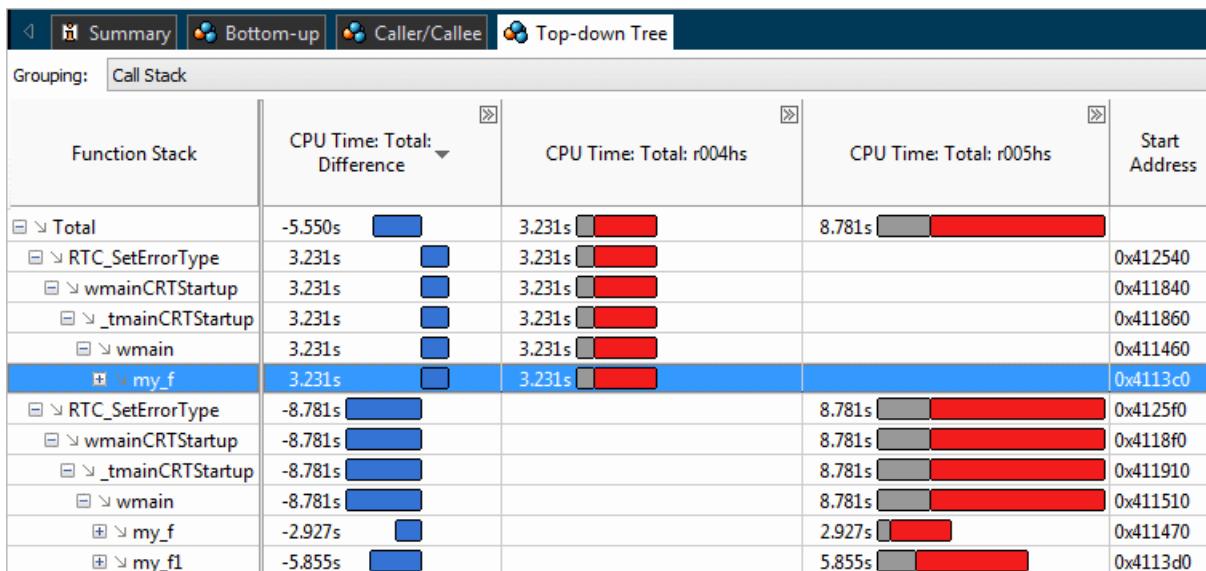
- **Bottom-up** window displays difference for functions and their callers per metric.
- **Top-down Tree** window displays difference for functions and their callees per metric.
- **Caller/Callee** window displays difference for a selected function, their callers and callees per metric.

Comparing Recompiled Binary Files

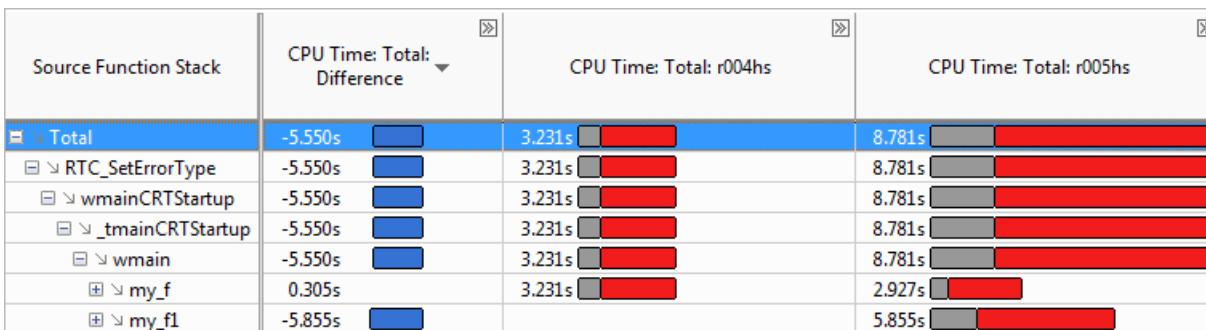
By default, the VTune Profiler displays compared functions grouped by the **Call Stack** granularity, which is based on function instances. But you may want to switch to the **Source Function Stack** grouping to get more accurate comparison results in the following cases:

- You slightly changed the source and recompiled the code.
- You changed compilation options and recompiled the code.
- You are comparing results compiled and collected for different Intel microarchitectures

For example, your binary with a `my_f` function was modified with adding a new function `my_f1` and new calls of this function. As a result, `my_f` address has changed. If you compare the results before and after the modification using the default **Call Stack** grouping, the VTune Profiler treats the same functions with different addresses as separate instances and does not compare them:



When the data is aggregated by **Source Function Stack**, the VTune Profiler ignores start addresses and compares functions by source file objects:



Bar Data Representation

If you chose the **Bar** format to display the performance data in the **Bottom-up** or **Top-down Tree** window, the VTune Profiler calculates the bar size as follows:

Result Data Column	Difference Column	
cell_data_value/ absolute_max_value_in_result_column	cell_data_value/ max(absolute_max_value_in_1st_result_column ,	absolute_max_value_in_corresponding_2nd_resu lt_column)

Example: Calculation of the Bar Size

The table below provides an example on how the VTune Profiler calculates the bar size in the compare mode based on the absolute max CPU time value and performance data per column:

	CPU Time:r001	CPU Time:r002	CPU Time:Difference
Absolute max value (calculated by the VTune Profiler internally but not exposed in the grid)	10s	20s	20s
Performance data	1s	3s	2s
Bar size	1s/ max(10s,2 0s)	3s/ max(10s,2 0s)	2s/max(10s,20s)

See Also

[Bottom-up Comparison](#)

[Comparison Summary](#)

[Top-down Tree Comparison](#)

[Difference Report](#)

from command line

[Comparison Summary](#)

When you click the **Compare Results**



button and select two results to compare, the **Summary** window shows the difference between these results.

NOTE

- You can compare any results that have common performance metrics. Intel® VTune™ Profiler will provide comparison data for these common metrics only.
- Make sure to close the results before comparing them.

Data provided in the **Summary** window vary depending on the viewpoint.

Difference in the Application Performance per Metrics

In the compare mode, the first metrics section displays the difference in the [performance metrics](#) values between the results you specified. In the example below, the VTune Profiler displays the Hotspots results difference as `result1_value - result2_value` (shown in the result tab title).

Elapsed Time [?] : 30.142s - 25.267s = 4.875s	
Total Thread Count:	Not changed, 9
Paused Time [?] :	Not changed, 0s
CPU Time [?] :	19.645s - 21.571s = -1.926s
Effective Time [?] :	17.167s - 19.661s = -2.493s
Spin Time [?] :	1.873s - 1.712s = 0.160s
Overhead Time [?] :	0.605s - 0.198s = 0.407s

You see that the code changes in the second result have slightly decreased the Elapsed time of the application in comparison with the baseline (result1), though the CPU Time has increased from 19.645 seconds to 21.571 seconds.

Clicking a metric in this section opens the Bottom-up view sorted by this metric in the Difference column.

Difference in the Performance of Hotspot Objects

In the compare mode, the **Top Hotspots** section displays the difference in performance values per object (object type depends on the viewpoint) between the results you specified. In the example below, the VTune Profiler displays the CPU Time difference for the most time consuming functions as **r000hs** value - **r004hs** value (see the result tab title).

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time [?]
grid_intersect	analyze_locks.exe	4.575s - 5.222s = -0.647s
sphere_intersect	analyze_locks.exe	3.590s - 4.809s = -1.219s
func@0x69e19df0	user32.dll	2.491s - 2.333s = 0.158s
PeekMessageA	user32.dll	1.510s - 1.537s = -0.027s
GdipDrawImagePointRectI	gdiplus.dll	1.245s - 1.607s = -0.361s
[Others]		6.235s - 6.063s = 0.172s

*NA is applied to non-summable metrics.

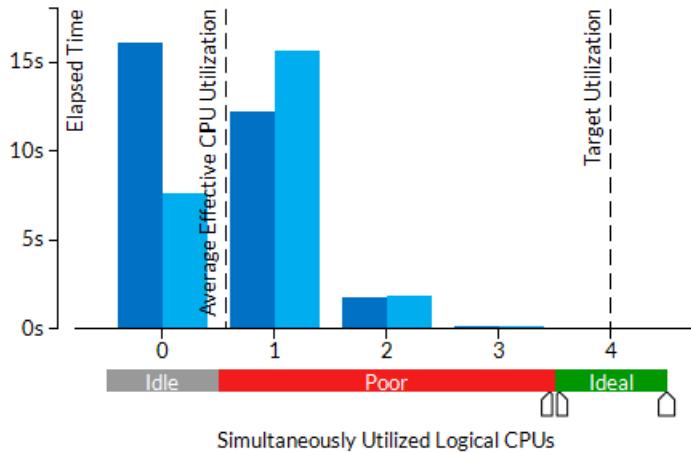
For this example, the second result introduced slight degradation in CPU Time for the first and second functions.

Performance Difference in Histograms

Depending on the viewpoint, the **Summary** window provides the histograms that show how certain metrics, like the [Thread Concurrency](#), [CPU Utilization](#), or Frame rate, have changed for the specified results. Bars for both results show up side by side. In the example below, the dark-blue bars correspond to the first result, and the light-blue bars correspond to the second result. Hover over a bar to see the tooltip with the detailed information:

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



The chart shows that the Elapsed time spent within the Poor processor utilization level has slightly increased with the second result. This means that the changes made for the second run have not optimized the utilization of the processor cores but introduced a slight optimization reducing the total Elapsed time.

Difference in Collection and Platform Info

The Collection and Platform section shows whether the result size and platform data has changed.

NOTE

You can click the



Copy to Clipboard button next to any summary section and copy its content to the clipboard.

See Also

[Compare Results](#)

[Bottom-up Comparison](#)

[Top-down Tree Comparison](#)

Bottom-up Comparison

To view the difference before and after optimization for a function and its callers, click the **Bottom-up** sub-tab for the comparison result you created using the **Compare Results** window.

In the compare mode, the **Bottom-up** window shows the data columns of the two results and a new column showing the difference between the two results for each program unit. The difference is calculated as <Result 1 Value> - <Result 2 Value>.

Example: Comparison for Hotspots Analysis Results

The **Bottom-up** window displays the data columns for each result and a **Difference** column that calculates the difference between the two results. By default, the **Difference** column is collapsed and displays the total difference data per CPU time. You may click the double-arrow icon to expand the column and see comparison data per utilization level.

Analysis Configuration		Collection Log		Summary		Bottom-up		Caller/Callee		Top-down Tree		Platform					
Function / Call Stack		CPU Time: Difference ▾						CPU Time: r000hs		CPU Time: r004hs							
		Effective Time by Utilization						Idle	Poor	Ok	Ideal	Over	Spin Time	Overhead Time			
► libm_sse2_pow_precise		0.007s	-0.042s	0s	0s	0s		0s		0s	0.063s	0.098s					
► pos2grid		-0.030s	-0.023s	0s	0s	0s		0s		0s	0.029s	0.081s					
► NtUserMsgWaitForMultiple		0s	-0.068s	0s	0s	0s		0s		0s	0.011s	0.078s					
► CoCreateInstance		0.688s	-1.016s	0s	0s	0s		0.006s		0s	0.694s	1.016s					
► GdipDrawImagePointRectI		-0.076s	-0.285s	0s	0s	0s		0s		0s	1.245s	1.607s					
► grid_intersect		-0.138s	-0.509s	0s	0s	0s		0s		0s	4.575s	5.222s					
► func@0x1003d260		0s	-0.828s	0s	0s	0s		0s		0s		0.828s					
► sphere_intersect		-0.221s	-0.999s	0s	0s	0s		0s		0s	3.590s	4.809s					

CPU time specific difference is calculated as <Result 1 CPU time> - <Result 2 CPU time>, which is **r000hs-r004hs** (see the tab title). Expand the first two columns to see the data used for the calculation.

Analysis Configuration		Collection Log		Summary		Bottom-up		Caller/Callee		Top-down Tree		Platform							
Function / Call Stack		CPU Time: Difference ▾						CPU Time: r000hs				CPU Time: r004hs							
		Effective Time by Utilization						Idle	Poor	Ok	Ideal	Over	Spin Time	Overhead Time					
► grid_intersect		-0.138s	-0.509s	0s	0s	0s	0s	0.614s	3.961s	0s	0s	0s	0s	0s					
► func@0x1003d260		0s	-0.828s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0.828s	0s					
► sphere_intersect		-0.221s	-0.999s	0s	0s	0s	0s	0.492s	3.098s	0s	0s	0s	0s	0s					

For the `grid_intersect` function in this example, the difference is $3.961s - 4.470s = -0.138s$ of Poor CPU utilization time, which means that serial CPU time has insignificantly increased after code modification (Result 2).

See Also

[Compare Results](#)

[Window: Bottom-up](#)

[Comparison Summary](#)

[Choose Data Format](#)

Top-down Tree Comparison

To understand how your application call tree has changed after your optimization and see the difference in performance metrics per function and its callees, click the **Top-down Tree** sub-tab and explore the **Top-down Tree** window.

In the compare mode, the **Top-down Tree** window shows the data columns of the two results and a new column showing the difference between the two results for each program unit. The difference is calculated as <Result 1 Value> - <Result 2 Value>.

The **Top-down Tree** window in the compare mode supports two types of grouping levels:

- **Function Stack** granularity groups the data by function instances. Use the **Start Address** column to identify different instances of the same source function or same loop.

- **Source Function Stack** granularity groups the data by source functions. In this mode, all instances of the same source function are aggregated into one function.

Example: Comparison for Hotspots Analysis Results

The function `foo()` is called from two places in your application, `bar1()` and `bar2()`. If you see that `foo()` became slower in result 2, use the **Top-down Tree** window (compare mode) to check whether it became slower when being called by `bar1()`, by `bar2()`, or both.

Tip

To compare results with stacks and without stacks, switch the **Call Stack Mode** filterbar option to **User/System functions** to attribute performance data to functions where samples occurred.

See Also

[Window: Top-down Tree](#)

[Bottom-up Comparison](#)

[Comparison Summary](#)

[Comparing Results](#)

Intel® VTune™ Profiler Command Line Interface

Intel® VTune™ Profiler provides a command line interface called the `vtune` tool. This is especially useful for remote analysis, scripted commands and conducting regular performance regression checks to monitor software performance over time.

The `vtune` command line interface includes an extensive set of options that you can use to execute almost every task that you handle through the GUI. You can initiate analysis via the command line, running it as a background task or on a remote system, then view the result or generate a report at your convenience.

Use the `vtune` tool for these purposes:

- Collect performance analysis data for your target application using your specified analysis type and other options.
- Generate reports from analysis results.
- Import data files collected remotely.
- Compare performance before and after optimization.

NOTE

- See the [VTune Profiler CLI Cheat Sheet](#) quick reference on VTune Profiler command line interface.
 - To access the most current command line documentation for `vtune`, enter: `vtune -help`.
 - When you perform a task through the VTune Profiler GUI, you can use the [command generation feature](#) to display the corresponding command and save it for future use.
 - You cannot create a project from the command line. You must use the GUI for this purpose.
-

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

[Run Command Line Analysis](#)

Work with Results from Command Line

from the command line

Generate Command Line Reports

vtune Command Syntax

Use the following Intel® VTune™ Profiler vtune command syntax:

`vtune <action> [-action-option] [-global-option] [[--] <target> [target-options]]`

`vtune` The name of the VTune Profiler command line tool.

`<action>` The action to perform, such as `collect` or `report`.

`[-action-option]` Action-options modify behavior specific to the action. You can have multiple action-options per action. Using an action-option that does not apply to the action results in a usage error.

NOTE

Long names of the options can be abbreviated. If the option consists of several words you can abbreviate each word, keeping the dash between them. Make sure an abbreviated version unambiguously matches the long name. For example, the `-option-name` option can be abbreviated as `-opt-name`, `-op-na`, `-opt-n`, or `-o-n`.

`[-global-option]` Global-options modify behavior in the same manner for all actions. You can have multiple global-options per action.

`[--] <target>` The target application to analyze.

NOTE

You may use `vtune` to analyze remote targets running on regular [Linux*](#) or [Android*](#) systems.

`[target-options]` Options for the application.

NOTE

See the [VTune Profiler CLI Cheat Sheet](#) quick reference on VTune Profiler command line interface.

Example

This example runs the Hotspots analysis for the `sample` target located at the `/home/test/` directory on a [Linux*](#) system, saves the analysis result in the `r001hs` subdirectory of the current directory, and displays the default summary report.

```
vtune -collect hotspots -result-dir r001hs -quiet /home/test/sample
```

where:

- `-collect` is an action

- `hotspots` is an argument of the action
- `-result-dir` is an action-option
- `r001hs` is an argument of the action-option
- `-quiet` is a global-option
- `sample` is a target

See Also

[vtune Actions](#)

[Run Command Line Analysis](#)

[Configure Analysis Options from Command Line](#)

vtune Actions

The `vtune` command tool of the Intel® VTune™ Profiler supports different command options.

Actions

<code>archive</code>	Archive collected results.
<code>collect</code>	Run the specified analysis type and collect data into a result.
<code>collect-with</code>	Run a custom hardware event-based sampling or user-mode sampling and tracing collection using your settings.
<code>command</code>	Issue a command to a running collect action.
<code>finalize</code>	Perform symbol resolution to finalize or re-resolve a result.
<code>help</code>	Display brief explanations of command line arguments.
<code>import</code>	Import one or more collection data files/directories.
<code>report</code>	Generate a specified type of report from an analysis result.
<code>version</code>	Display version information for the <code>vtune</code> tool.

NOTE

To access the most current command line documentation for an action, enter `vtune -help <action>`, where `<action>` is one of the available actions. To see all available actions, enter `vtune -help`.

Action Options

Action options define a behavior applicable to the specified action; for example, the `-result-dir` option specifies the result directory path for the `collect` action.

NOTE

To access the list of available action options for an action, enter `vtune -help <action>`, where `<action>` is one of the available actions. To see all available actions, enter `vtune -help`.

Action-Option Usage Rules:

- If opposing action-options are used on the same command line, the last specified action-option applies.
- An action-option that is redundant or has no meaning in the context of the specified action is ignored.
- Attempted use of an inappropriate action-option which would lead to unexpected behavior returns a usage error.

Global Options

Global options define a behavior applicable to all actions; for example, the `-quiet` option suppresses non-essential messages for all actions. You may have one or more global options per command.

NOTE

To access the list of available global options for an action, enter `vtune -help <action>`, where `<action>` is one of the available actions. To see all available actions, enter `vtune -help`.

Get Information on Analysis Options

VTune Profiler offers many ways to get information on analysis options.

- VTune Profiler can re-use analysis configuration options set in the GUI version and [command line](#) version of such a configuration. You can copy this command line to the clipboard and use it for the command line analysis. To do this, use the **Command Line...** button in the **Configure Analysis** window. This also works for custom analysis types.
- To get more information on an action, enter: `vtune -help <action>`. For example, this command returns help for the `collect-with` action:

```
vtune -help collect-with
```

- For information on a specific analysis type, enter: `vtune -help collect <analysis_type>` or `vtune -help collect-with <analysis_type>`. For example, this command returns help for the `threading` analysis type:

```
vtune -help collect threading
```

- For information on a specific report, enter: `vtune -help report <report_name>`. For example, this command returns help for the `callstacks` report:

```
vtune -help report callstacks
```

See Also

[vtune Command Syntax](#)

[Option Descriptions and General Rules](#)

Run Command Line Analysis

Default Installation Paths

Whether you downloaded Intel® VTune™ Profiler as a standalone component or with the Intel® oneAPI Base Toolkit, the default path for your `<install-dir>` is:

Operating System	Path to <code><install-dir></code>
Windows* OS	• C:\Program Files (x86)\Intel\oneAPI\

Operating System	Path to <install-dir>
	<ul style="list-style-type: none"> • C:\Program Files\Intel\oneAPI (in certain systems)
Linux* OS	<ul style="list-style-type: none"> • /opt/intel/oneapi/ for root users • \$HOME/intel/oneapi/ for non-root users
mac* OS	/opt/intel/oneapi/

NOTE Profiling support for macOS is deprecated and will be removed in a future release.

Run Predefined Analysis

The predefined analysis configurations already have most of the *knobs* (configuration options) set by default for your convenience. To run a predefined performance analysis, use the `-collect` action:

```
vtune-collect <analysis_type> [-target-system=<system>] [-knob <knobName=knobValue>]
[--] <target>
```

where:

- `<analysis_type>` is the type of analysis to run. To see the list of available analysis types, enter:
`vtune -help collect`
- `-target-system` is an option targeted for remote analysis and specifies a remote Linux* system or a Android* device
- `-knob` is a configuration option that modifies the analysis
- `[knobName=knobValue]` is the name of the specified knob and its value
- `<target>` is the path and name of the application to analyze. If you need to analyze a process, use the `-target-process` or `-target-pid` option to specify the process name or ID. For a system-wide analysis, no target specification is required.

Intel® VTune™ Profiler supports the following predefined analysis types:

Analysis Type	Description
performance-snapshot	Get an overview of issues that affect application performance on your target system.
hotspots	Analyze application flow and identify sections of code that take a long time to execute (hotspots).
anomaly-detection (preview)	Identify performance anomalies in frequently recurring intervals of code like loop iterations. Perform fine-grained analysis at the microsecond level.
threading	Collect data on how an application is using available logical CPU cores, discover where parallelism is incurring synchronization overhead, identify where an application is waiting on synchronization objects or I/O operations, and discover how waits affect application performance.
hpc-performance	Identify opportunities to optimize CPU, memory, and FPU utilization for compute-intensive or throughput applications. The HPC Performance Characterization analysis type is a starting point for understanding the performance landscape of your application. Use this analysis type to improve

Analysis Type	Description
	application performance by increasing the number of floating-point operations per second (GFLOPS) and reducing the overall application run time. The analysis collects data related to CPU, memory, and FPU utilization. Additional scalability metrics are available for applications that use OpenMP* or MPI runtime libraries.
memory-consumption	Analyze memory consumption by your Linux application, its distinct memory objects and their allocation stacks.
uarch-exploration (former general-exploration)	Collect hardware events for analyzing a typical client application. This analysis calculates a set of predefined ratios used for the metrics and facilitates identifying hardware-level performance problems.
memory-access	Identify memory-related issues, like NUMA problems and bandwidth-limited accesses, and attribute performance events to memory objects (data structures), which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.
sgx-hotspots (deprecated)	Analyze hotspots inside security enclaves for systems with the Intel® Software Guard Extensions (Intel® SGX) feature enabled. This analysis type uses the INST_RETIRE.PREC_DIST hardware event that emulates precise clockticks and helps identify performance-critical program units inside enclaves.
tsx-exploration (deprecated)	Collect events that help understand Intel® Transactional Synchronization Extensions (Intel® TSX) behavior and causes of transactional aborts.
tsx-hotspots (deprecated)	Monitor the UOPS_RETIRE.ALL_PS hardware event that emulates precise clockticks and identify performance-critical program units inside transactions.
gpu-hotspots (preview)	Identify Graphics Processing Unit (GPU) tasks with high GPU utilization and estimate the effectiveness of this utilization. This analysis type is intended for analysis of applications that use a GPU for rendering, video processing, and computations with explicit support of Intel® Media SDK and OpenCL™ software technology.
gpu-offload	Explore code execution on various CPU and GPU cores on your platform, correlate CPU and GPU activity, and identify whether your application is GPU or CPU bound.
graphics-rendering (preview)	Analyze the CPU/GPU utilization of your code running on the Xen virtualization platform. Explore GPU usage per GPU engine and GPU hardware metrics that help understand where performance improvements are possible. If applicable, this analysis also detects OpenGL-ES API calls and displays them on the timeline.
fpga-interaction	Analyze the CPU/FPGA interaction issues via exploring OpenCL kernels running on FPGA, identify the most time-consuming FPGA kernels.
io	Monitor utilization of the IO subsystems, CPU and processor buses. This analysis type uses the hardware event-based sampling collection and system-wide Ftrace* collection (for Linux* and Android* targets)/ETW collection (Windows* targets) to provide a consistent view of the storage sub-system combined with hardware events and an easy-to-use method to match user-level source code with I/O packets executed by the hardware.
system-overview	Monitor a general behavior of your target system and identify platform-level factors that limit performance.

Run Custom Analysis

If you need to run a modified version of the predefined analysis type, you may use the `-collect-with` action option to specify a data collection type and required configuration options (knobs):

```
vtune -collect-with <collection_type> [-target-system=<system>] [-knob
<knobName=knobValue>] [--] <target>
```

where

- `<collection_type>` is the type of analysis to run. To see the list of available collection types, enter:
`vtune -help collect-with`
- `-target-system` is an option targeted for remote analysis and specifies a remote Linux* system or a Android* device
- `<-knob>` is an option that configures the analysis
- `[knobName=knobValue]` is the name of specified knob and its value
- `<target>` is the path and name of the application to analyze. If you need to analyze a process, use the `-target-process` or `-target-pid` option to specify the process name or ID. For a system-wide analysis, no target specification is required.

Intel® VTune™ Profiler supports the following collection types:

Collector	Description
<code>runsa</code>	Profile your application using the counter overflow feature of the Performance Monitoring Unit (PMU).
<code>runss</code>	Profile the application execution and take snapshots of how that application utilizes the processors in the system. The collector interrupts a process, collects the value of all active instruction addresses and captures a calling sequence for each of these samples.

Next Steps

When the collection is complete, the VTune Profiler saves the data as an analysis result in the default or specified result directory. You can either [view the result in the GUI](#) or [generate a formatted analysis report](#).

See Also

[vtune Command Syntax](#)

[Generate Command Line Reports](#)

[Android* Targets](#)

[Set Up Remote Linux* Target](#)

[performance-snapshot Command Line Analysis](#)

Use Performance Snapshot when you want to see a summary of issues affecting your application. This analysis also includes recommendations for other analysis types that you can run next for a deeper investigation.

Syntax

```
vtune -collect performance-snapshot [-knob <knobName=knobValue>] [--] <target>
```

Knobs:

- `collect-memory-bandwidth`

Collect the data required to compute memory bandwidth.

Default value : `false`

Possible values : `true | false`

- `dram-bandwidth-limits`

Evaluate maximum achievable local DRAM bandwidth before starting the collection. This data is used to scale bandwidth metrics on the timeline and calculate thresholds.

Default value : `true`

Possible values : `true | false`

NOTE

For the most current information on available knobs (configuration options) for Performance Snapshot analysis, enter:

```
vtune -help collect performance-snapshot
```

Example

This example shows how to run Performance Snapshot on a Linux* myApplication application:

```
vtune -collect performance-snapshot -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- [Open the data collection result \(*.vtune\) in the VTune Profiler graphical interface.](#)

See Also

[Configure Analysis Options from Command Line](#)

hotspots Command Line Analysis

Hotspots analysis helps understand application flow and identify sections of code that get a lot of execution time (*hotspots*). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hotspots can be removed, while other hotspots are fundamental to the application functionality and cannot be removed.

Intel® VTune™ Profiler creates a list of functions in your application ordered by the amount of time spent in each function. It also can be configured to capture the call stacks for each of these functions so you can see how the hot functions are called.

Use the `-knob` option to specify a collection mode for the Hotspots analysis:

- `sampling-mode=sw` - User-Mode Sampling (default) used for profiling:
 - Targets running longer than a few seconds
 - A single process or a process-tree
 - Python and Intel runtimes
- `sampling-mode=hw` - Hardware Event-Based Sampling used for profiling:

- Targets running less than a few seconds
- All processes on a system, including the kernel

Syntax

```
vtune -collect hotspots -knob <knobName=knobValue> [--] <target>
```

Knobs: `sampling-mode`, `enable-stack-collection`, `sampling-interval`, `enable-characterization-insights`

NOTE

For the most current information on available knobs (configuration options) for the Hotspots analysis, enter:

```
vtune -help collect hotspots
```

Example

This example shows how to run the Hotspots analysis in the user-mode sampling mode for a Linux* myApplication:

```
vtune -collect hotspots -knob sampling-mode=sw -- /home/test/myApplication
```

This example shows how to run the Hotspots analysis in the hardware event-based sampling mode for a Windows* myApplication:

```
vtune -collect hotspots -knob sampling-mode=hw -knob sampling-interval=1 -- C:\test\myApplication.exe
```

NOTE

The hardware event-based sampling mode replaced the advanced-hotspots analysis starting with VTune Amplifier 2019.

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- [Open the data collection result \(*.vtune\)](#) in the VTune Profiler graphical interface.

See Also

[Hotspots Analysis for CPU Usage Issues](#)

[Configure Analysis Options from Command Line](#)

anomaly-detection Command Line Analysis

Use Anomaly Detection to identify performance anomalies in frequently recurring intervals of code like loop iterations. Perform fine-grained analysis at the microsecond level.

NOTE

This is a **PREVIEW FEATURE**. A preview feature may or may not appear in a future production release. It is available for your use in the hopes that you will provide feedback on its usefulness and help determine its future. Data collected with a preview feature is not guaranteed to be backward compatible with future releases.

Syntax

```
vtune -collect anomaly-detection [-knob <knobName=knobValue>] [--] <target>
```

Knobs:

- [ipt-regions-to-load](#)

Specify the maximum number (10-5000) of code regions to load for detailed analysis. To load details efficiently, maintain this number at or below 1000.

Default value : 1000

Range : 10-5000

- [max-region-duration](#)

Specify the maximum duration (0.001-1000ms) of analysis per code region.

Default value : 100 ms

Range : 0.001-1000ms

NOTE

For the most current information on available knobs (configuration options) for Anomaly Detection analysis, enter:

```
vtune -help collect anomaly-detection
```

Example

This example shows how to run Anomaly Detection analysis on a sample application called `myApplication`. The analysis runs over 1000 code regions, analyzing each region for 300 ms.

```
vtune -collect anomaly-detection -knob ipt-regions-to-load=1000 -knob max-region-duration=300  
-- /home/test/myApplication
```

If you want to transfer the collected data to a different system for analysis, you must archive the result by moving all related binaries to the result folder. After Anomaly Detection analysis completes, run this command:

```
vtune -archive -r <location_of_result>
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a `.txt` or `.csv` file
- [Open the data collection result \(*.vtune\)](#) in the VTune Profiler graphical interface.

See Also

[Configure Analysis Options from Command Line](#)

threading Command Line Analysis

Threading analysis helps identify the cause of ineffective processor utilization and shows where your application is not parallel. One of the most common problems is threads waiting too long on synchronization objects (locks). Performance suffers when waits occur while cores are under-utilized.

NOTE

Threading analysis combines and replaces the Concurrency and Locks and Waits analysis types available in previous versions of Intel® VTune™ Profiler.

Threading analysis uses [user-mode sampling and tracing collection](#). With this analysis you can estimate the impact each synchronization object has on the application and understand how long the application had to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O.

There are two groups of synchronization objects supported by the Intel® VTune™ Profiler:

- objects usually used for synchronization between threads, such as Mutex or Semaphore
- objects associated with waits on I/O operations, such as Stream

Syntax

```
vtune -collect threading [-knob <knobName=knobValue>] [--] <target>
```

Knobs: [sampling-interval](#)

NOTE

For the most current information on available knobs (configuration options) for the Threading analysis, enter:

```
vtune -help collect threading
```

Example

This example shows how to run the Threading analysis on a Linux* myApplication application:

```
vtune -collect threading -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a .txt or .csv file
- [Open the data collection result \(*.vtune\)](#) in the VTune Profiler graphical interface.

See Also

[Configure Analysis Options from Command Line](#)

memory-consumption Command Line Analysis

Use the Memory Consumption analysis for your Linux* native or Python* targets to explore memory consumption (RAM) over time and identify memory objects allocated and released during the analysis run.

During Memory Consumption analysis, the VTune Profiler data collector intercepts memory allocation and deallocation events and captures a call sequence (stack) for each allocation event (for deallocation, only a function that released the memory is captured).

Syntax

```
vtune -collect memory-consumption [-knob <knobName=knobValue>] [--] <target>
```

Knobs: `mem-object-size-min-thres`.

NOTE

For the most current information on available knobs (configuration options) for the Memory Consumption analysis, enter:

```
vtune -help collect memory-consumption
```

Example

This example shows how to run the Memory Consumption analysis on a Python test application:

```
vtune -collect memory-consumption -app-working-dir /usr/bin -- python /localdisk/sample/test.py
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[Memory Consumption Analysis](#)

configuration from GUI

[Memory Consumption and Allocations View](#)

hpc-performance Command Line Analysis

Intel® VTune™ Profiler introduces the HPC Performance Characterization analysis based on applications that are compute-sensitive.

HPC Performance Characterization analysis helps identify opportunities to optimize CPU, memory, and FPU utilization for compute-intensive or throughput applications. The HPC Performance Characterization analysis type is a starting point for understanding the performance landscape of your application. Use this analysis type to improve application performance by increasing the number of floating-point operations per second (GFLOPS) and reducing the overall application run time. The analysis collects data related to CPU, memory, and FPU utilization. Additional scalability metrics are available for applications that use OpenMP or MPI runtime libraries.

Syntax

```
vtune -collect hpc-performance [-knob <knobName=knobValue>] [--] <target>
```

Knobs: `sampling-interval`, `enable-stack-collection`, `collect-memory-bandwidth`, `dram-bandwidth-limits`, `analyze-openmp`, `collect-affinity`.

NOTE

For the most current information on available knobs (configuration options) for the HPC Performance Characterization analysis, enter:

```
vtune -help collect hpc-performance
```

Example

The following example runs the HPC Characterization analysis on a Linux* application with enabled memory bandwidth analysis:

```
vtune -collect hpc-performance -knob collect-memory-bandwidth=true ./home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[HPC Performance Characterization Analysis](#)

[Configure Analysis Options from Command Line](#)

uarch-exploration Command Line Analysis

Use the `uarch-exploration` value to launch the Microarchitecture Exploration analysis (formerly known as General Exploration) that is a good starting point to triage hardware issues in your application. Once you have used Hotspots analysis to determine hotspots in your code, you can perform Microarchitecture Exploration analysis to understand how efficiently your code is passing through the core pipeline. During Microarchitecture Exploration analysis, Intel® VTune™ Profiler collects a complete list of events for analyzing a typical client application. It calculates a set of predefined ratios used for the metrics and facilitates identifying hardware-level performance problems.

Syntax

```
vtune -collect uarch-exploration [-knob [knobName=knobValue]] [--] <target>
```

Knobs: `collect-memory-bandwidth`, `pmu-collection-mode`, `dram-bandwidth-limits`, `sampling-interval`, `collect-frontend-bound`, `collect-bad-speculation`, `collect-memory-bound`, `collect-core-bound`, `collect-retiring`.

By default, the Microarchitecture Exploration analysis runs in the detailed PMU collection mode and collects sub-metrics for all top-level metrics: CPU Bound, Memory Bound, Front-End Bound, Bad Speculation, and Retiring. If required, you may configure the `knob` option to disable collecting sub-metrics for a particular top-level metric.

NOTE

- For the most current information on available knobs (configuration options) for the Microarchitecture Exploration analysis, enter:


```
vtune -help collect uarch-exploration
```
 - The `general-exploration` analysis type value is deprecated. Make sure to use the `uarch-exploration` option instead.
-

Examples

This example runs the Microarchitecture Exploration analysis on a Linux* matrix app with enabled memory bandwidth analysis:

```
vtune -collect uarch-exploration -knob collect-memory-bandwidth=true -- /home/test/matrix
```

This example runs the Microarchitecture Exploration analysis on a Windows matrix app in the low-overhead summary profiling mode:

```
vtune -collect uarch-exploration -knob pmu-collection-mode=summary -- C:\samples\matrix.exe
```

This example runs the Microarchitecture Exploration analysis on a Linux matrix app in the default detailed profiling mode but disables the collection of the sub-metrics for the Bad Speculation and Core Bound top-level metrics:

```
vtune -collect uarch-exploration -knob collect-bad-speculation=false -knob collect-core-bound=false -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a [.txt](#) or [.csv](#) file
- Open the data collection result ([*.vtune](#)) in the [VTune Profiler graphical interface](#).

See Also

[Microarchitecture Exploration Analysis for Hardware Issues](#)

[Configure Analysis Options from Command Line](#)

memory-access Command Line Analysis

Memory Access analysis identifies memory-related issues, like NUMA problems and bandwidth-limited accesses, and attributes performance events to memory objects (data structures), which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.

Syntax

```
vtune -collect memory-access [-knob <knobName=knobValue>] [--] <target>
```

Knobs: [sampling-interval](#), [analyze-mem-objects](#) (Linux* targets only), [mem-object-size-min-thres](#) (Linux targets only), [dram-bandwidth-limits](#), [analyze-openmp](#).

NOTE

For the most current information on available knobs (configuration options) for the Memory Access analysis, enter:

```
vtune -help collect memory-access
```

Example

This example shows how to run the Memory Access analysis on a Linux* myApplication app, collect data on dynamic memory objects, and evaluate maximum achievable local DRAM bandwidth before the collection starts:

```
vtune -collect memory-access -knob analyze-mem-objects=true -knob dram-bandwidth-limits=true -- home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a .txt or .csv file
- [Open the data collection result \(*.vtune\) in the VTune Profiler graphical interface.](#)

See Also

[Memory Access Analysis for Cache Misses and High Bandwidth Issues](#)

[Configure Analysis Options from Command Line](#)

tsx-exploration Command Line Analysis

NOTE

This analysis is deprecated in the GUI and available from command line only.

TSX Exploration analysis type uses [hardware event-based sampling collection](#) and is targeted for the Intel® processors supporting Intel® Transactional Synchronization Extensions (Intel® TSX). This analysis type collects events that help understand Intel® Transactional Synchronization Extensions behavior and causes of transactional aborts.

Syntax

```
vtune -collect tsx-exploration [-knob <knobName=knobValue>] [--] <target>
```

Knobs: [analysis-step](#), [enable-user-tasks](#).

NOTE

For the most current information on available knobs (configuration options) for the TSX Exploration analysis, enter:

```
vtune -help collect tsx-exploration
```

Example

This example shows how to run the TSX Exploration analysis on a Linux* myApplication with enabled user tasks analysis:

```
vtune -collect tsx-exploration -knob enable-user-tasks=true -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a .txt or .csv file
- [Open the data collection result \(*.vtune\) in the VTune Profiler graphical interface.](#)

See Also

[Configure Analysis Options from Command Line](#)

tsx-hotspots Command Line Analysis

NOTE

This analysis is deprecated in GUI and available from command line only.

TSX Hotspots analysis type uses [hardware event-based sampling collection](#) and is targeted for the Intel® processors supporting Intel® Transactional Synchronization Extensions (Intel® TSX). This analysis type uses the UOPS_RETIRED.ALL_PS hardware event that emulates precise clockticks and helps identify performance-critical program units inside transactions.

Syntax

```
vtune -collect tsx-hotspots [-knob <knobName=knobValue>] [--] <target>
```

Knobs: [sampling-interval](#), [enable-stack-collection](#).

NOTE

For the most current information on available knobs (configuration options) for the TSX Hotspots analysis, enter:

```
vtune -help collect tsx-hotspots
```

Example

This example shows how to run the TSX Hotspots analysis on a Linux* myApplication with enabled call stacks and thread context switches advanced collection:

```
vtune -collect tsx-hotspots -knob enable-stack-collection=true -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a .txt or .csv file
- Open the data collection result (*.vtune) in the VTune Profiler graphical interface.

See Also

[Configure Analysis Options from Command Line](#)

sgx-hotspots Command Line Analysis

NOTE

This analysis is deprecated in GUI and available from command line only.

SGX Hotspots analysis type is targeted for systems with Intel Software Guard Extensions (Intel SGX) feature enabled. It uses the INST_RETIRED.PREC_DIST hardware event that emulates precise clockticks and helps identify performance-critical program units inside security enclaves. Using the precise event is mandatory for the analysis on the systems with the Intel SGX enabled because regular non-precise events do not provide a correct instruction pointer and therefore cannot be attributed to correct modules.

Syntax

```
vtune -collect sgx-hotspots [-knob <knobName=knobValue>] [--] <target>
```

Knobs: `sampling-interval`, `enable-user-tasks`.

NOTE

For the most current information on available knobs (configuration options) for the SGX Hotspots analysis, enter:

```
vtune -help collect sgx-hotspots
```

Example

The following example shows how to run the SGX Hotspots Analysis on a Linux* myApplication:

```
vtune -collect sgx-hotspots -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[Configure Analysis Options from Command Line](#)

gpu-hotspots Command Line Analysis

Use the `gpu-hotspots` value to launch the GPU Compute/Media Hotspots analysis to:

- Explore GPU kernels with high GPU utilization, estimate the effectiveness of this utilization, identify possible reasons for stalls or low occupancy and options.
- Explore the performance of your application per selected GPU metrics over time.
- Analyze the hottest SYCL* standards or OpenCL™ kernels for inefficient kernel code algorithms or incorrect work item configuration.

Configure Characterization Analysis

Use the **Characterization** configuration option to:

- Monitor the Render and GPGPU engine usage (Intel Graphics only)
- Identify the loaded parts of the engine
- Correlate GPU and CPU data

When you select the **Characterization** radio button, you can select platform-specific presets of GPU metrics. With the exception of the Dynamic Instruction Count preset, all other presets collect the following data about the activity of Execution Units (EU):

- EU Array Active
- EU Array Stalled
- EU Array Idle
- Computing Threads Started
- Thread Occupancy
- Core Frequency

Each preset introduces additional metrics:

- The **Overview** metric set includes additional metrics that track general GPU memory accesses such as Memory Read/Write Bandwidth and XVE pipelines utilization. These metrics can be useful for both graphics and compute-intensive applications.
- The **Global Memory Accesses** metric group includes additional metrics that show the bandwidth between the GPU and system memory as well as bandwidth between GPU stacks. The farther a memory level is located from an XVE, the greater the impact on its performance by unnecessary access operations to the memory level.
- The **LSE/SLM Accesses** metric group includes metrics which cover the XVE to L1 cache traffic. This metric group requires two application runs to collect information.
- The **HDC Accesses** metric group includes metrics which measure the traffic between XVE and L3, that is passing by the L1 cache.
- The **Full Compute** metric group is a combination of all of the other event sets. Therefore, it requires multiple application runs.
- The **Dynamic Instruction Count** metric group counts the execution frequency of specific classes of instructions. With this metric group, you also get an insight into the efficiency of SIMD utilization by each kernel.

NOTE

You can run the GPU Compute/Media Hotspots analysis in Characterization mode for Windows* and Linux* targets. However, for all presets (with the exception of the **Dynamic Instruction Count** preset), you must have root/administrative privileges to run the GPU Compute/Media Hotspots analysis in Characterization mode.

Alternatively, on Linux* systems, you can configure the system to allow further collections for non-privileged users. To do this, in the `bin64` folder of your installation directory, run the [`prepare-debugfs-and-gpu-environment.sh`](#) script with root privileges.

Configure Source Analysis

In the Source Analysis, VTune Profiler helps you identify performance-critical basic blocks, issues caused by memory accesses in the GPU kernels.

- **Basic Blocks Latency** option helps you identify issues caused by algorithm inefficiencies. In this mode, VTune Profiler measures the execution time of all basic blocks. Basic block is a straight-line code sequence that has a single entry point at the beginning of the sequence and a single exit point at the end of this sequence. During post-processing, VTune Profiler calculates the execution time for each instruction in the basic block. So, this mode helps understand which operations are more expensive.
- **Memory Latency** option helps identify latency issues caused by memory accesses. In this mode, VTune Profiler profiles memory read/synchronization instructions to estimate their impact on the kernel execution time. Consider using this option, if you ran the GPU Compute/Media Hotspots analysis in the Characterization mode, identified that the GPU kernel is throughput or memory-bound, and want to explore which memory read/synchronization instructions from the same basic block take more time.

In the **Basic Block Latency** or **Memory Latency** profiling modes, the GPU Compute/Media Hotspots analysis uses these metrics:

- **Estimated GPU Cycles**: The average number of cycles spent by the GPU executing the profiled instructions.
- **Average Latency**: The average latency of the memory read and synchronization instructions, in cycles.
- **GPU Instructions Executed per Instance**: The average number of GPU instructions executed per one kernel instance.
- **GPU Instructions Executed per Thread**: The average number of GPU instructions executed by one thread per one kernel instance.

If you enable the **Instruction count** profiling mode, VTune Profiler shows a breakdown of instructions executed by the kernel in the following groups:

Control Flow group	if, else, endif, while, break, cont, call, calla, ret, goto, jmpi, brd, brc, join, halt and mov, add instructions that explicitly change the ip register.
Send & Wait group	send, sends, sendc, sendsc, wait
Int16 & HP Float Int32 & SP Float Int64 & DP Float groups	<p>Bit operations (only for integer types): and, or, xor, and others.</p> <p>Arithmetic operations: mul, sub, and others; avg, frc, mac, mach, mad, madm.</p> <p>Vector arithmetic operations: line, dp2, dp4, and others.</p> <p>Extended math operations.</p>
Other group	Contains all other operations including nop.

In the **Instruction count** mode, VTune Profiler also provides **Operations per second** metrics calculated as a weighted sum of the following executed instructions:

- Bit operations (only for integer types):
 - and, not, or, xor, asr, shr, shl, bfre, bfe, bfi1, bfi2, ror, rol - weight 1
- Arithmetic operations:
 - add, addc, cmp, cmpp, mul, rndu, rndd, rnde, rndz, sub - weight 1
 - avg, frc, mac, mach, mad, madm - weight 2
- Vector arithmetic operations:
 - line - weight 2
 - dp2, sad2 - weight 3
 - lrp, pln, sada2 - weight 4
 - dp3 - weight 5
 - dph - weight 6
 - dp4 - weight 7
 - dp4a - weight 8
- Extended math operations:
 - math.inv, math.log, math.exp, math.sqrt, math.rsq, math.sin, math.cos (weight 4)
 - math.fdiv, math.pow (weight 8)

NOTE

The type of an operation is determined by the type of a destination operand.

Syntax

```
vtune -collect gpu-hotspots [-knob <knobName=knobValue>] -- <target> [target_options]
```

Knobs: [gpu-sampling-interval](#), [profiling-mode](#), [characterization-mode](#), [code-level-analysis](#), [collect-programming-api](#), [computing-task-of-interest](#), [target-gpu](#).

NOTE

For the most current information on available knobs (configuration options) for the GPU Compute/Media Hotspots analysis, enter:

```
vtune -help collect gpu-hotspots
```

Example

This example runs the `gpu-hotspots` analysis in the default characterization mode with the default overview GPU hardware metric preset:

```
vtune -collect gpu-hotspots -knob enable-gpu-runtimes=true -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[Optimize applications for Intel® GPUs with Intel® VTune Profiler](#)
[GPU Compute/Media Hotspots Analysis \(Preview\)](#)

[Configure Analysis Options from Command Line](#)

gpu-offload Command Line Analysis

Explore code execution on various CPU and GPU cores on your platform, correlate CPU and GPU activity, and identify whether your application is GPU or CPU bound.

Syntax

```
vtune -collect gpu-offload [-knob <knobName=knobValue>] -- <target> [target_options]
```

Knobs: `collect-cpu-gpu-bandwidth`, `collect-programming-api`, `enable-stack-collection`, `enable-characterization-insights`, `target-gpu`.

NOTE

For the most current information on available knobs (configuration options) for the GPU Offload analysis, enter:

```
vtune -help collect gpu-offload
```

Example

This example runs GPU Offload analysis with enabled tracing for GPU programming APIs on the specified Linux* application:

```
vtune -collect gpu-offload -knob collect-programming-api=true -- /home/test/myApplication
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[Optimize applications for Intel® GPUs with Intel® VTune Profiler](#)

GPU Offload Analysis

npu

Use the NPU Exploration analysis to profile and optimize artificial intelligence(AI) workloads running on Intel architectures.

Syntax

```
vtune -collect npu [-knob <knobName=knobValue>] -- <target> [target_options]
```

Knobs: `profiling_mode`, `sampling_interval`, `metrics_set`

NOTE

For the most current information on available knobs (configuration options) for the NPU Exploration analysis, enter:

```
vtune -help collect npu
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[NPU Exploration Analysis \(Preview\)](#)

graphics-rendering Command Line Analysis

Use the `graphics-rendering` value to launch the GPU Rendering analysis (preview) and estimate your code performance based on the GPU usage per engine and GPU hardware metrics.

It focuses on the following usage models:

- System-wide profiling on all virtual domains (Dom0, DomUs) running under the Xen* hypervisor to identify domains that take too many resources and introduce a bottleneck for the whole platform. Use the `-target-system` option to specify a remote machine connected to your host via SSH.
- Profiling of OpenGL-ES applications running on Linux* systems to detect performance-critical API calls. For this mode, specify the application to analyze or a process to attach to, using the `-target-process` or `-target-pid` options.

Prerequisites

For successful analysis, make sure to configure your system as follows:

- For Xen virtualization platforms:
 - [Virtualize CPU performance counters on a Xen platform](#) to enable full-scale event-based sampling.
 - Establish a [password-less SSH connection](#) to the remote target system with the Xen platform installed.
- To analyze Intel® HD and Intel® Iris® Graphics hardware events on a GPU, make sure to [set up your system for GPU analysis](#)

Syntax

```
vtune [--target-system=ssh:username@hostname[:port]] --collect graphics-rendering [--knob <knobName=knobValue>] -- [target] [target_options]
```

Knobs: `gpu-sampling-interval`, `gpu-counters-mode=render-basic`.

NOTE

For the most current information on available knobs (configuration options) for the GPU Rendering, enter:

```
vtune -help collect graphics-rendering
```

Example

This example runs system-wide GPU Rendering analysis for a remote Xen target:

```
host>./vtune --target-system=ssh:user1@172.16.254.1 --collect graphics-rendering --duration 0
```

This example profiles an OpenGL-ES app running the GPU Rendering analysis:

```
host>./vtune --collect graphics-rendering --target-process process1
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[GPU Rendering Analysis \(Preview\)](#)

[Profile Targets on a Xen* Virtualization Platform](#)

[Configure SSH Access for Remote Collection](#)

[Configure Analysis Options from Command Line](#)

fpga-interaction Command Line Analysis

Use the CPU/FPGA Interaction analysis to assess the balance between CPU and FPGA in systems with FPGA hardware that run Data Parallel C++ (SYCL) or OpenCL™ applications. Review FPGA time spent executing kernels, overall time for memory transfers between the CPU and FPGA, and wait time impact on CPU and FPGA work loads.

Syntax

```
vtune -collect fpga-interaction [-knob <knobName=knobValue>] [--] <target>
```

Knobs: `sampling-interval`, `enable-stack-collection`.

NOTE

For the most current information on available knobs (configuration options) for the CPU/FPGA Interaction analysis, enter:

```
vtune -help collect fpga-interaction
```

Example

This example runs the CPU/FPGA Interaction analysis on an application with stack collection enabled:

```
vtune -collect fpga-interaction -knob enable-stack-collection=true -- /home/test/myApplication
```

See Also

[CPU/FPGA Interaction Analysis](#)

io Command Line Analysis**Syntax**

```
vtune -collect io [-knob <knobName=knobValue>] [-- target] [target_options]
```

Knobs**Platform-Level Metric Knobs:**

Knob	Allowed Values	Default Value	Description
collect-pcie-bandwidth	true/false	true	Collect data for: <ul style="list-style-type: none">• Inbound bandwidth (Intel® Data Direct I/O)• Outbound bandwidth (Memory-Mapped I/O)• L3 misses• Average latencies of inbound I/O requests
mmio	true/false	false	Collect the data required to locate code that induces outbound I/O traffic by accessing devices through MMIO space.
iommu	true/false	false	Collect the data required to calculate performance metrics for Intel® Virtualization Technology for Directed I/O (Intel VT-d).
collect-memory-bandwidth	true/false	true	Collect the data required to compute memory, persistent memory and cross-socket bandwidth.
dram-bandwidth-limits	true/false	true	Evaluate maximum achievable local DRAM bandwidth before the collection starts. This data is used to scale bandwidth metrics on the timeline and calculate thresholds.

OS- and API-level Metric Knobs:

Knob	Allowed Values	Default Value	Description
dpidk	true/false	false	Collect DPDK metrics. Make sure DPDK is built with VTune Profiler support.
spdk	true/false	false	Collect SPDK metrics. Make sure SPDK is built with VTune Profiler support.
kernel-stack	true/false	false	Profile Linux kernel I/O stack.

Prerequisites

Linux* OS:

Load the sampling driver or use [driverless hardware event collection](#) (Linux).

See the [Input and Output analysis User Guide](#) for detailed prerequisites for each metric type.

FreeBSD* OS:

Install the FreeBSD target package and configure your system following [the instructions](#).

Examples

Example 1: Input and Output Analysis — Launch a Target Application

Run the Input and Output analysis with Intel® VT-d metrics collection enabled for the target application <app>:

```
vtune -collect io -knob iommu=true -- <app>
```

Example 2: Input and Output Analysis – Attach to Target Application

Run the Input and Output analysis with Intel VT-d and SPDK metrics collection and without MMIO access feature in the Attach to Process mode.

Attach by **process name**:

```
vtune -collect io -knob iommu=true -knob mmio=false -knob spdk=true --target-process=<process_name>
```

Or attach by **PID**:

```
vtune -collect io -knob iommu=true -knob mmio=false -knob spdk=true --target-pid=<pid>
```

Example 3: Input and Output Analysis - Profile System

Run a system-wide Input and Output analysis without specific target application for 30 seconds:

```
vtune -collect io --duration 30
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the [-report](#) action to view the data from command line.
- Use the [-report-output](#) action to write report to a [.txt](#) or [.csv](#) file
- Open the data collection result ([*.vtune](#)) in the [VTune Profiler graphical interface](#).

See Also

[Input and Output Analysis](#)

[Input and Output analysis](#)

[system-overview Command Line Analysis](#)

System Overview analysis evaluates general behavior of Linux* or Android* target systems and correlates power and performance metrics with IRQ handling. This analysis type uses the [driverless event-based sampling collection](#).

Syntax

```
vtune -collect system-overview [-knob <knobName=knobValue>] -- <target>
```

Knobs: [collecting-mode](#), [sampling-interval](#), [enable-interrupts-collection](#), [analyze-throttling-reasons](#).

NOTE

For the most current information on available knobs (configuration options) for the System Overview analysis, enter:

```
vtune -help collect system-overview
```

Example 1:

This example runs the System Overview analysis on a guest OS via Kernel-based Virtual Machine with specified kallsyms and modules files paths.

```
vtune -collect system-overview -analyze-kvm-guest -kvm-guest-kallsyms=/home/vtune/[guest]/  
kallsyms -kvm-guest-modules=/home/vtune/[guest]/modules
```

Example 2:

This example runs the System Overview analysis for the matrix application in the low-overhead hardware tracing mode.

```
vtune -collect system-overview -knob collecting-mode=hw-tracing -- /root/intel/vtune/sample/  
matrix/matrix
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[System Overview Analysis](#)

from GUI

[Analyze Latency Issues](#)

[Configure Analysis Options from Command Line](#)

runsa/runss Custom Command Line Analysis

Use the `collect-with` action to configure and run a custom analysis using any of the following data collectors:

- [runsa](#)
- [runss](#)

runsa

The [hardware event-based sampling collector](#) of the VTune Profiler profiles your application using the counter overflow feature of the Performance Monitoring Unit (PMU).

Syntax:

```
vtune -collect-with runsa [-knob <knobName=knobValue>] [--] <target>
```

NOTE

For the most current information on available knobs (configuration options) for the hardware event-based sampling, enter:

```
vtune -help collect-with runsa
```

To display a list of events available on the target PMU, enter:

```
vtune -collect-with runsa -knob event-config=? <target>
```

The command returns names and short descriptions of available events. For more information on the events, use [Intel Processor Events Reference](#)

Example 1:

This example runs a custom hardware event-based sampling collection for the `sample` application with the specified events:

```
vtune -collect-with runsa -knob event-
config=CPU_CLK_UNHALTED.CORE,CPU_CLK_UNHALTED.REF,INST_RETIREANY -- /home/test/sample
```

Example 2:

This example configures and runs a custom event-based sampling data collection with the stack size limited to 8192 bytes and defines a custom [Sample After value](#) for the CPU_CLK_UNHALTED.REF_TSC event using the `sa` option:

```
vtune -collect-with runsa -knob enable-stack-collection=true -knob stack-size=8192 -knob -knob
event-config=CPU_CLK_UNHALTED.REF_TSC:sa=1800000,CPU_CLK_UNHALTED
```

runss

The [user-mode sampling and tracing collector](#) profiles an application execution and takes snapshots of how that application utilizes the processors in the system. The collector interrupts a process, collects the value of all active instruction addresses and captures a calling sequence for each of these samples.

Syntax:

```
vtune-collect-with runss [-knob <knobName=knobValue>] [--] <target>
```

NOTE

For the most current information on available knobs (configuration options) for the user-mode sampling and tracing, enter:

```
vtune -help collect-with runss
```

Example:

This example runs user-mode sampling and tracing collection for the `sample` application with enabled loop analysis.

```
vtune -collect-with runss -knob analyze-loops=true -- /home/test/sample
```

What's Next

When the data collection is complete, do one of the following to view the result:

- Use the `-report` action to view the data from command line.
- Use the `-report-output` action to write report to a `.txt` or `.csv` file
- Open the data collection result (`*.vtune`) in the VTune Profiler graphical interface.

See Also

[collect-with](#)
action

[Configure Analysis Options from Command Line](#)

Configure Analysis Options from Command Line

For performance analysis via Intel® VTune™ Profiler command line interface (`vtune` tool), you can configure the following options:

Collect System-Wide Data from Command Line

To extend your analysis and collect performance data for other processes running on your system, you may choose between two options:

- [system analysis with a target application](#)
- [system analysis without a target application](#)

NOTE

System-wide collection is available for [Hardware Event-based Sampling Collection](#) types only.

System Analysis with a Target Application

To analyze your target application AND all other processes running on your system at the moment, specify your application and enable system-wide analysis with the `-analyze-system` option. In this mode, the collection duration is defined by the duration of your application execution.

This mode is particularly convenient for types of the collection that require an application to be launched. For example, you may run a Frame or Task analysis (available for application targets only) and collect system-wide data at the same time.

Example

This example runs the Hotspots analysis in the hardware event-based sampling for the `sample` application and collects data system-wide.

```
vtune -collect hotspots -knob sampling-mode=hw -analyze-system -- /home/test/sample
```

The following example runs the Microarchitecture Exploration analysis (former General Exploration) for the `sample` application, including user tasks specified in your code via the Task API, and collects data system-wide.

```
vtune -collect uarch-exploration -knob enable-user-tasks=true -- /home/test/sample
```

System Analysis without a Target Application

To profile your system without specifying a target application (equal to the **Profile System** target type in GUI), you just need to specify the collection duration.

Example

This example runs a system-wide Hotspots analysis hardware event-based sampling for 60 seconds.

```
vtune -collect hotspots -knob sampling-mode=hw --duration 60
```

See Also

[analyze-system](#)
option

Set Up Analysis Target from GUI

Collect Data on Remote Linux* Systems from Command Line

Intel® VTune™ Profiler enables you to collect data on a remote application from the host system ([Remote Performance Analysis Workflow for Linux* Systems](#)) via command line interface (`vtune`) and view the analysis result locally in the GUI. Remote data collection using the `vtune` command running on the host is similar to the native collection on the target except that the `target-system` option is added to the command line.

Prerequisites:

- Intel® VTune™ Profiler is installed on the local host.
- [Target Linux* system is set up for remote analysis](#).

Depending on your remote system, you may choose to install the VTune Profiler remote target package or full command line interface (`vtune`).

- A password-less [SSH access](#) to the target is set.
- Recommended: an analysis target located on a shared drive visible to both local and remote machines.

NOTE

If you plan to collect data remotely using the full-scale command line interface of the VTune Profiler installed on your target Linux system, see the topic [Running Command Line Analysis](#). You may use the **Command Line** option in the VTune Profiler graphical interface to automatically generate a command line for an analysis configuration selected in the GUI. Make sure to edit the generated command line for remote collection as described in the [Generating Command Line Configuration from GUI](#) topic.

Use the following command line syntax to run the analysis on remote Linux system:

```
host>./vtune -target-system=ssh:user@target <-action> <analysis_type> [<-knob>
[knobName=knobValue]] [-target-tmp-dir=PATH] [-target-install-dir=PATH] [--] <target>
```

where

- `-target-system=ssh:user@target` is a remote Linux target
- `<-action>` is the action to perform the analysis (`collect` or `collect-with`)
- `<analysis_type>` is the type of analysis
- `<-knob>` is a configuration option that modifies the analysis. For a list of available knobs, enter:

```
vtune -help <action> <analysis_type>
• [knobName=knobValue] is the name of specified knob and its value
• [-target-tmp-dir=PATH] is a path to the temporary directory on the remote system where
  performance results are temporarily stored
• [-target-install-dir=PATH] is a path to the VTune Profiler target package installed on the remote
  system
• <target> is the path and name of the application to analyze
```

Examples

Example 1: Event-based System-wide Sampling Collection

The command line below collects system-wide Hotspots analysis information without call stacks. This command automatically pulls in modules required for viewing results from the device and caches them in the `temp` directory on the host. This happens only on the first collection, all subsequent collections reuse modules from the cache.

```
host>./vtune -target-system=ssh:user1@172.16.254.1 -collect hotspots -knob sampling-mode=hw -
duration 10
```

For system-wide collection, a lot of modules running in the system during collection are copied from the target to the host, which may take a while. However, this happens only once since vtune caches target system modules on the host for faster access on the next collection. If you do not want the command to take the modules from the device, you can specify a local directory where modules will be searched first, for example:

```
host>./vtune -target-system=ssh:user1@172.16.254.1 -collect hotspots -knob sampling-mode=hw -duration 10 -search-dir /search/path
```

In the case above, <PATH> can be either a directory where modules are located, or it can be a pointer to the root file system of the target device. For example, when the collector searches for the /usr/lib64/libstdc++.so.6.0.16 file from the target device, it first tries <PATH>/usr/lib64/libstdc++.so.6.0.16, then it tries <PATH>/libstdc++.so.6.0.16, and only after that it attempts to copy the file from the target device.

Example 2: Event-based Sampling Collection

This example shows how to attach the analysis to a running application by its PID.

```
host>./vtune -target-system=ssh:user1@172.16.254.1 -collect hotspots -knob sampling-mode=hw -target-pid 333
```

Example 3: Advanced Event-based Sampling Collection

You can take any event supported by the Performance Monitoring Unit (PMU). Additionally, you can enable multiple event collection at a time.

The following example identifies potential latency or responsiveness issues:

```
host>./vtune -target=ssh:user1@172.16.254.1 -duration 10 -collect-with runsa -knob event-config='CPU_CLK_UNHALTED.REF:sa=20000'
```

This command line takes samples at ~2x the rate of a context switch, which gives you an approximately 20% performance hit.

See Also

[Set Up Remote Linux* Target](#)

[Set Up Linux* System for Remote Analysis](#)

[Specify Search Directories from Command Line](#)
from the command line

Configure GPU Analysis from Command Line

Use the –knob option for configuring Intel® VTune™ Profiler to profile applications that use a Graphics Processing Unit (GPU) for rendering, video processing, and computations. GPU analysis monitors overall GPU activity (graphics, media, and compute), collects Intel® HD Graphics and Intel® Iris® Graphics hardware metrics, and then shows this data correlated with CPU processes and threads.

The following knobs are supported for GPU analysis:

Knob Name	Supported Analysis Types	Description
enable-gpu-usage=true false	runss, runsa	Analyze frame rate and usage of Processor Graphics engines.

Knob Name	Supported Analysis Types	Description
gpu-counters-mode=none overview global-local-accesses compute-extended full-compute render-basic	gpu-hotspots, graphics-rendering, gpu-offload, runss, runsa	<p>Analyze performance data from Processor Graphics based on the GPU Metrics Reference.</p> <ul style="list-style-type: none"> • <code>overview</code> - track general GPU memory accesses such as Memory Read/Write Bandwidth, GPU L3 Misses, Sampler Busy, Sampler Is Bottleneck, and GPU Memory Texture Read Bandwidth. These metrics can be useful for both graphics and compute-intensive applications. • <code>global-local-accesses</code> - include metrics that distinguish accessing different types of data on a GPU: Untyped Memory Read/Write Bandwidth, Typed Memory Read/Write Transactions, SLM Read/Write Bandwidth, Render/GPGPU Command Streamer Loaded, and GPU EU Array Usage. This metrics are useful for compute-intensive workloads on the GPU. • <code>compute-extended</code> - analyze GPU activity on the Intel processor code name Broadwell. This metrics set is disabled for other systems. • <code>full-compute</code> - collect both <code>overview</code> and <code>compute-basic</code> metrics with the <code>allow-multiple-runs</code> option enabled to analyze all types of EUs array stalled/idle issues in the same view. • <code>render-basic</code> (preview) - collect Pixel Shader, Vertex Shader, and Output Merger metrics. <p>This option is available only for supported platforms with the Intel Graphics Driver installed.</p>
gpu-sampling-interval=<value in us>	gpu-hotspots, runss, runsa	Set the interval between GPU samples between 10 and 1000 microseconds. Default is 1000us. An interval of less than 100us is not recommended.
enable-gpu-runtimes=true false	gpu-hotspots, runss, runsa	Capture the execution time of OpenCL™ kernels and Intel Media SDK programs on a GPU, identify performance-critical GPU computing tasks, and analyze the performance per GPU hardware metrics.

NOTE

OpenCL kernels analysis is currently supported for Windows and Linux target systems with Intel HD Graphics and Intel Iris Graphics. [Intel® Media SDK Program Analysis Configuration](#) is supported for Linux targets only and should be started with root privileges.

Examples

Example 1: Running Analysis for an Intel Media SDK Application

This example starts `vtune` as root and launches the GPU Compute/Media Hotspots analysis for an Intel Media SDK application running on Linux:

```
vtune -collect gpu-hotspots -knob enable-gpu-runtimes=true -r quadrant_r001 -- BitonicSort
```

To analyze a remote Linux target from the Windows system, the same example looks as follows:

```
vtune -target-system=ssh:user1@172.16.254.1 -collect gpu-hotspots -knob enable-gpu-runtimes=true  
-r quadrant_r001 -- BitonicSort.exe
```

Example 2: Running Analysis with OpenCL Kernels Tracing

Perform GPU Compute/Media Hotspots or custom analysis, enabling the `enable-gpu-usage` knob to analyze GPU usage of a processor graphics engine, using the Overview `gpu-counters-mode` counter set, which is available only on a supported platform with an Intel Graphics Driver installed. Enable tracing of OpenCL kernels execution with the `enable-gpu-runtimes` option.

For example, to run GPU Compute/Media Hotspots analysis, collect GPU hardware metrics and trace OpenCL kernels on the BitonicSort application (`-g` is the option of the application), enter:

```
vtune -collect gpu-hotspots -knob gpu-counters-mode=overview -knob enable-gpu-runtimes=true --  
BitonicSort -g
```

GPU Analysis on Android* System

You can enable GPU analysis for algorithm analysis types on Android systems with Intel HD Graphics and Intel Iris Graphics by using the following knobs:

- `enable-gpu-usage` to analyze frame rate and usage of Intel HD Graphics and Intel Iris Graphics engines based on ftrace events
- `gpu-counters-mode` to analyze performance data from Intel HD Graphics and Intel Iris Graphics based on the preset counter sets
- `gpu-sampling interval` to specify a data collection interval between GPU samples

This example runs the GPU Compute/Media Hotspots analysis and monitors GPU usage.

```
host>./vtune -collect gpu-hotspots -target-system=android -r quadrant_r001 -target-process  
com.intel.fluid -knob enable-gpu-usage=true -knob gpu-counters-mode=overview
```

See Also

[knob](#)

option

[report](#)

action-option

[Hotspots Report](#)

Specify Search Directories from Command Line

Your binary and symbol files contain data that VTune Profiler uses when performing collections, finalizing results, generating reports, and similar actions. For proper module resolution, use the `search-dir` action-option to specify directories *on the host* that should be searched for binary and symbol files and `source-search-dir` option for searching source files.

If the `vtune` tool is not provided with the information it needs to find all the necessary files, the results may be skewed or the finalization process may fail altogether.

The finalization process writes collected data to a database, resolves symbolic information, and pre-computes data to make further analysis more efficient and responsive. Finalization can only succeed if it knows which directories to search.

During finalization, the result directory is set as the default search directory to make it easier to display the result in the GUI or generate a report from the result. When a report is generated, the `report` action requires the same modules that were used during data collection.

When generating a report from results that were imported from another system, use the `search-dir` and `source-search-dir` action-options to specify the search directories for system modules. When VTune Profiler searches for symbol/source data, the specified directories have a higher priority than absolute local paths.

To specify the search directory for symbol/binary files used by your target, run the `vtune` command using the `search-dir` option as follows:

```
vtune-report <report_type> -search-dir <search_dir> result-dir <result_dir>
```

To enable the source code view in the command line report, specify the search directory for source files using the `source-search-dir` option as follows:

```
vtune-report <report_type> -source-search-dir <search_dir> result-dir <result_dir>
```

- `<search_dir>` is the search directory to add
- `<result_dir>` is the result directory
- `<report_type>` is the type of report to display

Examples

This command generates a `callstacks` report on the `r001hs` hotspots result on a Windows* system, searching for symbol files in the `C:\Import\system_modules` high-priority search directory, and sends the report to `stdout`. `-R` is the short form of the `-report` action, and `-r` is the short form of the `result-dir` action-option.

```
vtune -R callstacks -search-dir C:\Import\system_modules -r C:\Import\r001hs
```

This command generates a `callstacks` report on the `r001hs` hotspots result on a Linux* system, searching for symbol files in the `home/system_modules` high-priority search directory, and sends the report to `stdout`. `-R` is the short form of the `-report` action, and `-r` is the short form of the `result-dir` action-option.

```
vtune -R callstacks -search-dir /home/system_modules -r /home/import/r001hs
```

When your binary/symbol files are in multiple directories, use the `search-dir` option multiple times so that all the necessary directories are searched.

```
vtune -collect hotspots -knob sampling-mode=hw -search-dir /home/my_system_modules -search-dir /home/other_system_modules -- /home/test/myApplication
```

This command opens the source view for the `foo` function annotated with the Hotspots analysis metrics data collected for the `r001hs` result. It uses the `/home/my_sources` directory to search for source files.

```
vtune -R hotspots -source-object function=foo -r /home/my_project/r001hs -source-search-dir /home/my_sources
```

See Also

[Import Results from Command Line](#)

[Search Directories](#)

from GUI

[Search Directories for Remote Linux* Targets](#)

Specify Result Directory from Command Line

It is generally safest to specify a PATH/name for the result directory when working on the command line. When using an action that takes a result as input, it is safer to specify the result directory, even if the result was created using the default directory.

- Be sure to specify the result directory when using an action that takes a result as input, especially the `finalize` or `import` action.

- You want to store the result in a different directory than the current working directory.
- The result is assigned a name other than the default. In this case, you would specify the result name when performing the `collect` or `collect-with` action, and also when generating a report or performing any other actions that take this result as input.

Use the `-result-dir | -r` action-option to specify the PATH/name of a result directory. This may be an absolute path, or a path relative to the current working directory.

- To specify the directory path but use the default naming conventions for the directory, just specify the path.
- To specify the name of the result directory, but have the result written to the current working directory, just specify a name for the result directory.

NOTE

Use the `user-data-dir` action-option to specify the base directory for result paths.

Example

This command runs the Hotspots analysis of `myApplication` in the current working directory, which is named `test`. The result is saved in a default-named directory under the `/home/test/` directory. If this was the first Hotspots analysis run, the result directory would be named `r000hs`.

```
vtune -collect hotspots -result-dir=/home/test/ -- /home/test/myApplication
```

To generate a report from this result, you must specify the result directory.

```
vtune -report hotspots -r=/home/test/ -- /home/test/myApplication
```

See Also

[result-dir](#) action-option

[user-data-dir](#)
action-option

Pause Collection from Command Line

Intel® VTune™ Profiler offers different ways to pause the collection process, to resume a paused collection, or to stop a running collect process from the command line.

Start collection in the paused mode, and then automatically resume collection

To start data collection in the paused mode, use the `start-paused` action option as follows:

```
vtune -collect <analysis_type> -start-paused [--] <target>
```

Resume a paused collection

There are two ways to resume a paused collection.

- To resume collection automatically after a specified amount of time, use the `resume-after` option.

```
vtune -collect <analysis_type> -start-paused -resume-after=<value> [--] <target>
```

where

- `<analysis_type>` is the type of analysis to run
- `<value>` is the time of delay in seconds
- `<target>` is the target to analyze

- To resume collection manually, use the `command` action with the `resume` argument.

```
vtune -command resume
```

Examples

This example starts the Hotspots analysis of the sample Linux* application in the paused mode, and then resumes collection after a 50 second pause.

```
vtune -collect hotspots -start-paused -resume-after=50 -- /home/test/sample
```

This example starts the Hotspots analysis of the sample Windows* application in the paused mode.

```
vtune -collect hotspots -start-paused -- C:\test\sample.exe
```

See Also

[resume-after action-option](#)

[command](#)

[action](#)

[Pause Data Collection](#)

Manage Analysis Duration from Command Line

Manage analysis duration for best results on short-running targets, or to [minimize collection overhead](#) on longer-running targets.

Use the `vtune` command interface to minimize duration while optimizing the analysis process.

- [Use Default Duration](#)
- [Adjust Collection Duration to Application](#)
- [Manually Interrupt and Restart Analysis](#)
- [Configure Collection Duration](#)

Use Default Duration

Intel® VTune™ Profiler provides predefined general analysis types to keep overhead to a reasonable level. The option reference topic for the `collect` action identifies analysis types that are recommended as starting points; and points out some more advanced analysis types that have higher overhead.

NOTE

To view all the analysis types that are available for your processor, use the command line help:

```
vtune -help collect
```

Adjust Collection Duration to Application

The sampling interval determines how much data is collected. The default sampling interval is *short*, which is appropriate for targets that complete in 1 - 15 minutes.

If your target runs shorter or longer than this, use the `target-duration-type` action-option to set the appropriate duration type, which adjusts the sampling interval.

- If the target takes less than 1 minute to run, specify *veryshort*.
- If the target takes 15 minutes to 3 hours to run, specify *medium*.
- If the target takes over 3 hours, specify *long*.

NOTE

For hardware event-based analysis types, a multiplier applies to the configured Sample After value.

Example

Perform a Hotspots analysis in the user-mode sampling mode using a medium sampling interval that is appropriate for targets with a duration of 15 minutes to 3 hours.

```
vtune -collect hotspots -target-duration-type medium -- myApp
```

Manually Interrupt and Restart Analysis

To pause an analysis manually, open a new terminal and use the [command](#) action with the *pause* argument, being sure to specify the result directory. The target process continues to run, but data collection is paused.

```
vtune -command pause -result-dir results/r002hs
```

To resume analysis, use the [command](#) action with the *resume* argument.

```
vtune -command resume -r results/r002hs
```

To stop analysis altogether, use the [command](#) action with the *stop* argument. Once analysis is stopped, it cannot be resumed.

```
vtune -command stop -r results/r002hs
```

Configure Collection Duration

VTune Profiler offers other ways to limit the analysis process. To stop analysis at a specified time after initiating target execution, use the [duration](#) option.

```
vtune -collect <analysis_type> -duration=<value> -- <target>
```

where

- <*analysis_type*> is the type of analysis to run
- <*value*> is the duration in seconds
- <*target*> is the target to analyze

NOTE

To start the analysis in the paused mode or pause the collection during the analysis, refer to [Pause Collection from the Command Line](#) section.

Examples

Example 1: Ending analysis after specified time

Start a Hotspots analysis of `myApplication` and end analysis after 60 seconds.

```
vtune -collect hotspots -duration=60 -- /home/test/myApplication
```

Example 2: Running an unlimited duration analysis

Run an unlimited duration Hotspots analysis, which will run until you stop it.

```
vtune -collect hotspots -duration=unlimited -result-dir results/r002hs
```

See Also

[Pause Data Collection](#)

from GUI

[target-duration-type](#)
action-option

[duration](#)
action-option

Limit Data Collection from Command Line

Limiting data collection prevents from collecting a large amount of data that may slow down the data processing. For example, it may happen when running Threading analysis on frequently contended applications or when analyzing long profiles.

Typically, the default maximum amount of raw data used by the Intel® VTune™ Profiler for the result file is enough to identify a problem.

To limit the amount of raw data, use any of the following options:

- [Set the maximum possible result file \(in MB\)](#)
- [Set the analysis timer for the last seconds of collection](#)

Set the Maximum Possible Result File (in MB)

Use the [data-limit](#) command line option to limit the amount of raw data to be collected by setting the maximum possible result size (in MB). VTune Profiler starts collecting data from the beginning of the target execution and suspends data collection when the specified limit for the result size is reached. For unlimited data size, specify 0.

```
vtune -collect <analysis_type> -data-limit=<value> -- <target>
```

Example

Start a Hotspots analysis on the specified Linux* target and limit the result size to 200 MB:

```
vtune -collect hotspots -data-limit=200 -- /home/test/myApplication
```

Set the Analysis Timer for the Last Seconds of Collection

Use the [ring-buffer](#) command line option to limit the amount of raw data to be collected by setting the timer that enables the analysis only for the last seconds before the target or collection is terminated. For example, if you specify 2 seconds as a time limit, the VTune Profiler starts the data collection from the very beginning but saves the collected data only for the last 2 seconds before you terminate the collection.

```
vtune -collect <analysis_type> -ring-buffer=<value> -- <target>
```

Example

Enable a Hotspots analysis on the specified Windows* target for the last 10 seconds before the collection is terminated:

```
vtune -collect hotspots -ring-buffer=10 -- C:\test\myApplication.exe
```

See Also

[data-limit](#)

command line option

[ring-buffer](#)

command line option

[Limit Data Collection](#)

from GUI

Work with Results from Command Line

Intel® VTune™ Profiler provides several ways to work with the analysis results from the command line:

View Command Line Results in the GUI

After generating, importing or finalizing a result from the command line, you can open your result in the graphical user interface for immediate access to the result windows and tools offered by the Intel® VTune™ Profiler.

View Results in Microsoft Visual Studio

To add a result to an existing Microsoft Visual Studio* project, do the following:

1. Open your project in Visual Studio.
2. In the **Intel VTune Profiler Results** folder, click the pull-down menu next to the **Profile with VTune Profiler** icon.
3. Select **Import result**.
4. Select the *.vtune result file and click the **Add** button.

The result appears in the **Intel VTune Profiler Results** folder. You can now work with the command line result exactly as with the result collected from GUI, for example: view source/assembly, filter performance data, or compare it with another result of the same analysis type.

View Results in the Standalone GUI

To open a result in the standalone interface:

1. Launch the standalone GUI interface of the VTune Profiler. To do this from the command line, enter:

```
vtune-gui
```

2. Click the



menu button, select **Open > Result...**, and navigate to the result file.

See Also

[Generate Command Line Reports](#)

[VTune Profiler Filenames and Locations](#)

Import Results from Command Line

You can collect performance data remotely with the Intel® VTune™ Profiler collectors (for example, SEP collector or Intel SoC Watch collector) or Linux* Perf* collector, import this data to the VTune Profiler project, and view the data in the graphical or command line interface. Use the [import](#) action to import data collection files. Currently the following data formats are supported:

- *.tb6/*.tb7 (sampling raw data files collected with the low-level SEP collector)
- *.perf (Perf data files)
- *.csv ([External Data Import](#) files in the predefined format)
- *.pwr (processed Intel SoC Watch files with [energy analysis](#) data)
- [Prerequisites for Importing a *.perf File](#)
- [Import Performance Profiler results and view data](#)
- [Import energy analysis results and view data](#)

Prerequisites for Importing a *.perf File

To import a *.perf file with hardware event-based sampling data collected by the Linux* Perf tool, make sure to run the Perf collection with the predefined command line options:

- For application analysis:

```
$ perf record -o <trace_file_name>.perf -call-graph dwarf -e cpu-cycles,instructions  
<application_to_launch>
```

- For process analysis:

```
$ perf record -o <trace_file_name>.perf -call-graph dwarf -e cpu-cycles,instructions
<application_to_launch> -p <PID> sleep 15
```

where the `-e` option is used to specify a list of events to collect as `-e <list of events>`; `-call-graph` option (optional) configures samples to be collected together with the thread call stack at the moment a sample is taken. See Linux Perf documentation on possible call stack collection options (for example, `dwarf`) and its availability in different OS kernel versions.

NOTE

The Linux* kernel exposes Perf API to the Perf tool starting from version 2.6.31. Any attempts to run the Perf tool on kernels prior to this version lead to undefined results or even crashes. See Linux Perf documentation for more details.

Import Performance Profiler results and view data

1. Copy the result directory to your local system.
2. Use the `import` action to import the required file, setting the imported result directory as a search directory:

```
vtune -import <result_path> -source-search-dir <search_path> -r <result_dir>
```

If you do not use the `result-dir` option, the VTune Profiler creates a new directory with the default name in the current working directory.

NOTE

To [import a CSV file with external data](#), use the `-result-dir` option and specify the name of an existing directory of the result that was collected by the VTune Profiler in parallel with the external collection. VTune Profiler adds the externally collected statistics to the result and provides integrated data in the **Timeline** pane.

3. You can use the command line to display the imported result in the VTune Profiler GUI, or generate a report to view it.

- In the GUI:

```
vtune-gui <result_dir>/<result>.vtune
```

- In the CLI:

```
vtune -report <report_type> -result-dir <result_dir>/<result>.vtune
```

NOTE

- Use the `search-dir` action-option to specify symbol and binary files locations for module resolution.
 - For Linux targets, make sure to generate the debug information for your binary files using the `-g` option for compiling and linking. This enables the VTune Profiler to collect accurate performance data.
 - To minimize the size of the result, you may use the `discard-raw-data` action-option, but this will prevent re-finalizing the result.
 - Imported result files may not have all the fields that are present in the VTune Profiler result files, so some types of data may be missing from the report.
-

Import energy analysis results and view data

Run the following command to create a VTune Profiler project with the Intel SoC Watch trace data:

```
vtune -import <path_to_file> -result-dir <project_folder>
```

where *<project_folder>* is the VTune Profiler project directory, for example, *r001*, or the full path to the result directory, for example, on Linux: /root/intel/vtune/projects/my_project/r001

NOTE

- *<project_folder>* must be a non-existing folder, or you will get an error.
 - The energy analysis data file has an extension of .pwr.
-

You may include a path with the project name to create the project in a directory other than the current directory.

VTune Profiler should start up and automatically open your project in the **Platform Power Analysis** viewpoint.

Examples

This command imports the /home/import/r001.tb6 data collection file on Linux, searching the same directory for binary and symbol information. The result is output to the current working directory.

```
vtune -import /home/Import/r001.tb6 -search-dir /home/import/r001hs
```

Generate the callstacks report from the imported r001hs Hotspots result, searching the /home/import/r001hs directory for binary and symbol information.

```
vtune -report callstacks -result-dir /home/import/r001hs -search-dir /home/import/binaries
```

See Also

[import](#)
action

[Import Results and Traces into VTune Profiler GUI](#)

[Search Directories](#)
from GUI

Re-finalize Results from Command Line

Results are finalized during collection by default, but sometimes finalization is suppressed, or a result that was finalized needs to be re-resolved. Here are some of the possible reasons:

- The no-auto-finalize action-option may be used to suppress finalization when performing the collect action. In this case, the finalize action must be performed before the result can be viewed or used to generate a report.
- Finalization may not have completed successfully. If you open the result in the GUI and see question marks or other unexpected characters, the usual cause is that vtune could not find all the source, binary and symbol files. When re-finalizing the result, use the search-dir action-option and make sure to specify all search directories.

NOTE

Raw collector data is used to re-finalize a result. If the collect action is performed with the discard-raw-data option, so that the raw data is deleted after the initial finalization, the result cannot be re-finalized.

Re-Finalize a Result

To force result re-finalization, run the `finalize` action using this general syntax:

```
vtune -finalize -result-dir <result_path> -search-dir <search_path>
```

where

`<result_path>` is the result directory and `<search_path>` is the search directory. Use the `-search-dir` option to specify directories for searching symbol and binary files.

Example

This example re-finalizes the `r001hs` result, searching for symbol files in the specified search directory.

```
vtune -finalize -result-dir r001hs -search-dir /home/import/system_modules
```

See Also

[finalize](#)
action

[Finalization](#)

Generate Command Line Reports

When you run a performance analysis from the command line, you [see the results in the Intel® VTune™ Profiler user interface](#). Use available options in the VTune Profiler GUI to filter and format the data.

You can also see the collected data as a report on the command line. The following sections describe the types of reports you can generate this way.

General Syntax

Use the following syntax to generate a report from the command line:

```
vtune report <report_type> -result-dir <result_path> [report_options]
```

where:

- `<report_type>` is the type of report that you want to create. To get the list of available report types, enter: `vtune -help report`. To display help for a specific report type, enter: `vtune -help report <report_type>`.
- `<result_path>` is a directory where your result file is located. If you do not specify a result directory, the VTune Profiler displays a report for the latest collected result.
- `[report_options]` are action options used to manage the selected report. To view a list of available report action options, enter: `vtune -help report <report_type>`.

NOTE

`-R` is the short form of the `report` action. `-r` is the short form of the `result-dir` action-option. The syntax `vtune -R <report_type> -r <result_path>` is a valid syntax to generate a command line report.

Usage Considerations

- You generate command line reports from existing results. When using the command line, you cannot collect data and generate a timeline report simultaneously. Therefore, you cannot use the `collect` and `report` actions in the same command.
- The analysis type used to collect a result determines the report types for that result that can be generated from the command line.

- By default, a report is written in text format to `stdout` and is not saved to a file. To save, filter and format reports, see the topics on [Saving and Formatting Reports](#) as well as [Filtering and Grouping Reports](#).

Types of Command Line Reports

Use the `vtune` command to generate these types of reports:

<code>affinity</code>	Display binding of a thread to a range of sockets, physical, and logical cores.
<code>callstacks</code>	Report full stack data for each hotspot function; identify the impact of each stack on the function CPU or Wait time. You can use the <code>group-by</code> or <code>filter</code> options to sort the data by: <ul style="list-style-type: none"> • <code>callstack</code> • <code>function</code> • <code>function-callstack</code>
<code>exec-query</code>	
<code>gprof-cc</code>	Report a call tree with the time (CPU and Wait time, if available) spent in each function and its children.
<code>hotspots</code>	Display collected performance metrics according to the selected analysis type and identify program units that took the most CPU time (hotspots).
<code>hw-events</code>	Display the total number of hardware events.
<code>platform-power-analysis</code>	Display CPU sleep time, wake-up reasons and CPU frequency scaling time.
<code>summary</code>	Report on the overall performance of your <code>target</code> .
<code>timeline</code>	Display metric data over time and distributed over time intervals.
<code>top-down</code>	Report call sequences (stacks) detected during collection phase, starting from the application root (usually, the <code>main()</code> function). Use this report to see the impact of program units together with their callees.
<code>vectspots</code>	Display statistics that help identify code regions to tracing on a HW simulator.

Example

This example displays a Hotspots report for the `r001hs` result. The report shows CPU time for the functions of the target in descending order, starting with the most time-consuming function.

```
vtune -report hotspots -r r001hs
```

Function	CPU Time	CPU Time:Effective	Time:Idle	CPU
Time:Poor	CPU Time:Effective	Time:Ok	CPU Time:Effective	CPU
Time:Over	CPU Time:Effective	Time:Ok	CPU Time:Effective	CPU
grid_intersect	3.371s	3.371s	0s	
0s	3.371s	0s	0s	
sphere_intersect	2.673s	2.673s	0s	
0s	2.673s	0s	0s	
render_one_pixel	0.559s	0.559s	0s	

```

0s           0.559s           0s
0s           0s
...

```

See Also

[report action](#)

[Save and Format Command Line Reports](#)

[Manage Data Views](#)

from GUI

[Filter and Group Command Line Reports](#)

Summary Report

Similar to the **Summary** window, available in GUI, the `summary` report provides overall performance data of your target. Intel® VTune™ Profiler automatically generates the summary report when data collection completes. To disable this report, use the `no-summary` option in your command when performing a `collect` or `collect-with` action.

Use the following syntax to generate the Summary report from a preexisting result:

```
vtune -report summary -result-dir <result_path>
```

The summary report output depends on the collection type:

- [User-mode Sampling and Tracing Collection Summary Report](#)
- [Hardware Event-based Sampling Collection Summary Report](#)

User-mode Sampling and Tracing Collection Summary Report

For User-Mode Sampling and Tracing Collection results, the summary report includes the following sections:

- Collection and Platform Information
- CPU Information
- Summary per basic analysis metrics

Example 1: User-Mode Sampling Hotspots Summary

This example generates the summary report for the `r000hs` Hotspots analysis result on Windows*:

```
vtune -report summary -r r000hs
```

```

Elapsed Time: 1.857s
CPU Time: 10.069s
  Effective Time: 10.069s
  Idle: 0.000s
  Poor: 1.294s
  Ok: 6.381s
  Ideal: 2.395s
  Over: 0s
Spin Time: 0s
Overhead Time: 0s
Total Thread Count: 9
Paused Time: 0s

```

Top Hotspots		
Function	Module	CPU Time
multiplyl	matrix.exe	10.069s

```
Collection and Platform Info
  Application Command Line: C:\temp\samples\en\C++\matrix_vtune\matrix\vc14\Win32\Release
  \matrix.exe
    Operating System: Microsoft Windows 10
    Computer Name: my-computer
    Result Size: 5 MB
    Collection start time: 09:41:57 06/09/2018 UTC
    Collection stop time: 09:41:58 06/09/2018 UTC
    Collector Type: Event-based counting driver,User-mode sampling and tracing
    CPU
      Name: Intel(R) Processor code named Skylake
      Frequency: 4.008 GHz
      Logical CPU Count: 8
```

Example 2: Threading Summary

This example generates a summary report for the Threading analysis result r003tr. The summary portion of the report shows that the multithreaded target spent 64 seconds waiting, with an average concurrency of only 1.073:

```
vtune -report summary -r r003tr
```

```
Summary
-----
Average Concurrency: 1.073
Elapsed Time: 13.911
CPU Time: 11.031
Wait Time: 64.468
Average CPU Usage: 0.768
```

To identify the cause of the wait, view the result in the GUI performance pane, or generate a performance report.

Hardware Event-based Sampling Collection Summary Report

For [Hardware Event-based Sampling Collection](#) results, the summary report includes the following information (if available):

- Collection and Platform information
- Microarchitecture Exploration metrics
- CPU information
- GPU information
- Summary per basic analysis metrics
- Event summary
- Uncore Event summary

For some analysis types, the command-line summary report provides an issue description for metrics that exceed the predefined threshold. If you want to skip issues in the summary report, do one of the following:

- Use the `-report-knob show-issues=false` option when generating the report, for example: `vtune -report summary -r r001hpc -report-knob show-issues=false`
- Use the `-format=csv` option to view the report in the CSV format, for example: `vtune -report summary -r r001hpc -format=csv`

Example 3: Hardware Event-Based Sampling Hotspots Summary

This example generates the summary report for the r001hs Hotspots analysis (hardware event-based sampling mode) result on Windows* OS.

```
vtune -report summary -r r001hs
```

Elapsed Time: 3.986s

CPU Time: 1.391s

CPI Rate: 0.860

Wait Time: 65.023s

Inactive Time: 14.819s

Total Thread Count: 25

Paused Time: 0s

Hardware Events

Hardware Event Type Per Sample	Hardware Event Count	Hardware Event Sample Count	Events
-----------------------------------	----------------------	-----------------------------	--------

CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE 10000030	24,832,593		8
--	------------	--	---

CPU_CLK_UNHALTED.REF_TSC 24000000	3,471,208,416		120
--------------------------------------	---------------	--	-----

CPU_CLK_UNHALTED.REF_XCLK 10000030	43,877,874		14
---------------------------------------	------------	--	----

CPU_CLK_UNHALTED.THREAD 24000000	3,903,569,890		127
-------------------------------------	---------------	--	-----

FP_ARITH_INST_RETired.SCALAR_DOUBLE 20000030	943,046,424		14
---	-------------	--	----

INST_RETired.ANY 24000000	4,536,715,682		140
------------------------------	---------------	--	-----

UOPS_EXECUTED.THREAD 20000030	5,282,967,942		72
----------------------------------	---------------	--	----

UOPS_RETired.RETIRE_SLOTS 20000030	5,587,595,565		76
---------------------------------------	---------------	--	----

Collection and Platform Info

Application Command Line: C:\samples\tachyon\vc10\analyze_locks_Win32_Release
\analyze_locks.exe C:\samples\tachyon\dat\balls.dat

Operating System: Microsoft Windows 10

Computer Name: My Computer

Result Size: 13 MB

Collection start time: 12:12:52 24/07/2018 UTC

Collection stop time: 12:13:03 24/07/2018 UTC

Collector Type: Event-based sampling driver

CPU

Name: Intel(R) Processor code named Skylake ULT

Frequency: 2.496 GHz

Logical CPU Count: 4

Use the **Elapsed Time** metric as your performance baseline to estimate your optimizations.

Example 4: HPC Performance Characterization Summary

This command generates the summary report for the HPC Performance Characterization analysis result and skips issue descriptions:

```
vtune -report summary -r r001hpc -report-knob show-issues=false

Elapsed Time: 23.182s
GFLOPS: 14.748
Effective Physical Core Utilization: 58.0%
    Effective Logical Core Utilization: 13.920 Out of 24 logical CPUs
    Serial Time: 0.069s (0.3%)
    Parallel Region Time: 23.113s (99.7%)
        Estimated Ideal Time: 14.010s (60.4%)
        OpenMP Potential Gain: 9.103s (39.3%)
Memory Bound: 0.446
    Cache Bound: 0.175
    DRAM Bound: 0.216
    NUMA: % of Remote Accesses: 38.3%
FPU Utilization: 2.7%
    GFLOPS: 14.748
        Scalar GFLOPS: 4.801
        Packed GFLOPS: 9.947
Collection and Platform Info
    Application Command Line: ./sp.B.x
    User Name: vtune
    Operating System: 3.10.0-327.el7.x86_64 NAME="Red Hat Enterprise Linux Server" VERSION="7.2
(Maipo)" ID="rhel" ID_LIKE="fedora" VERSION_ID="7.2" P
    RETTY_NAME="Red Hat Enterprise Linux Server 7.2 (Maipo)" ANSI_COLOR="0;31" CPE_NAME="cpe://
o:redhat:enterprise_linux:7.2:GA:server" HOME_URL="https://w
ww.redhat.com/" BUG_REPORT_URL="https://bugzilla.redhat.com/" REDHAT_BUGZILLA_PRODUCT="Red Hat
Enterprise Linux 7" REDHAT_BUGZILLA_PRODUCT_VERSION=7.
2 REDHAT_SUPPORT_PRODUCT="Red Hat Enterprise Linux" REDHAT_SUPPORT_PRODUCT_VERSION="7.2"
    Computer Name: nntvtune235
    Result Size: 1 GB
    Collection start time: 19:04:30 13/06/2017 UTC
    Collection stop time: 19:04:53 13/06/2017 UTC
    Name: Intel(R) Xeon(R) E5/E7 v2 Processor code named Ivytown
    Frequency: 2.694 GHz
    Logical CPU Count: 24
    CPU
        Name: Intel(R) Xeon(R) E5/E7 v2 Processor code named Ivytown
        Frequency: 2.694 GHz
        Logical CPU Count: 24
```

Example 5: Memory Access Summary

This command generates the summary report for the Memory Access analysis result collected on Windows and shows issue descriptions:

```
vtune -report summary -r r001macc

Elapsed Time: 7.917s
CPU Time: 6.473s
Memory Bound: 21.9% of Pipeline Slots
| The metric value is high. This may indicate that a significant fraction
| of execution pipeline slots could be stalled due to demand memory load
| and stores. Explore the metric breakdown by memory hierarchy, memory
| bandwidth information, and correlation by memory objects.
```

```

| L1 Bound: 8.0% of Clockticks
|   This metric shows how often machine was stalled without missing the
|   L1 data cache. The L1 cache typically has the shortest latency.
|   However, in certain cases like loads blocked on older stores, a load
|   might suffer a high latency even though it is being satisfied by the
|   L1.
|
| L2 Bound: 3.0% of Clockticks
| L3 Bound: 5.0% of Clockticks
|   This metric shows how often CPU was stalled on L3 cache, or contended
|   with a sibling Core. Avoiding cache misses (L2 misses/L3 hits)
|   improves the latency and increases performance.
|
| DRAM Bound: 4.1% of Clockticks
|   DRAM Bandwidth Bound: 0.4% of Elapsed Time
|   Memory Latency: 0.000
Loads: 10,137,704,122
Stores: 3,208,896,264
LLC Miss Count: 1,750,105
Average Latency (cycles): 11
Total Thread Count: 21
Paused Time: 0s
System Bandwidth
  Max DRAM System Bandwidth: 15 GB

Bandwidth Utilization
Bandwidth Domain  Platform Maximum  Observed Maximum  Average Bandwidth  % of Elapsed Time with
High BW Utilization(%)
-----
-----
DRAM, GB/sec      15              11.300          0.4%
2.836
Collection and Platform Info
  Application Command Line: C:\samples\tachyon\vc10\analyze_locks_Win32_Release
\analyze_locks.exe "C:\samples\tachyon\dat\balls.dat"
  Operating System: Microsoft Windows 10
  Computer Name: My Computer
  Result Size: 31 MB
  Collection start time: 09:33:44 07/06/2017 UTC
  Collection stop time: 09:33:52 07/06/2017 UTC
  CPU
    Name: Intel(R) Processor code named Skylake ULT
    Frequency: 2.496 GHz
    Logical CPU Count: 4

```

The **Bandwidth Utilization** section in the summary report shows the following metrics:

- **Platform Maximum:** Expected maximum bandwidth for the system. This value can be automatically estimated using micro-benchmark at the start of analysis or hard-coded based on theoretical bandwidth limits.
- **Observed Maximum:** Maximum bandwidth observed during the analysis. If the value is close to the Platform Maximum, your workload is probably bandwidth-limited.
- **Average Bandwidth:** Average bandwidth utilization during the analysis.
- **% of Elapsed Time with High BW Utilization:** Percentage of Elapsed time spent heavily utilizing system bandwidth.

This information is provided for all kinds of bandwidth domains you have in the result (DRAM, MCDRAM, QPI, and so on).

See Also

[report action](#)
[no-summary action-option](#)
[Window: Summary](#)
in GUI

Hotspots Report

Use the `hotspots` command line report to identify program units (for example: functions, modules, or objects) that take the most processor time (Hotspots analysis), underutilize available CPUs or have long waits (Threading analysis), and so on.

Use the `hotspots` report to view hottest GPU computing tasks (or their instances) identified with the [gpu-hotspots](#) or [gpu-offload](#) analysis.

The report displays the hottest program units in the descending order by default, starting from the most performance-critical unit. The command-line reports provide the same data that is displayed in the default GUI analysis [viewpoint](#).

NOTE

To display a list of available groupings for a Hotspots report, enter `vtune -report hotspots -r <result_dir> group-by=?`. If you do not specify a result directory, the latest result is used by default.

Examples

Example 1: Hotspots Report with Module Grouping

This example opens the Hotspots report for the `r001hs` Hotspots analysis result and groups the data by module.

```
vtune -report hotspots -r r001hs -group-by module
```

Module	CPU Time
analyze_locks	10.080s
KERNELBASE	0.679s
ntdl	0.164s
...	

Example 2: Hotspots Report with Limited Items

This example displays the Hotspots report for the `r001hs` analysis result including only the top two functions with the highest CPU Time values. Functions having insignificant impact on performance are excluded from output.

```
vtune -report hotspots -r r001hs -limit 2
```

Function	CPU Time
grid_intersect	5.489s
sphere_intersect	3.590s

Example 3: Report per OpenCL Kernels

This example shows how to view the collected data per OpenCL kernels submitted and executed on the GPU:

```
vtune -report hotspots -group-by=computing-task -r r000gh
```

Computing Task	Work Size:Global	Computing Task:Total Time	Data Transferred:Size	EU
Array:Active(%)	L3 <-> GTI Total Bandwidth, GB/sec			
AdvancePaths	65536			
13.170s		25.0%		22.928
Init	65536			
0.006s		34.4%		45.802
Intersect	65536			
49.139s		61.5%		23.149
Sampler	65536			
6.525s		76.4%		11.745
InitFrameBuffer	362432			
0.000s		4.7%		17.456
clEnqueueReadBuffer			1.045s	3
GB	1.5%		8.840	

Example 4: Report Grouped per SYCL Task Instances

This example filters and groups the collected data by SYCL task instances:

```
vtune -report hotspots -group-by=computing-instance -r r000gh
```

Computing Task	Instance	Work Size:Global	Computing Task:Total Time	Data Transferred:Size	GPU Time
CopyVector2	2	6553600			
0.190s		0.190s			
clEnqueueReadBuffer	1				
0.034s		400 MB	0.034s		

See Also

[Summary Report](#)

[Filter and Group Command Line Reports](#)

Hardware Events Report

Intel® VTune™ Profiler counts the number of hardware events during the [Hardware Event-based Sampling Collection](#) to help you understand how the application utilizes available hardware resources. Use the hw-events report type to display hardware events count per application functions in the descending order by default.

Example

This example generates the hw-events report for the specified Hotspots analysis (hardware event-based sampling mode).

```
vtune -report hw-events -r r001hs
```

Function	Hardware Event Count:	Hardware Event Count:	Hardware Event
Count:	Hardware Event Count:	Module	
CPU_CLK_UNHALTED.REF_TSC (K)	INST_RETIREANY (K)	CPU_CLK_UNHALTED.THREAD (K)	
	BR_INST_RETIRENEARTAKEN (K)		

grid_intersect	11,901,341		
16,145,531	17,464,710	1,620,825	analyze_locks
sphere_intersect	7,944,651		
10,759,847	11,794,832	934,564	analyze_locks
grid_bounds_intersect	845,537		
1,190,025	1,299,113	86,424	analyze_locks
Gdiplus::Graphics::DrawImage	667,500		
1,255,001	1,246,747	47,194	analyze_locks
video::next_frame	241,619		
279,866	212,356	24,419	analyze_locks
pos2grid	195,869		
269,137	294,850	18,410	analyze_locks
tri_intersect	173,193		
271,435	296,786	14,919	analyze_locks
shader	172,992		
269,044	314,679	21,040	analyze_locks
Raypt	162,623		
206,349	204,826	26,100	analyze_locks
...			

See Also

[report action](#)

[Filter and Group Command Line Reports](#)
from command line

Callstacks Report

Intel® VTune™ Profiler collects call stack information during [User-Mode Sampling and Tracing Collection](#) or [Hardware Event-based Sampling Collection](#) with stack collection enabled. Use the `callstacks` report to see how the hot functions are called. This report type focuses on call sequences, beginning from the functions that take most CPU time.

You can use the `-column` option to filter the `callstacks` report and focus on the specific metric, for example:

```
vtune -report -callstacks -r r001ah -column="CPI Rate"
```

NOTE

To display a list of columns available for `callstacks` report, enter: `vtune -report callstacks -r <result_dir> column=?`

Examples

Example 1: Callstacks Report with Limited Items

The following example generates a `callstacks` report for the most recent analysis result and limits the number of functions and function stacks to 5 items.

```
vtune -report callstacks -limit 5
```

On Windows*:

Function Source File	Function Stack Start Address	CPU Time	Module	Function (Full)
grid_intersect		5.436s	analyze_locks.exe	grid_intersect
grid.cpp	0x40d340			

intersect.cpp	intersect_objects	1.918s	analyze_locks.exe	intersect_objects(struct ray *)
	0x402840			
shade.cpp	shader	0s	analyze_locks.exe	shader(struct ray *)
	0x404730			
trace_rest.cpp	trace	0s	analyze_locks.exe	trace(struct ray *)
	0x402370			
	render_one_pixel	0s	analyze_locks.exe	render_one_pixel
analyze_locks.cpp	0x401db0			
...				

On Linux*:

Function (Full)	Function Stack Source File	CPU Time	Module	Function
	Start Address			
initialize_2D_buffer		22.746s	tachyon_find_hotspots	
initialize_2D_buffer	find_hotspots.cpp	0x4018f0		
	render_one_pixel	22.746s	tachyon_find_hotspots	
render_one_pixel	find_hotspots.cpp	0x401950		
	draw_trace	0s	tachyon_find_hotspots	
draw_trace(void)	find_hotspots.cpp	0x401d70		
	thread_trace	0s	tachyon_find_hotspots	
thread_trace(thr_parms*)	find_hotspots.cpp	0x401ef0		
	trace_shm	0s	tachyon_find_hotspots	
trace_shm	trace_rest.cpp	0x410a20		
	trace_region	0s	tachyon_find_hotspots	
trace_region	trace_rest.cpp	0x410aa0		
	rt_renderscene	0s	tachyon_find_hotspots	
rt_renderscene(void*)	api.cpp	0x402360		
	tachyon_video	0s	tachyon_find_hotspots	
tachyon_video	video.cpp	0x402240		
main	main	0s	tachyon_find_hotspots	
	video.cpp	0x4013e0		
__libc_start_main	__libc_start_main	0s	libc.so.6	
	libc-start.c	0x21dd0		
_start	_start	0s	tachyon_find_hotspots	
	[Unknown]	0x40149c		
grid_intersect		7.282s	tachyon_find_hotspots	
grid_intersect	grid.cpp	0x408930		
	intersect_objects	2.756s	tachyon_find_hotspots	
intersect_objects(ray*)	intersect.cpp	0x40a400		
	shader	0s	tachyon_find_hotspots	
shader(ray*)	shade.cpp	0x40eae0		
...				

Example 2: Callstacks Report with Callstack Grouping

This example generates a callstacks report for the r001tr result that is grouped by function call stacks.

```
vtune -report callstacks -r r001tr -group-by callstack
```

On Windows*:

Function/Function Stack	Wait Time	Module	Function (Full)
tbb::internal::acquire_binsem_using_event	20.005s	tbb.dll	

```
tbb::internal::acquire_binsem_using_event

func@0x10003350           13.857s  gdiplus.dll      func@0x10003350
func@0x1000c1f0             0s    gdiplus.dll      func@0x1000c1f0
BaseThreadInitThunk         0s    KERNEL32.DLL   BaseThreadInitThunk
func@0x6b2dacf0             0s    ntdll.dll       func@0x6b2dacf0
func@0x6b2daccf             0s    ntdll.dll       func@0x6b2daccf

video::main_loop           10.111s  analyze_locks.exe video::main_loop(void)
main                         0s    analyze_locks.exe main
WinMain                      0s    analyze_locks.exe WinMain
_tmainCRTStartup            0s    analyze_locks.exe _tmainCRTStartup
[Unknown stack frame(s)]      0s    [Unknown]        [Unknown stack frame(s)]
BaseThreadInitThunk          0s    KERNEL32.DLL   BaseThreadInitThunk
func@0x6b2dacf0             0s    ntdll.dll       func@0x6b2dacf0
...

```

On Linux*:

Function/Function Stack (Full)	Wait Time	Module	Function
draw_task::operator() (tbb::blocked_range<int> const&) const	98.698s	tachyon_analyze_locks	draw_task::operator()
tbb::interface6::internal	0s	tachyon_analyze_locks	
tbb::interface6::internal	0s	tachyon_analyze_locks	
execute<tbb::interface6::internal	0s	tachyon_analyze_locks	
execute::interface6::internal	0s	tachyon_analyze_locks	
[TBB parallel_for on draw_task]	0s	tachyon_analyze_locks	
tbb::interface6::internal::execute(void)	0s	libtbb.so.2	
[TBB Dispatch Loop]	0s	libtbb.so.2	
tbb::internal::local_wait_for_all(tbb::task&, tbb::task*)			
...			

See Also

[report action](#)

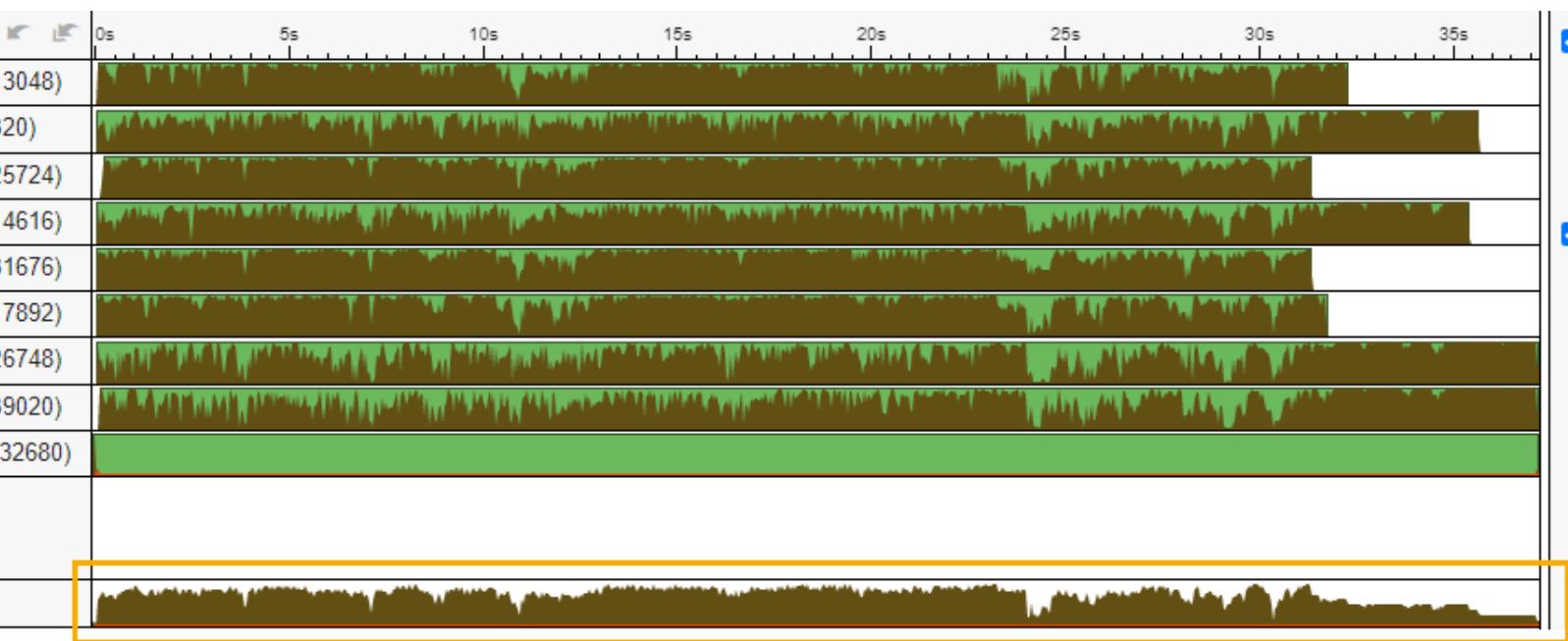
[Filter and Group Command Line Reports](#)

Timeline Report

Like the **Timeline** window in the Intel® VTune™ Profiler user interface, the Timeline report provides information about a metric which changed over time. This is a tabular report where each row displays the average metric value in each time interval.

For example, this command generates a timeline report that captures the **CPU Utilization** section highlighted in the Timeline pane below:

```
vtune -result-dir <result_path> -report timeline -report-knob column-by=CPUTime -report-knob bin-count=5
```



When you run the command, you see the following information:

```
vtune: Using result path `C:\VTune\Projects\sample (matrix)\r005hs'
vtune: Executing actions 75 % Generating a report
timeBin  Bin Start Time  Bin End Time  CPU Time:Self
-----  -----  -----  -----
0          0.000      7.440    621.445s
1          7.440     14.879    620.543s
2         14.879     22.319    675.243s
3         22.319     29.758    573.878s
4         29.758     37.198    404.154s
vtune: Executing actions 100 % done
```

Generate a Timeline Report from an Existing Result

Run this command:

```
vtune -result-dir <result_path>
      -report timeline
      -report-knob column-by=<metric name>
      [ -report-knob sort-column-by=<metric name> ]
      [ -report-knob group-by=<grouper name> ]
      [ -report-knob bin-count=<30 by default> ]
      [ -report-knob start=<0 by default> ]
      [ -report-knob end=<end of collection by default> ]
      [ -report-knob time-format=<seconds|milliseconds|events> ]
      [ -report-knob object-names=True|False by default ]
      [ -report-knob query-type=interval|... ]
```

where:

- <metric name> is the metric information you want. For example, CPUTime, OvertimeBandwidth, ContextSwitches, Task, PMUEventCount/PMUEventType etc.

- <grouper name> is an optional name given to a grouper. During data collection, every metric on the timeline gets detected within a context. The metric is then attributed to the context. You use this context as the <grouper name> to group metric actions. Some examples of <grouper name> are Thread, UncorePackage, Task etc. Each instance of the <grouper name> generates an output table for that collection of metric actions. For example, if a parallel application runs on eight threads, setting group-by=Thread generates eight timeline reports. To see the overall behavior of the metric over time, do not set <grouper name>.

Example: Timeline Report for CPU Time

This example generates a timeline report for the r000hs result of a Hotspots analysis. The timeline report shows CPU Time utilization over 30 intervals:

```
vtune.exe -r r000hs -report timeline -report-knob column-by=CPUTime
```

timeBin	Bin Start Time	Bin End Time	CPU Time:Self
0	0.000	1.240	5.456s
1	1.240	2.480	6.318s
2	2.480	3.720	6.462s
3	3.720	4.960	6.502s
4	4.960	6.200	6.638s
5	6.200	7.440	5.908s
6	7.440	8.679	6.659s
7	8.679	9.919	6.018s
8	9.919	11.159	5.555s
9	11.159	12.399	5.599s
10	12.399	13.639	6.404s
11	13.639	14.879	6.994s
12	14.879	16.119	6.997s
13	16.119	17.359	6.290s
14	17.359	18.599	6.986s
15	18.599	19.839	6.859s
16	19.839	21.079	6.496s
17	21.079	22.319	6.883s
18	22.319	23.558	7.044s
19	23.558	24.798	4.416s
20	24.798	26.038	5.306s
21	26.038	27.278	6.219s
22	27.278	28.518	6.140s
23	28.518	29.758	5.305s
24	29.758	30.998	5.874s
25	30.998	32.238	5.617s
26	32.238	33.478	3.909s
27	33.478	34.718	3.688s
28	34.718	35.958	3.232s
29	35.958	37.198	1.927s

Example: Timeline Report for CPU Time within Time Range

In this example, a timeline report is generated from the r006ue result of a Microarchitecture analysis. The data is collected between the second and fifth seconds. The data collected during these three seconds is presented over 50 intervals of time.

```
vtune -r r006ue -report timeline -report-knob column-by=CPUTime -report-knob start=200000000000 -report-knob end=500000000000 -report-knob bin-count=50
```

timeBin	Bin Start Time	Bin End Time	CPU Time:Self
0	2.000	2.060	583.614s
1	2.060	2.120	596.239s
2	2.120	2.180	568.513s
3	2.180	2.240	656.714s
4	2.240	2.300	593.027s
5	2.300	2.360	582.537s
6	2.360	2.420	686.536s
7	2.420	2.480	630.049s
8	2.480	2.540	683.360s
9	2.540	2.600	449.970s
10	2.600	2.660	534.815s
11	2.660	2.720	523.006s
12	2.720	2.780	563.003s
13	2.780	2.840	650.275s
14	2.840	2.900	590.479s
15	2.900	2.960	644.241s
16	2.960	3.020	646.289s
17	3.020	3.080	644.978s
18	3.080	3.140	634.378s
19	3.140	3.200	627.582s
20	3.200	3.260	588.956s
21	3.260	3.320	621.873s
22	3.320	3.380	158.051s
23	3.380	3.440	170.440s
24	3.440	3.500	216.458s
25	3.500	3.560	121.819s
26	3.560	3.620	351.148s
27	3.620	3.680	256.142s
28	3.680	3.740	385.892s
29	3.740	3.800	507.566s
30	3.800	3.860	459.971s
31	3.860	3.920	495.019s
32	3.920	3.980	503.530s
33	3.980	4.040	565.219s
34	4.040	4.100	526.778s
35	4.100	4.160	541.870s
36	4.160	4.220	569.609s
37	4.220	4.280	474.287s
38	4.280	4.340	585.829s
39	4.340	4.400	625.578s
40	4.400	4.460	656.474s
41	4.460	4.520	438.410s
42	4.520	4.580	519.766s
43	4.580	4.640	414.919s
44	4.640	4.700	577.235s
45	4.700	4.760	596.569s
46	4.760	4.820	570.871s

47	4.820	4.880	586.414s
48	4.880	4.940	532.267s
49	4.940	5.000	564.387s

Example: Timeline Report as a CSV File

When you collect a significantly large volume of data, consider exporting the timeline report to a CSV file for easier data management.

In this example, a timeline report generated from the r008hs result (of a Hotspots analysis) is saved as a CSV file (r008hs_timeline.csv). The collected data is split into 1000 intervals of time.

```
vtune -r r008hs -report timeline -report-knob column-by=CPUTime -report-knob bin-count=1000 -
format=csv -csv-delimiter=semicolon -report-output r008hs_timeline.csv
```

The contents of r008hs_timeline.csv contain:

```
timeBin;Bin Start Time;Bin End Time;CPU Time:Self
0;2.000;2.060;583.614
1;2.060;2.120;596.239
...
```

Example: Timeline Report of PMU Core Events Grouped by Threads

Run this command to generate a timeline report that groups PMU core events by threads and sorts the groups by event counts.

```
vtune -r <result dir> -report timeline -report-knob group-by=Thread -report-knob sort-column-
by=PMUEventCount -report-knob column-by=PMUEventCount/PMUEventType
```

Example: Timeline Report of PMU Uncore Events Grouped by Packages

Run this command to generate a timeline report that groups PMU uncore events by packages and sorts the groups by uncore event counts.

```
vtune -r <result dir> -report timeline -report-knob group-by=UncorePackage -report-knob sort-
column-by=UncoreEventCount -report-knob column-by=UncoreEventCount/UncoreEventType
```

Example: Timeline Report of Context Switches per Thread

Run this command to generate a timeline report of context switches per thread, using a time format of millisecond

```
vtune -r <result dir> -report timeline -report-knob query-type=interval -report-knob group-
by=Thread -report-knob sort-column-by=ContextSwitchCount -report-knob column-by=ContextSwitches -
report-knob bin-count=1000000000 -report-knob time-format=millisecond
```

Example: Timeline Report of Tasks per Thread

Run this command to generate a timeline report of tasks per thread and also displays the names of the tasks.

```
vtune -r <result dir> -report timeline -report-knob query-type=interval -report-knob group-
by=Thread -report-knob sort-column-by=TaskTime -report-knob column-by=Task -report-knob object-
names=True -report-knob bin-count=1000000000
```

See Also

[Timeline pane](#)

[Window: Top-down Tree](#)

Top-down Report

Similar to the **Top-down** window, available in GUI, the Top-down represents call sequences (stacks) detected during collection phase starting from the application root. Use the top-down report to explore the call sequence flow of the application and analyze the time spent in each program unit and on its callees.

NOTE

Intel® VTune™ Profiler collects information about program unit callees only during [User-Mode Sampling and Tracing Collection](#) or [Hardware Event-based Sampling Collection](#) with stack collection enabled.

Examples

Example 1: Hotspots Top-down Report

This example displays the report for the specified Hotspots analysis in the user-mode sampling mode with functions stacks limited to 5 elements.

```
vtune -report top-down -r r001hs -limit 5
```

Function Stack	CPU Time:Total	CPU Time:Effective	CPU Time:Total	CPU Time:Spin	CPU Time:Total
CPU Time:Overhead	Time:Total				
Total	100.000%		100.000%		
100.000%		100.000%		99.835%	
func@0x6b2daccf	99.853%		99.835%		
100.000%		100.000%		99.835%	
func@0x6b2dacf0	99.853%		99.835%		
100.000%		100.000%		99.835%	
BaseThreadInitThunk	99.853%		99.835%		
100.000%		100.000%		97.876%	
thread_video	95.614%		97.876%		
78.195%		0.0%			

Example 2: Hotspots Report with Enabled Call Stack Collection (Linux*)

This command runs the Hotspots analysis in the hardware event-based sampling mode with enabled call stack collection.

```
vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -- /home/tachyon
```

The following command generates the top-down report for the previously collected result and shows the result for columns with the `time:total` strings in the title.

```
vtune -report top-down -r r001hs -column=time:total
```

Function Stack	CPU Time:	CPU Time:	CPU Time:	Context Switch Time:
Context Switch Time:	Context Switch Time:			
Wait Time:Total	Total	Effective	Time:Total	Spin Time:Total
	Inactive	Time:Total		Total
Total	100.000%		100.000%	100.000%
100.000%		100.000%	100.000%	
func@0x6b2daccf	97.595%		97.704%	89.202%
65.777%	90.121%		62.893%	
func@0x6b2dacf0	97.595%		97.704%	89.202%

65.777%	90.121%	62.893%	
BaseThreadInitThunk	97.595%	97.704%	89.202%
65.777%	90.121%	62.893%	
threadstartex	67.091%	67.855%	8.335%
29.825%	9.027%	32.289%	
...			

Example 3: Hotspots Report with Disabled Stack Collection (Windows*)

This command runs the Hotspots analysis in the hardware event-based sampling mode with disabled call stack collection.

```
vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=false -- C:\tachyon\tachyon.exe
```

This command generates the top-down report for the previously collected result, and shows the result for columns with the `time:total` string in the title. The report does not include information about program unit callees, as it was not collected during the analysis.

```
vtune -report top-down -r r001hs -column=time:total
```

Function Stack	CPU Time:Total	CPU Time:Effective	Time:Total	CPU Time:Spin	Time:Total
Total	100.000%		100.000%		100.000%
grid_intersect	50.172%		50.213%		0.0%
sphere_intersect	31.740%		31.766%		0.0%
grid_bounds_intersect	3.766%		3.769%		0.0%
pos2grid	0.778%		0.778%		0.0%
...					

See Also

[report action](#)
[Window: Top-down Tree](#)

GPU Compute/Media Hotspots Report

When you have results from multiple runs of the [GPU Compute/Media Hotspots analysis](#), you can merge these results to get a consolidated view of the metrics.

To generate this consolidated report, use the `vtune-results-merger.py` script. You can find this script in the `bin64` folder where you installed Intel® VTune™ Profiler on your system.

Requirements:

- You must use analysis results for the same application.
- All analysis results must be collected on the same machine.
- The same GPU (or set of GPUs) must be used for the analysis runs.
- This script requires a minimum of two results.

Syntax

```
vtune-results-merger.py [-h] [-format {csv,text}] [-csv-delimiter {comma,semicolon,colon,tab}] [-report-output REPORT_OUTPUT] first_result-dir result_dir[result_dir ...]
```

Required Arguments:

- `first_result_dir` : This is the location of the first result

- `result_dir` : This is the location of the second or additional result. Make sure to save a single result in each folder. Specify the path for each additional result.

Optional Arguments:

- **-h, --help** : Display help options
 - **-format {csv,text}** : Specify an output format (.CSV|.TXT) for the report
 - **-csv-delimiter {comma,semicolon,colon,tab}** : Specify a delimiter character for the CSV output
 - **-report-output REPORT_OUTPUT** : Write the output of the report to a file

gprof-cc Report

You can use the Intel® VTune™ Profiler command line interface to display analysis results in gprof-like format. The `gprof-cc` report shows how much time is spent in each program unit, its callers and callees. The report is sorted by time spent in the function and its callees.

Example

This example generates a gprof-cc report from the r001hs hotspots result.

The empty lines divide the report into entries, one for each function. The first line of the entry shows the caller of the function, the second line shows the called function, and the following lines show function callees. The Index by function name portion of the report shows the function index sorted by function name.

```

vtune -report gprof-cc -r r001hs

Index % CPU Time:Total CPU Time:Self CPU Time:Children Name Index
----- ----- -----
func@0x6b2dacf0 0.0 11.319 [3]
[1] 100.0 0.0 11.319 [1]
BaseThreadInitThunk 0.030 0.0 [36]
GetSphere 0.0 0.554 [23]
_tmainCRTStartup 0.0 0.016 [44]
func@0x1000c1f0 0.0 10.709 [10]
thread_video 0.0 0.010 [49]
threadstartex 0.0 <spontaneous>
[2] 100.0 0.0 11.319 [2]
func@0x6b2daccf 0.0 11.319 [3]
func@0x6b2dacf0 0.0 11.319 [2]
func@0x6b2daccf 0.0 11.319 [3]
[3] 100.0 0.0 11.319 [3]
func@0x6b2dacf0 0.0 11.319 [1]
BaseThreadInitThunk 0.0 10.709 [9]
thread_trace 0.0 10.709 [TBB parallel for on class
[4] 94.61 0.0

```

```

draw_task] [4]           0.0      10.709
draw_task::operator()    [5]
                        0.0      10.709      [TBB parallel_for on class
draw_task] [4]
[5]   94.61            0.0      10.709
draw_task::operator()    [5]
                        0.436      0.0
video::next_frame       [26]
                        0.020      10.234
render_one_pixel        [13]
                        0.018      0.0
drawing_area::~drawing_area [42]
...
Index by function name

Index  Function
-----
[96]  ColorAccum
[30]  ColorAddS
[15]  ColorScale
[137] CreateCompatibleBitmap
[138] DeleteObject
[211] EngAcquireSemaphore
[139] EngCopyBits
[212] EtwEventRegister
[45]  ExAcquirePushLockExclusiveEx
[35]  ExAcquireResourceExclusiveLite
...

```

Difference Report

Comparing two results from the command line is a quick way to check for your application regressions. Use the following syntax to create the difference report for the specified analysis results:

```
vtune -report <report_name> -r <result1_path> -r <result2_path>
```

where

- <report_name> is the type of report for comparison
- <result1_path> is a directory where your first result file is located
- <result2_path> is a directory where your second result file is located

Example

This example compares r001hs and r002hs Hotspots analysis results collected on Linux and displays CPU time difference for each function of the analyzed application. In the result for the optimized application (r002hs), a new main function is running for 0.010 seconds, while the Hotspot function algorithm_2 is optimized by 1.678 seconds.

```
vtune -report hotspots -r r001hs -r r002hs
```

Function	Module	Result 1:CPU Time	Result 2:CPU Time	Difference:CPU Time
algorithm_1	matrix	1.225	1.222	0.003
algorithm_2	matrix	3.280	1.602	1.678
main	matrix	0	0.010	-0.010

Generate a Difference Report for Regression Testing

Use the `vtune` command to test your code for regressions on a daily basis:

1. Create a baseline.

- Run the `vtune` tool to analyze your target using a particular analysis type. For example:

On Linux*:

```
vtune -collect hotspots -- sample
```

On Windows*:

```
vtune -collect hotspots -- sample.exe
```

The command runs a Hotspots analysis on the `sample` or `sample.exe` target and writes the result to the current working directory. A Summary report is written to `stdout`.

- Generate a report to use as a baseline for further analysis. For example:

```
vtune -report hotspots -result-dir r001hs
```

This creates a Hotspots report that shows the CPU time for each function of the `sample` or `sample.exe` target.

2. Update your source code to optimize the target application.

3. Create and run the script that:

- On Linux: Sets the path to the `vtune` installation folder
- On Windows: Invokes `sep-vars.cmd` in the Intel® VTune™ Profiler installation folder to set up the environment.
- Starts the `vtune` command to collect performance data.
- Runs the `vtune` command to compare the current result with the initial baseline result and displays the difference. For example:

```
vtune -R hotspots -r r001hs -r r002hs
```

This example compares CPU time for each function in results `r001hs` and `r002hs` and displays both results side-by-side with the calculated difference. The positive difference between the performance values indicates an improvement for result 2. The negative difference indicates a regression.

NOTE

You can compare results of the same analysis type or performance metrics only.

4. The test is passed if no regressions found.

5. Repeat steps 2-4 on a regular basis.

Installation Information

Whether you downloaded Intel® VTune™ Profiler as a standalone component or with the Intel® oneAPI Base Toolkit, the default path for your `<install-dir>` is:

Operating System	Path to <code><install-dir></code>
Windows* OS	<ul style="list-style-type: none"> <code>C:\Program Files (x86)\Intel\oneAPI\</code> <code>C:\Program Files\Intel\oneAPI\</code> (in certain systems)
Linux* OS	<ul style="list-style-type: none"> <code>/opt/intel/oneapi/</code> for root users <code>\$HOME/intel/oneapi/</code> for non-root users

Operating System	Path to <install-dir>
macOS*	/opt/intel/oneapi/

For OS-specific installation instructions, refer to the [VTune Profiler Installation Guide](#).

See Also

[vtune Command Syntax](#)

[Filter and Group Command Line Reports](#)

View Source Objects from Command Line

For better understanding of a performance problem, it is important to associate a hotspot with the source code and exact machine instruction(s) that caused this hotspot. To do this, you can open the source/assembly code directly from the command line. Use the `-source-object` option to switch a report to the source or assembly view mode, including associated performance data. Here is the command syntax for viewing source objects in the command line:

```
vtune -report <report_name> -source-object <object_type>[=><value>] -result-dir
<result_path>
```

where

- `report_name` is the specified report type (hotspots or hw-events)
- `object_type` is the object type name. Possible values are: module, source-file, function.
- `value` is the application unit name for which source or assembly data should be displayed
- `result_dir` is a directory where your result file is located

Examples

Example 1: Report Displaying Source Data

This example generates a hotspots report that displays source data for the `grid_intersect` function. The report is filtered to display only data columns with `source`, `instructions`, `cpi` values in the title. Since the result directory is not specified, the most recent hotspots analysis result is used.

```
vtune -report hotspots -source-object function=grid_intersect -column=source,instructions,cpi
```

Source	Line	Source	Instructions	Retired	CPI	Rate
461		}				
462						
463						
464		/* the real thing */				
465		static void grid_intersect(grid * g, ray * ry)				
466		{				
467		1.301				
468						
469		flt tnear, tfar, offset;				
470		vector curpos, tmax, tdelta, pdeltaX, pdeltaY, pdeltaZ, nXp, nYp, nZp;				
471		gridindex curvox, step, out;				
472		int voxindex;				
473		objectlist * cur;				
474						48,867,664

```

475             if (ry->flags &
RT_RAY_FINISHED)
...

```

Example 2: Report with Grouped Assembly Data

This example generates a hardware events report that displays assembly data grouped by basic block and then address. The report is filtered to display only data columns with block, source, function, instructions, assembly, cpi, address values in the title.

```
vtune -report hotspots -r /home/results/r002hs -source-object function=grid_intersect -group-by=block,address -column=block,source,function,instructions,assembly,cpi,address
```

Basic Block	Instructions Retired	CPI	Rate	Assembly	
Source Line	Function (Full)	Source File	Function Range	Size	Start Address
0x40d340	39,900,000	2.238	Block 1		
[Unknown]	[Unknown]	[Unknown]		0	
0x40d340	3,800,000	2.000	sub esp, 0xd8		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d346	0	mov eax, dword ptr [0x4130e0]			
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d34b	7,600,000	0.750	xor eax, esp		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d34d	3,800,000	4.500	mov dword ptr [esp+0xd4], eax		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d354	5,700,000	0.333	push esi		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d355	1,900,000	1.000	mov esi, dword ptr [esp+0xe4]		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d35c	1,900,000	10.000	push edi		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d35d	3,800,000	0.500	mov edi, dword ptr [esp+0xe4]		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d364	1,900,000	2.000	mov dword ptr [esp+0x74], edi		
466	grid_intersect	grid.cpp	0x646		0x40d340
0x40d368	3,800,000	3.500	test byte ptr [esi+0x8], 0x8		
475	grid_intersect	grid.cpp	0x646		0x40d340
0x40d36c	5,700,000	0.667	jnz 0x40d96f <Block 64>		
475	grid_intersect	grid.cpp	0x646		0x40d340
0x40d372	9,500,000	3.800	Block 2		
[Unknown]	[Unknown]	[Unknown]		0	
0x40d372	0	0.000	lea eax, ptr [esp+0x50]		
478	grid_intersect	grid.cpp	0x646		0x40d340
0x40d376		push eax			
478	[Unknown]	[Unknown]	[Unknown]	[Unknown]	
0x40d377	1,900,000	11.000	lea eax, ptr [esp+0x8c]		
478	grid_intersect	grid.cpp	0x646		0x40d340
0x40d37e	1,900,000	0.000	push eax		
478	grid_intersect	grid.cpp	0x646		0x40d340
0x40d37f	3,800,000	1.000	push esi		
478	grid_intersect	grid.cpp	0x646		0x40d340
0x40d380	0	push edi			
478	grid_intersect	grid.cpp	0x646		0x40d340
0x40d381	1,900,000	1.000	call 0x40e4a0 <grid_bounds_intersect>		
478	grid_intersect	grid.cpp	0x646		0x40d340
0x40d386	15,200,000	2.375	Block 3		
[Unknown]	[Unknown]	[Unknown]	[Unknown]	0	

```

0x40d386      13,300,000  2.286 add esp, 0x10
478      grid_intersect  grid.cpp    0x646          0x40d340
0x40d389      1,900,000   3.000 test eax, eax
478      grid_intersect  grid.cpp    0x646          0x40d340
0x40d38b          0           jz 0x40d96f <Block 64>
478      [Unknown]     [Unknown]  [Unknown]        [Unknown]
0x40d391      3,800,000   2.000 Block 4
[Unknown]     [Unknown]     [Unknown]        0
0x40d391          0           0.000 movsd xmm0, qword ptr [esp+0x88]
481      grid_intersect  grid.cpp    0x646          0x40d340
0x40d39a      3,800,000   1.000 comisd xmm0, qword ptr [esi+0x48]
481      grid_intersect  grid.cpp    0x646          0x40d340
0x40d39f          0           jnbe 0x40d96f <Block 64>
481      grid_intersect  grid.cpp    0x646          0x40d340
0x40d3a5      5,700,000   2.000 Block 5
[Unknown]     [Unknown]     [Unknown]        0
0x40d3a5      1,900,000   1.000 sub esp, 0x8
484      grid_intersect  grid.cpp    0x646          0x40d340
0x40d3a8      1,900,000   1.000 lea eax, ptr [esp+0x10]
484      grid_intersect  grid.cpp    0x646          0x40d340

```

See Also

[report action](#)

[source-object action-option](#)

[Filter and Group Command Line Reports](#)

Save and Format Command Line Reports

By default, a report is written to `stdout` in text format, but `vtune` provides several options to control the report format:

- [Save a Report to a File](#)
- [Limit Line Width](#)

Save a Report to a File

When generating a report from the command line, use the `report-output` option to save this report in the specified format. By default, most types of reports are saved in text format, but you may also choose CSV. Whichever file type you choose, a number of options are available so you can format your report.

Here is the basic command syntax:

```
vtune -report <report_type> -result-dir <dir> -report-output <path/filename.ext>
```

where:

- `<report_name>` is the type of report to create.
- `<dir>` is the location of the result directory.
- `<path/filename.ext>` is the PATH, filename and file extension of the report file to be created.

NOTE

To be sure the correct result is used, use the `result-dir` option to specify the result directory. If not specified when generating a report, the report uses the highest numbered compatible result in the current working directory.

Examples:

- Generate a Hotspots report from the r001hs result on Linux*, and save it to /home/test/MyReport.txt in text format.

```
vtune -report hotspots -result-dir r001hs -report-output /home/test/MyReport.txt
```

- Generate a hotspots report in the CSV format from the most recent result and save it in the current Linux working directory. Use the [format](#) option with the `csv` argument and the `csv-delimiter` option to specify a delimiter, such as `comma`.

```
vtune -R hotspots -report-output MyReport.csv -format csv -csv-delimiter comma
```

```
Module,Process,CPU Time
worker3.so,main,10.735worker1.so,main,5.525worker2.so,main,3.612worker5.so,main,3.103worker4.so,main,1.679main,main,0.064
```

- Generate a vtune report with UNC events. Group results by package for this purpose.

```
vtune -report hw-events -group-by package -r unc
```

Limit Line Width

To limit line width for readability, use the [report-width](#) option and specify the maximum number of characters per line before wrapping occurs.

Example:

Output a Hotspots report from the most recent result as a text file with a maximum width of 60 characters per line.

```
vtune -report hotspots -report-width 60 -report-output MyHotspotsReport.txt
```

See Also

[report](#)
action

[Filter and Group Command Line Reports](#)

Filter and Group Command Line Reports

You can manage `vtune` reports from command line by using the following options:

- [Group data by a granularity level](#)
- [Sort the data in the ascending or descending order by metric](#)
- [Filter a report by:](#)
 - [program unit name](#)
 - [system functions in the call stack](#)
 - [column name](#)
 - [time interval](#)

Group Report Data

Use the [group-by](#) option to group data in your report by some value, such as function. For multiple grouping levels, add arguments separated by commas (no spaces). Grouping columns show up first in the view.

NOTE

To display a list of available groupings for a particular report, use `-help report <report_name>`.

Examples:

- Write stack information for all functions in the threading analysis result r001tr and group data by call stack:

```
vtune -report callstacks -r r001tr -group-by callstack
```

- Generate a hotspots report grouping data in this order: **Process, Process ID, Module, and Function**:

```
vtune -report hotspots -r r002hs -group-by process,process-id,module,function
```

Sort Report Data

There are a pair of options that you can use to sort report data: `sort-asc` and `sort-desc`. Use the `sort-asc` action-option to organize a report in ascending order of the specified field(s), or use `sort-desc` to sort it in descending order. You can specify up to three different fields.

Example:

Generate a report from the Microarchitecture Exploration r001ue result, and sort data in ascending order by the event columns `INST_RETIREANY` and `CPU_CLK_UNHALTED.CORE`.

```
vtune -report hw-events -r r001ue -sort-asc INST_RETIREANY,CPU_CLK_UNHALTED.CORE
```

Filter Reports by Program Unit

You can narrow down your report to display performance data for a particular program unit by using this option:

```
filter <program_unit> [= | !=] <name>
```

where:

- `<program_unit>` is one of the following values: basic-block, frame, function, function-sync-obj, module, process, source-file, source-line, sync-obj, task, thread, computing-task, computing-instance
- `<= | !=>` are the operators 'equal to' (include) or 'not equal to' (exclude or filter out)
- `<value>` is the value to include or exclude

NOTE

- To display a list of available filters for a particular report, use `-report <report_name> -result <result_dir> -filter=? .`
 - To specify multiple filter items, use multiple `-filter` option attributes. Multiple values for the same column are combined with 'OR'. Values for different columns are combined with 'AND'.
-

Examples:

- Display a Hotspots report on the most recent result in the current working directory, but only include data on the `sample` module:

```
vtune -report hotspots -filter module=sample
```

- Include data from both `sample.dll` and `sample2.dll` modules, excluding all other modules:

```
vtune -report hotspots -filter module=sample.dll -filter module=sample2.dll
```

- Display a Hotspots report that includes data for all processes except `app`:

```
vtune -report hotspots -filter process!=app
```

Filter by Call Stack Mode

You can filter the report by call stack display mode to set whether system functions display in the call stack data in your report `call-stack-mode`

Possible values: all, user-only, user-plus-one.

Example:

Generate a Hotspots report from the most recent compatible result, group the result data by function, and only display user functions and system functions called directly from user functions:

```
vtune -report hotspots -group-by function -call-stack-mode user-plus-one
```

Filter by Column Name

To display only particular columns providing [Reference](#)/event data, use the `column` option and specify a full name of the required column(s) or its substring.

NOTE

To display a list of columns available for a particular report, type: `vtune -report <report_name> -r <result_dir> column=?`

Examples:

- Show grouping and data columns only for event columns with the `*INST_RETIRED.*` string in the title:

```
vtune -R hw-events -r r000hs --column=INST_RETIRED.
```

- Show grouping and data columns only for columns with the `Idle` and `Spin` strings in the title:

```
vtune -R hotspots -r r001hs --column=Idle,Spin
```

Filter by Time Interval

To view data for a specific time range only, use the `time-filter <begin_time>:<end_time>` option, where:

- `<begin_time>` is the elapsed time in seconds for the start of the included time range. If unspecified, the time range begins at zero.
- `<end_time>` is the elapsed time in seconds for the end of the included time range. If unspecified, the time range ends at total elapsed time.

Examples:

- Generate a Hotspots report from the `r001tr` result, grouped by the value in the function column. For the `time-filter`, the start of the range is specified as `1.25`, and the end of the range is left unbounded, so the report includes data starting from `1.25` seconds of elapsed time to the time when analysis completes:

```
vtune -report hotspots -result-dir r001tr -group-by function -time-filter 1.25:
```

- Generate a Hotspots report from the `r001tr` result, grouped by the value in the function column. For the `time-filter`, the start of the range is not specified, and the end of the range is `5.0`, so the report includes data from the start of the analysis data to `5.0` seconds of elapsed time:

```
vtune -report hotspots -result-dir r001tr -group-by function -time-filter :5.0
```

- Generate a report for both start and end values of the range specified, so the report includes data from `1.25` second to `5.0` seconds of elapsed time:

```
vtune -report hotspots -result-dir r001tr -group-by function -time-filter 1.25:5.0
```

[Save and Format Command Line Reports](#)

[Manage Data Views](#)

[group-by](#)

action-option

```

sort-asc
  action-option
sort-desc
  action-option
filter
  action-option
column
  action-option

```

Command Line Usage Scenarios

This section describes the following Intel® VTune™ Profiler command line usage scenarios:

- [Use VTune Profiler Server in Containers](#)
- [Android* Target Analysis from the Command Line](#)
- [OpenMP* Analysis from the Command Line](#)
- [Java* Code Analysis from the Command Line](#)

Use VTune Profiler Server in Containers

Intel® VTune™ Profiler Server offers additional command line interface options that help make its usage in containerized environments more convenient.

These command line options are designed to trigger certain actions inside VTune Profiler Server in order to make it more convenient to run VTune Profiler Server inside a container.

All of these options apply to the `vtune-backend` binary.

Custom Base URL

You can use the `--base-url` option to request a custom base URL to access VTune Profiler Server. This option can be useful when a static port is required for VTune Profiler Server access while running VTune Profiler Server inside a Docker* container.

Format:

```
--base-url=http(s)://<host>:<port>/<pathname>/
```

Usage Example:

1. Enable SSH port forwarding on the host:

```
ssh -L 127.0.0.1:3000:127.0.0.1:8080 <ssh host name>
```

2. Run VTune Profiler Server with custom URL and port:

```
vtune-backend --web-port=8080 --base-url=https://127.0.0.1:3000
```

VTune Profiler Server prints out a URL in the format `https://127.0.0.1:3000?one-time-token=<token>`, which clients can use to access the server.

Usage Statistics Collection

Allow or decline the collection of usage statistics for the Intel® Software Improvement Program from the command line.

Use This	To Do This
<code>--usage-statistics-opt-in</code>	Allow the collection of usage statistics
<code>--usage-statistics-opt-out</code>	Do not allow the collection of usage statistics
<code>--print-usage-statistics-agreement</code>	Print agreement text for the Intel Software Improvement Program

Suppress Automatic Help Tours

VTune Profiler automatically activates the interactive help tour the first time VTune Profiler is started.

Use the `--suppress-automatic-help-tours` option to prevent VTune Profiler from showing help tours on first start.

See Also

[Install VTune Profiler Server](#) Set up Intel® VTune™ Profiler as a web server, using a lightweight deployment intended for personal use or a full-scale corporate deployment supporting multi-user environment.

[Web Server Interface](#) Use Intel® VTune™ Profiler in a web server mode to get an easy on-boarding experience, benefit from a collaborative multi-user environment, and access a common repository of collected performance results.

[Cookbook: Using VTune Profiler Server in HPC Clusters](#)

Android* Target Analysis from the Command Line

Use the Intel® VTune™ Profiler to collect data on a remote Android application from the host system (remote usage mode) via command line interface (`vtune`) and view the analysis result locally from the command line or GUI.

You may run the following analysis types on Android systems:

- [Hotspots analysis](#) (user-mode sampling mode)
- [Hardware event-based sampling analysis types](#)
- [Custom analysis](#)

Configure and Run Performance Analysis on Android System

Remote data collection using the `vtune` command running on the host is very similar to the native collection on the target except that the `target-system` option is added to the command line.

Prerequisites: Make sure to prepare a target Android* system and your application for analysis.

To run an analysis on an Android device:

1. Launch your application on the target device.
2. Find out <pid> or <name> of the application running on remote Android system. For example, you can use adb shell ps command for the purpose:

```
adb shell ps
...
root    2956  2      0      c1263c67 00000000 S kworker/u:3
u0_a34  8485  174   770232 54260 ffffffff 00000000 R com.intel.tbb.example.tachyon
shell   8502  235  2148   1028  00000000 b76bcf46 R ps
...
```

3. **Optional:** If you have several Android devices, you may set the `ANDROID_SERIAL` environment variable to specify the device you plan to use for analysis. For example:

```
export ANDROID_SERIAL= emulator-5554 or export ANDROID_SERIAL=10.23.235.47:5555
```

4. On the development host, run `vtune` to collect data.

By default, the `vtune` utility is located in the following directory:

- On Windows*: <install-dir>\bin{32,64}
- On Linux*: <install-dir>/bin{32,64}

Use the following syntax to run an analysis:

```
host>./vtune -target-system=android:<deviceName> -<action> <analysis_type> [-duration <duration_value>] [-r <result_path>] [-search-dir=<search_dir>] [-source-search-dir=<source_search_dir>] - <target_application>
```

where:

- *<deviceName>* is the name of your Android device, for example: Medfield2B3E703C . If you do not specify the name of the device, the VTune Profiler uses the default device specified with `adb`. You do not need to specify the device name if you set the `ANDROID_SERIAL` environment variable before the collection.
- *<action>* is the action to perform the analysis (`collect` or `collect-with`)
- *<analysis_type>* is a predefined analysis type, such as `hotspots`, `uarch-exploration`, and so on
- *<duration_value>* is the duration in seconds
- *<result_path>* is a PATH/name of the directory where a result is stored
- *<search_dir>* is a path to search for binary files used by your Android application
- *<source_search_dir>* is a path to search for source files used by your Android application
- *<target_application>* is an application to analyze. The command option depends on analysis target type:
 - To specify an application (a native Linux* application running on Android) or a script to analyze, enter the path to the application or the script on your host system.

NOTE

This target type is not supported for the Hotspots analysis of Android applications.

- To specify an Android application package to analyze, enter the name of the Android package installed on a remote device.
- To specify a particular process to attach to and analyze, use the `-target-process` command to specify application by process name or the `-target-pid` command to specify the application by process PID.
- To profile your Android system, do not specify target application.

NOTE

System-wide analysis is possible on rooted devices only.

5. **Optional:** You can send `pause` and `resume` commands during collection from another console window, for example:

```
host>./vtune -C pause -r tachyon_r001
```

```
host>./vtune -C resume -r tachyon_r001
```

6. If you do not specify analysis duration, you can stop analysis by pressing **Ctrl-C** or sending the `stop` command from another console on the host development system:

```
vtune -r tachyon_r001 -C stop
```

NOTE

You may use the **Command Line...** option in the VTune Profiler graphical interface to automatically generate a command line for an analysis configuration selected in the GUI.

Hotspots Analysis (User-Mode Sampling)

In this mode, you can:

- Run analysis *without* root access (although, root access is required for Java* analysis)
- Run the Hotspots analysis (if a target process or PID is specified) to identify functions that take the most time to execute (*hotspots*)
- Explore call stacks
- View C/C++ generated functions/source
- (**If installed**) Automatically obtain Java function names for functions that have been JITed and drill down to either JIT assembly or Java source or DEX Byte Code

NOTE

Java analysis is not supported for the 4th Generation Intel® Core™ processors (based on Intel microarchitecture code name Haswell).

Example

This example runs Hotspots analysis on target Android system.

```
host>./vtune -collect hotspots -target-system=android -r tachyon_r@@@ --
com.intel.tbb.example.tachyon
```

Event-Based Sampling Analysis

In this mode, you can:

- Use hardware event-based sampling analysis types with event groups predefined by Intel architects
- View C/C++ generated functions/source
- Explore call stacks (if a target process or PID is specified)
- Analyze performance system wide (if call stack analysis is disabled)
- (**If installed**) Automatically obtain Java function names for functions that have been JITed and drill down to either JIT assembly or Java source or DEX Byte Code

NOTE

Java analysis is not supported for the 4th Generation Intel® Core™ processors (based on Intel microarchitecture code name Haswell) or systems using ART.

The following event-based sampling analysis types are supported by the VTune Profiler on Android systems:

- [hotspots](#)
- [uarch-exploration](#)
- [memory-access](#)
- [system-overview](#)

To associate JITed Java functions to samples in the system-wide event-based sampling, you have the following two options:

- Specify `-target-process Process.Name` for the process you are interested in similar to how you do this for the event-based call stack collection.
- For any process you are interested in, copy the JIT files for the PID of that process into the `data.0` directory, and re-resolve the results in the VTune Profiler GUI:

1. Collect results:

```
host>./vtune -collect <analysis_type> -duration=60 -target-system=android -r system_wide_r@@@
```

2. Find PID of interesting processes:

```
adb shell ps | [grep MyApp]
u0_a79 1762 141 575912 75468 ffffffff 4006f2ef Scom.android.MyApp
```

3. Copy all jit files for processes you are interested in to the data.0 directory:

```
adb pull /data/vtune/results/localhost.1762*.jit system_wide_r000/data.0
```

Examples

Example 1: Microarchitecture Exploration Analysis

This example launches specified Android package and collects a complete list of events required to analyze typical client applications running on the 4th Generation Intel Core processor.

```
host>./vtune -collect uarch-exploration -target-system=android -r tachyon_r@@@ -target-process com.intel.tbb.example.tachyon
```

Example 2: Call Stack Analysis

By default, the VTune Profiler does not collect stack data during hardware event-based sampling. To enable call stack analysis, use the enable-stack-collection=true knob. For example:

```
host>./vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -target-system=android -r tachyon_r@@@ -target-process com.intel.tbb.example.tachyon
```

Example 3: System-wide Data Collection

To analyze performance of your target application and all other processes running on the Android system, use the --duration option and do not specify an analysis target.

```
host>./vtune -collect hotspots -knob sampling-mode=hw -target-system=android -duration=60 -r system_wide_r@@@
```

Example 4: Unplugged Mode Collection

This example configures the Hotspots analysis for the application on an Android system that will be launched after disconnecting the device from the USB cable or a network:

```
host>./vtune --collect hotspots --target-system=android -unplugged-mode -r quadrant_r@@@ --target-process com.intel.fluid
```

Custom Analysis

Use the -collect-with option to configure VTune Profiler to run a custom user-mode sampling and tracing (runss) or event-based sampling (runsa) analysis and take other than default configuration options. For example, to run a custom event-based sampling analysis, use the -collect-with option and specify required event counters with the -knob event-config option as follows:

```
host>./vtune -collect-with runsa -target-process com.intel.tbb.example.tachyon -r system_wide_r001 -knob collection-detail=stack-sampling [-event-mux] -knob event-config=CPU_CLK_UNHALTED.REF_TSC:sa=1800000,CPU_CLK_UNHALTED
```

NOTE

To display a list of events available on the target PMU, enter:

```
vtune -collect-with <collector> -target-system=android:deviceName -knob event-config=? <target_application>
```

You can take any counter that the Performance Monitoring Unit (PMU) of that processor supports. Additionally, you can enable multiple counters at a time. Each processor supports only a specific number of counters that can be taken at a time. You can take more events than the processor supports by using the `-event-mux` option, which will round robin the events you specified on the available counters in that processor.

NOTE

Typically, you are recommended to use analysis types with the predefined sets of counters. Use of specific counters is targeted for advanced users. Please note that names of some counters may not exactly correspond to the analysis scope provided with these counters.

After collecting these counters, [import the result](#) to the VTune Profiler GUI and explore the [Microarchitecture Exploration](#) data.

See Also

[Android* Targets](#)

[Set Up Android* System](#)

[Prepare an Android* Application for Analysis](#)

OpenMP* Analysis from the Command Line

Use the Intel® VTune™ Profiler command line interface for performance analysis of OpenMP applications compiled with Intel® Compiler.*

Prerequisites:

- To analyze OpenMP parallel regions, make sure to compile and run your code with the Intel® oneAPI DPC++/C++ Compiler version 2023.2.0 (or newer). If an obsolete version of the OpenMP runtime libraries is detected, VTune Profiler provides a warning message. In this case the collection results may be incomplete.

To access the newest OpenMP analysis options described in the documentation, make sure you always use the latest version of the Intel compiler.

- On Linux*, to analyze an OpenMP application compiled with GCC*, make sure the GCC OpenMP library (`libgomp.so`) contains symbol information. To verify, search for `libgomp.so` and use the `nm` command to check symbols, for example:

```
nm libgomp.so.1.0.0
```

If the library does not contain any symbols, either install/compile a new library with symbols or generate debug information for the library. For example, on Fedora* you can install GCC debug information from the `yum` repository:

```
yum install gcc-debuginfo.x86_64
```

OpenMP is a fork-join parallel model, which starts with an OpenMP program running with a single master serial-code thread. When a parallel region is encountered, that thread forks into multiple threads, which then execute the parallel region. At the end of the parallel region, the threads join at a barrier, and then the master thread continues executing serial code. It is possible to write an OpenMP program more like an MPI program, where the master thread immediately forks to a parallel region and constructs such as `barrier` and `single` are used for work coordination. But it is far more common for an OpenMP program to consist of a sequence of parallel regions interspersed with serial code.

Ideally, parallelized applications have working threads doing useful work from the beginning to the end of execution, utilizing 100% of available CPU core processing time. In real life, useful CPU utilization is likely to be less when working threads are waiting, either actively spinning (for performance, expecting to have a short wait) or waiting passively, not consuming CPU. There are several major reasons why working threads wait, not doing useful work:

- **Execution of serial portions (outside of any parallel region):** When the master thread is executing a serial region, the worker threads are in the OpenMP runtime waiting for the next parallel region.
- **Load imbalance:** When a thread finishes its part of workload in a parallel region, it waits at a barrier for the other threads to finish.
- **Not enough parallel work:** The number of loop iterations is less than the number of working threads so several threads from the team are waiting at the barrier not doing useful work at all.
- **Synchronization on locks:** When synchronization objects are used inside a parallel region, threads can wait on a lock release, contending with other threads for a shared resource.

Use VTune Profiler to understand how an application utilizes available CPUs and identify causes of CPU underutilization.

Configure and Run an Analysis

To run the OpenMP analysis from the command line, use the `threading` or `hpc-performance` analysis types. For example:

```
vtune -collect hpc-performance -- myApp
```

The HPC Performance Characterization analysis generates a summary report with OpenMP metrics and descriptions of detected performance issues.

For the Threading and HPC Performance Characterization analysis types, OpenMP analysis option is enabled by default. You may also [create a custom analysis](#) and explicitly enable this knob option: `analyze-openmp=true`. For example:

```
vtune -collect-with runsa -knob analyze-openmp=true -knob enable-user-tasks=true -- myApp
```

View Summary Report Data

When the data collection is complete, the VTune Profiler automatically generates the summary report. Similar to the **Summary** window, available in GUI, the summary report provides overall performance data of your target.

Use the following syntax to generate the Summary report from a pre-existing result:

```
vtune -report summary -result-dir <result_path>
```

For HPC Performance Characterization analysis, the command-line summary report provides an issue description for metrics that exceed the predefined threshold. If you want to skip issues in the summary report, do one of the following:

- Use the `-report-knob show-issues=false` option when generating the report, for example: `vtune -report summary -r r001hpc -report-knob show-issues=false`
- Use the `-format=csv` option to view the report in the CSV format, for example: `vtune -report summary -r r001hpc -format=csv`

Explore the OpenMP Analysis section of the summary report for inefficiencies in parallelization of the application:

```
Serial Time: 0.069s (0.3%)  
Parallel Region Time: 23.113s (99.7%)  
    Estimated Ideal Time: 14.010s (60.4%)  
    OpenMP Potential Gain: 9.103s (39.3%)  
    | The time wasted on load imbalance or parallel work arrangement is
```

| significant and negatively impacts the application performance and
| scalability. Explore OpenMP regions with the highest metric values.
| Make sure the workload of the regions is enough and the loop schedule
| is optimal.

This section shows the [Collection Time](#) as well as the duration of serial (outside of any parallel region) and parallel portions of the program. If the serial portion is significant, consider options to minimize serial execution, either by introducing more parallelism or by doing algorithm or microarchitecture tuning for sections that seem unavoidably serial. For high thread-count machines, serial sections have a severe negative impact on potential scaling (Amdahl's Law) and should be minimized as much as possible.

Estimate Potential Gain

To estimate the efficiency of CPU utilization in the parallel part of the code, use the [Potential Gain](#) metric. This metric estimates the difference in the Elapsed time between the actual measurement and an idealized execution of parallel regions, assuming perfectly balanced threads and zero overhead of the OpenMP runtime on work arrangement. Use this data to understand the maximum time that you may save by improving parallel execution.

Use the `hotspots` report to identify the hottest program units. Use the following command to list the top five parallel regions with the highest Potential Gain metric values:

```
vtune -report hotspots -result-dir r001hpc -group-by=region -sort-desc="OpenMP Potential Gain:Self" -column="OpenMP Potential Gain:Self" -limit 5
```

where

- `-report hotspots` is the `hotspots` report type
- `-group-by=region` is the action-option to group data in the report by OpenMP Regions
- `-sort-desc="OpenMP Potential Gain:Self"` is the action-option to sort data by OpenMP Potential Gain in descending order
- `-column="OpenMP Potential Gain:Self"` is the action-option to display only the OpenMP Potential Gain metric in the report
- `-limit 5` is the action-option to set the number of top items to include in the report

The command above produces the following output:

OpenMP Region	OpenMP Potential Gain
compute_rhs_omp\$parallel:24@/root/work/apps/OMP/SP/rhs.f:17:433	3.417s
x_solve_omp\$parallel:24@/root/work/apps/OMP/SP/x_solve.f:27:315	0.920s
z_solve_omp\$parallel:24@/root/work/apps/OMP/SP/z_solve.f:31:321	0.913s
y_solve_omp\$parallel:24@/root/work/apps/OMP/SP/y_solve.f:27:310	0.806s
pinvr_omp\$parallel:24@/root/work/apps/OMP/SP/pinvr.f:20:41	0.697s

If Potential Gain for a region is significant, you can go deeper and analyze inefficiency metrics like Imbalance by barriers. Use the following command:

```
vtune -report hotspots -result-dir r001hpc -group-by=region,barrier -sort-desc="OpenMP Potential Gain:Self" -column="OpenMP Potential Gain" -limit 5
```

where

- `-report hotspots` is the `hotspots` report type
- `-group-by=region, barrier` is the action-option to group data in the report by OpenMP Regions and OpenMP Barrier-to-Barrier Segment
- `-sort-desc="OpenMP Potential Gain:Self"` is the action-option to sort data by OpenMP Potential Gain in descending order
- `-column="OpenMP Potential Gain"` is the action-option to display the metrics with OpenMP Potential Gain string (including OpenMP Potential Gain: Imbalance and others)

- limit 3 is the action-option to set the number of top items to include in the report

The command above produces the output that includes the following data:

OpenMP Region	OpenMP Barrier-to-Barrier Segment	OpenMP Potential Gain	OpenMP Potential Gain:Imbalance	OpenMP Potential Gain:Lock Contention	OpenMP Potential Gain:Creation	OpenMP Potential Gain:Scheduling
compute_rh s_omp \$parallel: 24@/root/ work/OMP/S P/ rhs.f:17:4 33 x_solve_ \$omp \$parallel: 24@/home/ root/ work/OMP/S P/ x_solve.f: 27:315 z_solve_ \$omp \$parallel: 24@/root/ work/OMP/S P/ z_solve.f: 31:321 y_solve_ \$omp \$parallel: 24@/root/ work/OMP/S P/ y_solve.f: 27:310	compute_rh s_omp \$loop_barr ier_segm t@/root/ work/OMP/S P/ rhs.f:285 x_solve_ \$omp \$loop_barr ier_segm t@/root/ work/OMP/S P/ x_solve.f: 315 z_solve_ \$omp \$loop_barr ier_segm t@/root/ work/OMP/S P/ z_solve.f: 321 y_solve_ \$omp \$loop_barr ier_segm t@/root/ work/OMP/S P/ y_solve.f: 310	0.985s 0.920s 0.913s 0.806s	0.982s 0.904s 0.910s 0.803s	0s 0.012s 0.000s 0.000s	0s 0.000s 0.000s 0.000s	0.000s 0.000s 0.000s 0.000s

Analyze the **OpenMP Potential Gain** columns data that shows a breakdown of Potential Gain in the region by representing the cost (in elapsed time) of the inefficiencies with a normalization by the number of OpenMP threads. Elapsed time cost helps decide whether you need to invest into addressing a particular type of inefficiency. VTune Profiler can recognize the following types of inefficiencies:

- **Imbalance:** threads are finishing their work in different time and waiting on a barrier. If imbalance time is significant, try dynamic type of scheduling. Intel OpenMP runtime library from Intel Parallel Studio Composer Edition reports precise imbalance numbers and the metrics do not depend on statistical accuracy as other inefficiencies that are calculated based on sampling.
- **Lock Contention:** threads are waiting on contended locks or "ordered" parallel loops. If the time of lock contention is significant, try to avoid synchronization inside a parallel construct with reduction operations, thread local storage usage, or less costly atomic operations for synchronization.
- **Creation:** overhead on a parallel work arrangement. If the time for parallel work arrangement is significant, try to make parallelism more coarse-grain by moving parallel regions to an outer loop.
- **Scheduling:** OpenMP runtime scheduler overhead on a parallel work assignment for working threads. If scheduling time is significant, which often happens for dynamic types of scheduling, you can use a "dynamic" schedule with a bigger chunk size or "guided" type of schedule.
- **Atomics:** OpenMP runtime overhead on performing atomic operations.
- **Reduction:** time spent on reduction operations.

Limitations

VTune Profiler supports the analysis of parallel OpenMP regions with the following limitations:

- Maximum number of supported lexical parallel regions is 512, which means that no region annotations will be emitted for regions whose scope is reached after 512 other parallel regions are encountered.
- Regions from nested parallelism are not supported. Only top-level items emit regions.
- VTune Profiler does not support static linkage of OpenMP libraries.

See Also

[Cookbook: OpenMP* Code Analysis Method](#)

[MPI Code Analysis](#)

Java* Code Analysis from the Command Line

Intel® VTune™ Profiler provides a low-overhead user-mode sampling and tracing and hardware event-based sampling analysis of the JIT compiled code executed with Oracle® JDK or OpenJDK*. The analysis of the interpreted Java methods is [limited](#).

You may use the [hardware event-based sampling](#) data collection that monitors hardware events in the CPU's pipeline and can identify coding pitfalls limiting the most effective execution of instructions in the CPU. The [hardware performance metrics](#) are available and can be displayed against the application modules, functions, and Java code source lines. You may also run the [hardware event-based sampling collection with stacks](#) when you need to find out a call path for a function called in a driver or middleware layer in your system.

Configure Java Collection

Use the following syntax to configure Java analysis from the command line:

```
vtune -collect <analysis_type> [-[no-]follow-child] [-mrte-mode=<mrte_mode_value>] [<-knob> <knob_name=knob_option>] [--] <target>
```

where

- <analysis_type> is the type of analysis to run
- -[no-]follow-child is an action option to collect data on the processes spawned by the target process. It is recommended to enable the option for applications launched by a script. The option is enabled by default.
- <mrte_mode_value> is a profiling mode for the managed code. The auto mode is enabled by default.
- <-knob> is an option that configures the analysis
- [knobName=knobValue] is the name of the specified knob and its value
- <target> is the path and name of the application to analyze

NOTE

To see all knobs available for a predefined analysis type, enter:

```
vtune -help collect <analysis_type>
```

To see knobs for a custom analysis type, enter:

```
vtune -help collect-with <analysis_type>
```

Examples

Example 1: Running Java Analysis

The following command line runs the Hotspots analysis on a `java` command on Linux*:

```
vtune -collect hotspots -- java -Xcomp -Djava.library.path=native_lib/ia32 -cp /home/Design/Java/mixed_call MixedCall 3 2
```

Example 2: Running Analysis for Embedded Java Command

You may embed your `java` command in a batch file or executable script before running the analysis. For example, on Windows* create a `run.bat` file with the following command:

```
java.exe -Xcomp -Djava.library.path=native_lib\ia32 -cp C:\Design\Java\mixed_call MixedCall 3 1
```

The following command line runs the Hotspots analysis on a specified batch file with embedded `java` command:

```
vtune -collect hotspots -- run.bat
```

Example 3: Attaching Analysis to Java Process

In case your Java application needs to run for some time or cannot be launched at the start of this analysis, you may attach the VTune Profiler to the Java process. To do this, specify the following analysis target: `--target-process java`.

NOTE

The dynamic attach mechanism is supported only with the Java Development Kit (JDK).

The following example attaches the Hotspots analysis to a running Java process on Linux:

```
vtune -collect hotspots --target-process java
```

View Summary Report

VTune Profiler automatically generates the summary report when data collection completes. Similar to the **Summary** window, available in GUI, the command line report provides overall performance data of your Java target.

NOTE

For more information on analyzing the summary report data, refer to the [Summary Report](#) section.

Examples

The following example generates the summary report for the Hotspots analysis result. For [user-mode sampling and tracing analysis](#) results, the summary report includes Collection and Platform information, CPU information and summary per the basic metrics.

On Windows:

```
Collection and Platform Info
-----
Parameter          r001hs

-----
Operating System    Microsoft Windows 10
Result Size        21258782
Collection start time 11:58:36 15/04/2019 UTC
Collection stop time 11:58:50 15/04/2019 UTC

CPU
---
Parameter          r001hs

-----
Name               4th generation Intel(R) Core(TM) Processor family
Frequency         2494227391
Logical CPU Count 4

Summary
-----
Elapsed Time:      12.939
CPU Time:          14.813
Average CPU Usage: 1.012
```

On Linux:

```
Collection and Platform Info
-----
Parameter          r002hs

-----
Application Command Line /tmp/java_mixed_call/src/run.sh

Operating System    3.16.0-30-generic NAME="Ubuntu"
VERSION="14.04.2 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.2 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
Computer Name       10.125.21.55

Result Size         11560723

Collection start time 13:55:00 05/02/2019 UTC
Collection stop time 13:55:10 05/02/2019 UTC

CPU
---
Parameter          r001hs

-----
Name               3rd generation Intel(R) Core(TM) Processor family
Frequency         3492067692
Logical CPU Count 8
```

Summary

```
-----
Elapsed Time:      10.183
CPU Time:         19.200
Average CPU Usage: 1.885
```

This example generates the summary report for the Hotspots analysis (hardware event-based sampling mode) result. For [hardware event-based sampling analysis](#) results, the summary report includes Collection and Platform information, CPU information, summary per the basic metrics, and an event summary.

Collection and Platform Info

```
-----
Parameter          r002hs
-----
Operating System    3.16.0-30-generic NAME="Ubuntu"
VERSION="14.04.2 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.2 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
Result Size        171662827
Collection start time 10:44:34 15/04/2019 UTC
Collection stop time   10:44:50 15/04/2019 UTC
```

CPU

```
-----
Parameter          r002hs
-----
Name               4th generation Intel(R) Core(TM) Processor family
Frequency          2494227445
Logical CPU Count  4
```

Summary

```
-----
Elapsed Time:      15.463
CPU Time:         6.392
Average CPU Usage: 0.379
CPI Rate:          1.318
```

Event summary

Hardware Event Type	Hardware Event Count:Self	Hardware Event Sample Count:Self	Events
Per Sample			

INST_RETIRED.ANY	13014608235	8276	1900000
CPU_CLK_UNHALTED.THREAD	17158609921	8207	1900000
CPU_CLK_UNHALTED.REF_TSC	15942400300	5163	1900000
BR_INST_RETIRED.NEAR_TAKEN	1228364727	4648	200003
CALL_COUNT	213650621	75413	1
ITERATION_COUNT	370567815	84737	1
LOOP_ENTRY_COUNT	162943310	70069	1

Identify Hottest Methods

Use the `hotspots` command line report as a starting point for identifying program units (for example: functions, modules, or objects) that take the most processor time (Hotspots analysis), underutilize available CPUs or have long waits (Threading analysis), and so on.

The report displays the hottest program units in the descending order by default, starting from the most performance-critical unit. The command-line reports provide the same data that is displayed in the default GUI analysis [viewpoints](#).

NOTE

- To display a list of available groupings for a `hotspots` report, enter: `vtune -report hotspots -r <result_dir> group-by=?`.
- To set the number of top items to include in a report, use the `limit` action option: `vtune -report <report_type> -limit <value> -r <result_dir>`

Examples

This example generates the `hotspots` report for the Hotspots analysis result and groups the data by module. The result file is not specified and VTune Profiler uses the latest analysis result.

```
vtune -report hotspots
```

On Windows:

Function	CPU Time	CPU Time:Effective Time	CPU Time:Effective Time:Idle	CPU Time:Effective Time:Poor	CPU Time:Effective Time:Ok	CPU Time:Effective Time:Over	CPU Time:Spin Time	CPU Time:Overhead Time	Module	Function
(Full)	Source File	Start Address								
consume_time	10.371s	10.371s								
0s	10.341s		0.020s							
0.010s		0s		0s					0s	
mixed_call.dll	consume_time		mixed_call.c	0x180001000						
NtWaitForSingleObject	1.609s			0s						
0s		0s			0s					
0s			1.609s						0s	ntdll.dll
NtWaitForSingleObject	[Unknown]	0x1800906f0								
WriteFile	0.245s		0.245s							
0.009s		0.190s			0.030s					
0.016s		0s		0s					0s	
KERNELBASE.dll	WriteFile		[Unknown]	0x180001c50						
func@0x707d5440	0.114s		0.010s							
0s		0.010s			0s					
0s		0s	0.104s						0s	jvm.dll
func@0x707d5440	[Unknown]	0x707d5440								
func@0x705be5c0	0.072s		0.025s							
0s		0.025s			0s					
0s		0s	0.047s						0s	jvm.dll
func@0x705be5c0	[Unknown]	0x705be5c0								
...										

On Linux:

```
Function          CPU Time  CPU Time:Effective Time  CPU Time:Effective Time:Idle  CPU
Time:Effective Time:Poor  CPU Time:Effective Time:Ok   CPU Time:Effective Time:Ideal  CPU
Time:Effective Time:Over  CPU Time:Spin Time  CPU Time:Overhead Time  Module           Function
(Full)    Source File  Start Address
-----
-----
[libmixed_call.so]  17.180s          17.180s
0s                  17.180s          0s
0s                  0s              0s
[libmixed_call.so]  [Unknown]      0
[libjvm.so]         1.698s          1.698s
0.020s             1.678s          0s
0s                  0s              0s
[libjvm.so]         [Unknown]      0
[libpthread.so.0]   0.136s          0.136s
0s                  0.136s          0s
0s                  0s              0s
[libpthread.so.0]   [Unknown]      0
[libtpsstool.so]   0.052s          0.052s
0s                  0.052s          0s
0s                  0s              0s
[libtpsstool.so]   [Unknown]      0
...

```

The following example generates the `hotspots` report for the specified Hotspots analysis result (hardware event-based sampling mode), sets the number of items to include in the report to 3, and groups the report data by application module.

```
vtune -report hotspots -limit 3 -r r002hs -group-by module
```

On Windows:

```
Module          CPU Time  CPU Time:Effective Time  CPU Time:Effective Time:Idle  CPU
Time:Effective Time:Poor  CPU Time:Effective Time:Ok   CPU Time:Effective Time:Ideal  CPU
Time:Effective Time:Over  CPU Time:Spin Time  CPU Time:Overhead Time  Instructions Retired CPI
Rate  Wait Rate  CPU Frequency Ratio Context Switch Time  Context Switch Time:Wait Time
Context Switch Time:Inactive Time  Context Switch Count  Context Switch Count:Preemption
Context Switch Count:Synchronization  Module
Path
-----
-----
mixed_call.dll  15.294s          15.294s
0.419s          14.871s          0.004s
0s              0s              0s
21,148,958,284 1.907           0.000           1.149
1.401s          0s
26,769          26,769
Java\module Java\java_mixed_call\vc9\bin32\mixed_call.dll
jvm.dll        0.582s          0.582s
0.033s          0.547s          0.002s
0s              0s              0s
0               C:\work\module

```

```

792,807,896      1.513      0.437      0.899
0.047s           0.005s
462              451
(x86)\Java\jre8\bin\client\jvm.dll
ntoskrnl.exe     0.404s      0.404s
0.034s           0.370s
0s               0s          0s          0.001s
660,557,183     1.096      0.000
0.780

C:\WINDOWS\system32\ntoskrnl.exe
...

```

On Linux:

```

Module      CPU Time  CPU Time:Effective Time  CPU Time:Effective Time:Idle  CPU
Time:Effective Time:Poor  CPU Time:Effective Time:Ok  CPU Time:Effective Time:Ideal  CPU
Time:Effective Time:Over  CPU Time:Spin Time  CPU Time:Overhead Time  Instructions Retired  CPI
Rate  Wait Rate  CPU Frequency Ratio  Context Switch Time  Context Switch Time:Wait Time
Context Switch Time:Inactive Time  Context Switch Count  Context Switch Count:Preemption
Context Switch Count:Synchronization  Module
Path
-----
-----
-----
libmixed_call.so  15.294s      15.294s
0.419s           14.871s      0.004s
0s               0s          0s          0s
21,148,958,284   1.907      0.000      1.149
1.401s           0s
26,769           26,769
java_mixed_call/src/libmixed_call.so
libjvm.so         0.582s      0.582s
0.033s           0.547s      0.002s
0s               0s          0s          0s
792,807,896     1.513      0.437      0.899
0.047s           0.005s
462              451
0.042s           0.042s
11   /tmp/
java_mixed_call/src/libmjvm.so
...
...
```

Analyze Stacks

To get the maximum performance out of your Java application, writing and compiling performance critical modules of your Java project in native languages, such as C or even assembly. This will help your application take advantage of vectorization and make complete use of powerful CPU resources. This way of programming helps to employ powerful CPU resources like vector computing (implemented via SIMD units and instruction sets). In this case, compute-intensive functions become hotspots in the profiling results, which is expected as they do most of the job. However, you might be interested not only in hotspot functions, but in identifying locations in Java code these functions were called from via a JNI interface. Tracing such cross-runtime calls in the mixed language algorithm implementations could be a challenge.

Use the `callstacks` report to display full stack data for each hotspot function and identify the impact of each stack on the function CPU or Wait time.

NOTE

To display a list of available groupings for a `callstacks` report, enter `vtune -report callstacks -r <result_dir> group-by=?.`

Example

The following command line generates the `callstacks` report for the specified Hotspots analysis result.

On Windows:

Function (Full)	Function Stack Source File	CPU Time	Module	Function
		Start Address		
consume_time		10.371s	mixed_call.dll	
consume_time	mixed_call.c	0x180001000		
MixedCall::CallNativeFunc(int)	MixedCall.java	10.371s	[Compiled Java code]	
MixedCall::foo4		0s	[Compiled Java code]	
MixedCall::foo4(int)	MixedCall.java	0x186c1ae3		
MixedCall::foo3		0s	[Compiled Java code]	
MixedCall::foo3(int)	MixedCall.java	0x186bb583		
MixedCall::foo2		0s	[Compiled Java code]	
MixedCall::foo2(int)	MixedCall.java	0x186bb583		
MixedCall::foo1		0s	[Compiled Java code]	
MixedCall::foo1(int)	MixedCall.java	0x186bb583		
MixedCall::run		0s	[Compiled Java code]	
MixedCall::run()	MixedCall.java	0x186bb19d		
call_stub		0s	[Dynamic code]	
call_stub	[Unknown]	0x18010827		
...				

On Linux:

Function (Full)	Function Stack Source File	CPU Time	Module	Function
		Start Address		
[libmixed_call.so]		17.180s	libmixed_call.so	
[libmixed_call.so]	[Unknown]	0		
	[libmixed_call.so]	8.600s	libmixed_call.so	
[libmixed_call.so]	[Unknown]	0		
	MixedCall::CallNativeFunc	0s	[Compiled Java code]	
MixedCall::CallNativeFunc(int)	MixedCall.java	0x7fb63937eec0		
MixedCall::foo4	MixedCall.java	0s	[Compiled Java code]	
MixedCall::foo4(int)	MixedCall.java	0x7fb6393831e3		
MixedCall::foo3	MixedCall.java	0s	[Compiled Java code]	
MixedCall::foo3(int)	MixedCall.java	0x7fb63938046c		
MixedCall::foo2		0s	[Compiled Java code]	
MixedCall::foo2(int)	MixedCall.java	0x7fb63938046c		
MixedCall::foo1		0s	[Compiled Java code]	
MixedCall::foo1(int)	MixedCall.java	0x7fb63938046c		
MixedCall::run		0s	[Compiled Java code]	
MixedCall::run()	MixedCall.java	0x7fb63938009b		
...				

Analyze Hardware Metrics

VTune Profiler provides an advanced profiling option of optimizing Java applications for the CPU microarchitecture utilized in your platform. Although Java and JVM technology is intended to free a developer from hardware architecture specific coding, once Java code is optimized for the current Intel microarchitecture, it will most probably keep this advantage for future generations of CPUs.

VTune Profiler counts the number of hardware events during the [hardware event-based sampling collection](#) to help you understand how your Java application utilizes available hardware resources. Use the `hw-events` report type to display hardware events count per application functions in the descending order by default.

NOTE

To display a list of available groupings for a `hw-events` report, enter `vtune -report hw-events -r <result_dir> group-by=?`.

Example

This example generates the `hw-events` report for the specified Hotspots analysis (hardware event-based sampling mode) result.

On Windows:

```

Function           Hardware Event Count:INST_RETIRED.ANY   Hardware Event
Count:CPU_CLK_UNHALTED.THREAD Hardware Event Count:CPU_CLK_UNHALTED.REF_TSC   Hardware Event
Count:BR_INST_RETIRED.NEAR_TAKEN Hardware Event Count:ITERATION_COUNT   Hardware Event
Count:LOOP_ENTRY_COUNT   Hardware Event Count:CALL_COUNT   Context Switch Time   Context Switch
Time:Wait Time   Context Switch Time:Inactive Time   Context Switch Count   Context Switch
Count:Preemption   Context Switch Count:Synchronization   Module           Function (Full)
Source File   Start Address
-----
-----
-----
consume_time           8,649,248,560
28,577,118,234          126,927,912
25,656,728,125          0
126,914,825          0s
0           0.217s
0.217s           4,147
4,147           0  mixed_call.dll  consume_time
mixed_call.c 0x180001000
NtWaitForSingleObject      1,683,967,360
3,955,057,542
716,832,500           200,003
0           0
66,678
223.825s           62.467s
161.358s
9,030           5,158
3,873  ntdll.dll
NtWaitForSingleObject [Unknown] 0x1800906f0
WriteFile           1,207,593,104
1,022,685,972
1,713,743,550
0           0
0           61,803
0.340s
0.003s           0.337s
962
8  KernelBase.dll  WriteFile
[Unknown]
0x180001c50

```

On Linux:

```

Function           Hardware Event Count:INST_RETIRE.DANY   Hardware Event
Count:CPU_CLK_UNHALTED.THREAD  Hardware Event Count:CPU_CLK_UNHALTED.REF_TSC Context Switch
Time  Context Switch Time:Wait Time  Context Switch Time:Inactive Time  Context Switch Count
Context Switch Count:Preemption  Context Switch Count:Synchronization  Module
Function (Full)    Source File  Start Address
-----
-----
[libmixed_call.so]          21,148,958,284
40,338,264,445            35,096,009,324
1.401s                      0s
26,769                      26,769
[libmixed_call.so] [libmixed_call.so] [Unknown] 0
[libjvm.so]                  792,807,896
1,199,773,286              1,335,034,092
0.047s                      0.005s
462                          451
[libjvm.so] [Unknown] 0
...

```

Limitations

VTune Profiler supports analysis of Java applications with some limitations:

- System-wide profiling is not supported for managed code.
- The JVM interprets some rarely called methods instead of compiling them for the sake of performance. VTune Profiler does not recognize interpreted Java methods and marks such calls as !Interpreter in the restored call stack.

If you want such functions to be displayed in stacks with their names, force the JVM to compile them by using the `-Xcomp` option (show up as [Compiled Java code] methods in the results). However, the timing characteristics may change noticeably if many small or rarely used functions are being called during execution.

- When opening source code for a hotspot, the VTune Profiler may attribute events or time statistics to an incorrect piece of the code. It happens due to JDK Java VM specifics. For a loop, the performance metric may slip upward. Often the information is attributed to the first line of the hot method's source code.
- Consider events and time mapping to the source code lines as approximate.
- For the user-mode sampling based Hotspots analysis type, the VTune Profiler may display only a part of the call stack. To view the complete stack on Windows, use the `-Xcomp` additional command line JDK Java VM option that enables the JIT compilation for better quality of stack walking. On Linux, use additional command line JDK Java VM options that change behavior of the Java VM:
 - Use the `-Xcomp` additional command line JDK Java VM option that enables the JIT compilation for better quality of stack walking.
 - On Linux* x86, use client JDK Java VM instead of the server Java VM: either explicitly specify `-client`, or simply do not specify `-server` JDK Java VM command line option.
 - On Linux x64, specify `-XX:-UseLoopCounter` command line option that switches off on-the-fly substitution of the interpreted method with the compiled version.
- Java application profiling is supported for the Hotspots and Microarchitecture analysis types. Support for the Threading analysis is limited as some embedded Java synchronization primitives (which do not call operating system synchronization objects) cannot be recognized by the VTune Profiler. As a result, some of the timing metrics may be distorted.
- There are no dedicated libraries supplying a user API for collection control in the Java source code. However, you may want to try applying the native API by wrapping the `__itt` calls with JNI calls.

See Also

[Java* Code Analysis
from GUI](#)

[Enable Java* Analysis on Android* System](#)

[Stitch Stacks for Intel® oneAPI Threading Building Blocks or OpenMP* Analysis](#)

Command Line Interface Reference

Select an item from the Table of Contents to continue.

NOTE

See the [VTune Profiler CLI Cheat Sheet](#) quick reference on VTune Profiler command line interface.

Option Descriptions and General Rules

All option descriptions in the Intel® VTune™ Profiler Command Line Interface Reference follow the general rules and templates described below.

Option Description

Each option description provides the following information:

- A short description of the option.
- **Products:** This section lists the names of products supporting this option.
- **GUI Equivalent:** This section shows the equivalent of the option in the integrated development environment (IDE)/standalone GUI client. If no equivalent is available, None is specified.
- **Syntax:** This section describes the command line syntax of the option.
- **Arguments:** This section lists the arguments related to the option. If it has no arguments, None is specified.
- **Default:** This section shows the default setting for the option.
- **Modifiers:** This section lists the modifiers for the described action. The section is only available for actions.
- **Actions Modified:** This section lists the actions modified by the described option. The section is only available for modifiers.
- **Description:** This section provides the full description for the option.
- **Alternate Options:** These options can be used instead of the described option. If no alternate options are available, None is specified.
- **Example:** This is a typical usage example of the option.
- **See Also:** This section provides links to further information related to the option such as other options or corresponding GUI procedures.

General Rules

- Options can be preceded by a single dash ("") or a double dash ("--").
- Option names and values can be separated with a space (" "), or an equal sign ("=").
- Options defining the collection are specified before the analyzed target and can appear on the command line in any order. Options related to the target are specified after the target
- You cannot combine options with a single dash. For example, -q and -c options cannot be specified as -qc option.

- Options may have short and long names. Short names consist of one letter. Long names consist of one or more words separated by dashes. Both short and long names are case-sensitive. Long and short option names can be used interchangeably. For example, you may use `-report` or `-R` to generate a report.
- Long names of the options can be abbreviated. If the option consists of several words you can abbreviate each word, keeping the dash between them. Make sure an abbreviated version unambiguously matches the long name. For example, the `-option-name` option can be abbreviated as `-opt-name`, `-op-na`, `-opt-n`, or `-o-n`.
- If the abbreviation is ambiguous between two available options, a syntax error is reported.
- You can disable Boolean default options by specifying `-no-<optionname>` from the command line. For example, to avoid displaying a summary report after analysis, run `vtune` with the `-no-summary` option. Conversely, if the default is `-no-<option>`, you can disable it by specifying `<optionname>`.
- You can specify multiple values for the option by using the option several times, or by using the option once and specifying comma-separated values (make sure there are no spaces around the commas). The examples below are equivalent and specify two filters for the `r001tr` result when generating a hotspots report.

On Linux*:

```
vtune -R hotspots -r r001tr -filter module=tachyon -filter module=vmlinux
```

```
vtune -R hotspots -r r001tr -filter module=tachyon,vmlinux
```

On Windows*:

```
vtune -R hotspots -r r001tr -filter module=ntdll.dll -filter module=main.exe
```

```
vtune -R hotspots -r r001tr -filter module=ntdll.dll,main.exe
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

allow-multiple-runs

Enable multiple runs to achieve more precise results for hardware event-based collections.

GUI Equivalent

Allow multiple runs option in the [WHAT](#) pane

Syntax

```
-allow-multiple-runs  
-no-allow-multiple-runs
```

Default

```
-no-allow-multiple-runs
```

Actions Modified

`collect`, `collect-with`

Description

By default, `no-allow-multiple-runs` is enabled, and a `collect` or `collect-with` action performs a single analysis run. Performing multiple analysis runs can provide more precise results for hardware event-based collections. To enable event multiplexing, specify `allow-multiple-runs`.

Example

This example runs the target application twice, collecting different events on each run.

```
vtune -collect hotspots -allow-multiple-runs -- /home/test/sample
```

See Also

[Allow Multiple Runs or Multiplex Events from GUI](#)

[vtune Command Syntax](#)

analyze-kvm-guest

Analyze a KVM guest OS running on your system.

GUI Equivalent

[Analyze KVM guest OS option in the WHAT pane](#)

Syntax

```
-analyze-kvm-guest | -no-analyze-kvm-guest
```

Default

```
-no-analyze-kvm-guest
```

Actions Modified

[collect-with](#)

Description

Analyze a KVM guest OS running on your system. For successful analysis, make sure to do the following:

1. Copy these files from the guest OS to your local file system:
 - /proc/kallsyms
 - /proc/modules
 - any guest OS's modules of interest (vmlinuz, any *.ko files, and so on)
2. Specify a Linux target system for analysis using the [target-system](#) option.
3. Configure your VTune Profiler analysis target by using the [kvm-guest-kallsyms](#), [kvm-guest-modules](#), and [search-dir](#) options to specify paths to the files copied in step 1 for accurate module resolution.
4. Configure your [collect-with](#) by using the [knob ftrace-config=<events>](#) option to specify Linux FTrace* events tracking IRQ injection process.

Example

Enable a custom hardware event-based sampling collection for the KVM guest OS and collect irq, softirq, workq, and kvm FTrace events:

```
vtune --target-system=ssh: user1@172.16.254.1 -collect-with runsa -knob event-
config=CPU_CLK_UNHALTED.REF_TSC:sa=3500000,CPU_CLK_UNHALTED.THREAD:sa=3500000,INST_RETIREDA.NY:sa
=3500000 -knob enable-stack-collection=true -knob ftrace-config=irq,softirq,workq,kvm -analyze-
kvm-guest -kvm-guest-kallsyms=/home/vtune/[guest]/kvm.kallsyms -kvm-guest-modules=/home/vtune/
[guest]/kvm.modules --search-dir sym:p=/home/vtune/ --target-pid 9791
```

See Also

[Profile KVM Kernel and User Space on the KVM System from GUI](#)

[Targets in Virtualized Environments](#)

Profile Targets on a KVM* Guest System

[knob](#)
[ftrace-config](#)
[kvm-guest-modules](#)

[kvm-guest-kallsyms](#)

[vtune Actions](#)

[vtune Command Syntax](#)

analyze-system

Enable analysis of all processes running on the system.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Advanced** section > **Analyze system-wide** option

Syntax

`-analyze-system`
`-no-analyze-system`

Default

`no-analyze-system`

Actions Modified

`collect`, `collect-with`

Description

For [hardware event-based analysis types](#), `no-analyze-system` is enabled by default, so only the target process is analyzed. Use `analyze-system` if you want to analyze all processes running on the system. Data on CPU consumption for these other processes shows how they affect the performance of the target process.

Example

Perform the Hotspots analysis (hardware event-based sampling mode) of all processes running on the system.

```
vtune -collect hotspots -knob sampling-mode=hw -analyze-system -- /home/test/sample
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

app-working-dir

Specify the application directory in auto-generated commands.

GUI Equivalent

Configure Analysis window > **HOW** pane > **Launch Application** target type

Syntax

```
-app-working-dir=<PATH>
```

Arguments

A string containing the PATH/name.

Default

Default is the current working directory.

Actions Modified

`collect, collect-with`

Description

If your data files are stored in a separate location from the application, use the `app-working-dir` option to specify the application working directory.

Example

This command line example changes the application directory to `C:\myAppDirectory` (on Windows*) and to `/home/myAppDirectory`(on Linux*) to run the `myApp` application, uses binary and symbol files found in the directory specified by the `search-dir` option to finalize the result, writes the result in the default result directory, and then returns to the working directory.

On Windows:

```
vtune-cl -collect hotspots -app-working-dir C:\myAppDirectory -search-dir C:\mySources --  
myApp.exe
```

On Linux:

```
vtune-cl -collect hotspots -app-working-dir /home/myAppDirectory -search-dir /home/mySources --  
myApp
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

archive

Archive collected results.

Syntax

```
-archive -result-dir <PATH>
```

Description

Archive collected performance results and view them later on another system.

Examples

Archive Hotspots analysis results from the specified Linux directory.

```
vtune -archive -result-dir /temp/test/baseline
```

call-stack-mode

Choose how to show system functions in the call stack.

GUI Equivalent

Toolbar: Filter > Call Stack Mode menu

Syntax

```
-call-stack-mode <value>
```

Arguments

<value> - Type of call stack display. The following values are available:

Argument	Description
all	Display both system and user functions.
user-only	Show user functions only.
user-plus-one	Show user functions and system functions called directly from user functions.

Default

user-plus-one Collected data is attributed to user functions and system functions called directly from user functions.

Actions Modified

`collect, finalizeimportreport`

Description

Use the `call-stack-mode` option when performing data collection, finalization or importation, to set call stack data attribution for the result or report. If set for collection, finalization or importation, this sets the default view when the result is opened in the GUI, and applies to any reports unless overridden in the command used to generate the report.

Example

Generate a hotspots result and include system as well as user functions in the call stack. This is now the project-level setting, and if the result is viewed in the GUI, the call stack shows both user functions and system functions.

```
vtune -collect hotspots -call-stack-mode all -- myApp.exe
```

This command generates a hotspots report from the most recent hotspots analysis result, groups the result data by function, and then overrides the project-level setting so that the call stack shows user functions plus system functions called directly from user functions.

```
vtune -report hotspots -group-by function -call-stack-mode user-plus-one
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

collect

Run the specified analysis type and collect data into a result.

GUI Equivalent

Configure Analysis window > **HOW** pane

Syntax

```
-collect <analysis_type>  
-c <analysis_type>
```

Arguments

analysis_type Type of performance analysis. The following analysis types and configurable knobs are supported:

anomaly-detection Identify performance anomalies in frequently recurring intervals of code like loop iterations. Perform fine-grained analysis at the microsecond level.

- -knob `ipt-regions-to-load` to specify the maximum number (10-5000) of code regions to load for detailed analysis. To load details efficiently, maintain this number at or below 1000.
- -knob `max-region-duration` to specify the maximum duration (0.001-1000ms) of analysis per code region.

Collection type: user-mode sampling and tracing collection or hardware event-based sampling.

hotspots

Identify your most time-consuming source code using one of the available collection modes:

- -knob `sampling-mode=sw` (former Basic Hotspots) to collect hotspots and stack information based on the user-mode sampling and tracing, which does not require sampling drivers but incurs higher collection overhead). This mode cannot be used to profile a system, but must either launch an application/process or attach to one.
- -knob `sampling-mode=hw` (former Advanced Hotspots) to sample all processes on the system and identify hotspots.

Collection type: user-mode sampling and tracing collection or hardware event-based sampling.

Knobs: `enable-characterization-insights`, `enable-stack-collection`, `sampling-interval`, `sampling-mode`.

threading

Analyze how your application is using available logical CPU cores, discover where parallelism is incurring synchronization overhead, find how waits affect your application's performance, and identify potential candidates for parallelization.

Collection type: user-mode sampling and tracing collection.

Knobs: `sampling-interval`.

memory-consumption	Analyze memory consumption by your Linux application, its distinct memory objects and their allocation stacks. Collection type: user-mode sampling and tracing collection. Knobs: mem-object-size-min-thres.
hpc-performance	Identify opportunities to optimize CPU, memory, and FPU utilization for compute-intensive or throughput applications. Collection type: hardware event-based sampling collection. Knobs: enable-stack-collection, collect-memory-bandwidth, sampling-interval, dram-bandwidth-limits.
uarch-exploration (formerly known as general-exploration)	Identify and locate the most significant hardware issues that affect the performance of your application. Use this analysis type as a starting point for microarchitecture analysis. Collection type: hardware event-based sampling collection. Knobs: enable-stack-collection, collect-memory-bandwidth, enable-user-tasks.
memory-access	Measure a set of metrics to identify memory access related issues (for example, specific for NUMA architectures). Collection type: hardware event-based sampling collection. Knobs: sampling-interval, dram-bandwidth-limits, analyze-openmp; Linux only: analyze-mem-objects, mem-object-size-min-thres.
sgx-hotspots (deprecated)	Analyze hotspots inside security enclaves for systems with the Intel® Software Guard Extensions (Intel® SGX) feature enabled. Collection type: hardware event-based sampling collection. Knobs: enable-stack-collection, enable-user-tasks.
tsx-exploration (deprecated)	Analyze Intel® Transactional Synchronization Extensions (Intel® TSX) usage. Collection type: hardware event-based sampling collection. Knobs: enable-user-tasks, analysis-step.
tsx-hotspots (deprecated)	Analyze hotspots inside transactions. Knobs: enable-user-tasks, enable-stack-collection.
cpugpu-concurrency (deprecated)	Enable the CPU/GPU Concurrency analysis and explore code execution on the various CPU and GPU cores in your system, correlate CPU and GPU activity and identify whether your application is GPU or CPU bound. Knobs: sampling-interval, enable-user-tasks, enable-user-sync, enable-gpu-usage, gpu-counters-mode, enable-gpu-runtimes.
gpu-hotspots	Identify GPU tasks with high GPU utilization and estimate the effectiveness of this utilization. Collection type: hardware event-based sampling collection.

	Knobs: gpu-sampling-interval, enable-gpu-usage, gpu-counters-mode, enable-gpu-runtimes, enable-stack-collection.
gpu-profiling (deprecated)	Analyze GPU kernel execution per code line and identify performance issues caused by memory latency or inefficient kernel algorithms. Collection type: hardware event-based sampling collection. Knobs: gpu-profiling-mode, kernels-to-profile.
graphics-rendering (preview)	Analyze the CPU/GPU utilization of your code running on the Xen virtualization platform. Explore GPU usage per GPU engine and GPU hardware metrics that help understand where performance improvements are possible. If applicable, this analysis also detects OpenGL-ES API calls and displays them on the timeline. Collection type: hardware event-based sampling collection. Knobs: gpu-sampling-interval, gpu-counters-mode.
fpga-interaction	Analyze the CPU/FPGA interaction issues via exploring OpenCL kernels running on FPGA, identify the most time-consuming FPGA kernels. Collection type: hardware event-based sampling collection. Knobs: sampling-interval, enable-stack-collection.
io	Monitor utilization of the IO subsystems, CPU and processor buses. Collection type: hardware event-based sampling collection. Knobs: collect-pcie-bandwidth, mmio, iommu, collect-memory-bandwidth, dram-bandwidth-limits, dpdk, spdk, kernel-stack.
system-overview	Evaluate general behavior of Linux* or Android* target systems and correlate power and performance metrics with IRQ handling. Collection type: hardware event-based sampling collection. Knobs: collection-detail.

NOTE

For Android* systems, VTune Profiler provides GPU analysis only on processors with Intel® HD Graphics and Intel® Iris® Graphics. You cannot view the collected results in the CLI report. To view the results, open the result file in GUI.

Default

OFF The `vtune` command runs no data collection unless the `collect` action is specified.

Modifiers

`[no]-allow-multiple-runs, [no]-analyze-system, data-limit, discard-raw-data, duration, finalization-mode, [no]-follow-child, knob, mrte-mode, quiet, resume-after, return-app-exitcode, ring-buffer, search-dir, start-paused, , strategy, [no]-summary, target-duration-type, target-pid, target-process, target-system, trace-mpi, no-unplugged-mode, user-data-dir, verbose`

Description

Use the `collect` action to perform analysis and collect data. By default, this process performs the specified type of analysis, collects and finalize data into a result file, and outputs a Summary report to stdout. In most cases you will want to use the `search-dir` action-option to specify the search directory. Some analysis types support the `knob` option, which allow you to specify additional level settings.

There are many options that you can use to customize the behavior of the `collect` action to suit your purposes. For example, you can choose whether to analyze a child process only, whether to start collection after a certain amount of time has elapsed, or whether to perform collection without finalizing the result. There are a few examples included in this topic. For more information, use one of the `help` commands described below, or browse or search this documentation for information on the type of analysis you wish to perform.

NOTE

To access the most current command line documentation for an action, enter `vtune -help <action>`, where `<action>` is one of the available actions. To see all available actions, enter `vtune -help`.

To view a list of analysis types supported for your processor:

```
vtune -help collect
```

To view detailed information on the supported analysis type:

```
vtune -help collect <analysis_type>
```

This command displays a description for the specified analysis type and its configuration options (knobs).

Alternate Options

`collect-with`

The `collect-with` action performs the same basic functions as the `collect` action, but provides additional knob settings for custom configuration.

Examples

This command runs the hotspots analysis in the hardware event-based sampling mode for a Linux `myApp` application, writes the result to the default directory, and outputs a summary report by default.

```
vtune -collect hotspots -knob sampling-mode=hw -- /home/test/sample
```

For best results, specify the search directories. This example collects a default-named hotspots result, searching for symbol files in the `home/import/system_modules` high-priority search directory.

```
vtune -collect hotspots -search-dir /home/import/system_modules -- /home/test/sample
```

You can use the `target-pid` or `target-process` options to attach a Hotspots collection to a running process. In this example, `target-pid` is used to attach the collection to a running process whose ID is 1234.

```
vtune -collect hotspots -target-pid 1234
```

The `no-auto-finalize` action-option start a Threading analysis, collect performance data, and exit without finalizing the result.

```
vtune -collect threading -no-auto-finalize -- /home/test/sample
```

See Also

[Run Command Line Analysis](#)

`collect-with`

action

Analyze Performance

in GUI

vtune Command Syntax

collect-with

Run a custom hardware event-based sampling or user-mode sampling and tracing collection using your settings.

GUI Equivalent

Custom Analysis

Syntax

`-collect-with <collector_name>`

Arguments

collector_name	Description
<code>runsa</code>	Perform hardware event-based sampling collection.
<code>runss</code>	Perform user-mode sampling and tracing collection.

Modifiers

[no-]allow-multiple-runs, analyze-kvm-guest, [no-]analyze-system, app-working-dir, call-stack-mode, cpu-mask, data-limit, discard-raw-data, duration, finalization-mode, [no-]follow-child, inline-mode, knob, mrte-mode, quiet, result-dir, resume-after, return-app-exitcode, ring-buffer, search-dir, start-paused, strategy, [no-]summary, target-duration-type, target-pid, target-process, no-unplugged-mode, user-data-dir, verbose

Description

Use the `collect-with` action when you want finer control over analysis settings than the `collect` action can offer. Specify both the collector type and the `knob`. The collector type determines the type of collection, and the knob determines the level or granularity. Lower levels are coarser grained, while higher levels are finer grained. The analysis process includes finalization of the result, and a summary report is displayed by default.

For the `runsa` (event-based sampling) collector, the `event-config` knob option specifies the list of events to collect. To display a list of events available on the target PMU, enter:

```
vtune -collect-with runsa -knob event-config=? <target>
```

The command returns names and short descriptions of available events. For more information on the events, use [Intel Processor Events Reference](#)

NOTE

- To access the most current command line documentation for the `collect` or `collect-with` action, enter `vtune -help collect` or `vtune -help collect-with`.
 - For the most current information on available knobs, enter `vtune -help collect <analysis_type>` or `vtune -help collect-with <analysis_type>`, where `<analysis_type>` is the type of analysis you wish to perform.
-

Alternate Options

Use the `collect` action with predefined settings.

Example

This example runs the hardware event-based sampling collector for the sample Linux* application on the specified events and displays a summary report.

```
vtune -collect-with runsa -knob event-
config=CPU_CLK_UNHALTED.CORE,CPU_CLK_UNHALTED.REF,INST_RETIRE...
```

See Also

[Hardware Event-based Sampling Collection](#)

Custom Analysis

in GUI

column

Specify substrings for the column names to display only corresponding columns in the report.

GUI Equivalent

Toolbar: Command

Syntax

`-column=<string>`

Arguments

`<string>` - Full name of the column or its substring.

Actions Modified

`report , report-output`

Description

Filter in the `report` to display only data columns (typically corresponding to performance metrics or hardware events) with the specified `<string>` in the title. For example, specify `-column=Total` to view only Total metrics in the report. Columns used for data grouping are always displayed.

To display a list of columns available for a particular report, type: `vtune -report <report_name> -r <result_dir> -column=?.`

Example

Display grouping and data columns only for event columns with the `*INST_RETIRE.*` string in the title:

```
vtune -R hw-events -r r000ue -column=INST_RETIRE...
```

Obtain a list of columns available for the hw-events report for a Microarchitecture Exploration analysis result on Linux*:

```
vtune -report hw-events -r /temp/test/r029ue/r029ue -column=?
vtune: Using result path '/temp/test/r029ue/r029ue'
```

```
Available values for '-column' option are:
Function
Module
Hardware Event Count:UOPS_RETIRED.ALL_PS:Self
Module
Function (Full)
Source File
Module Path
Start Address
```

See Also

[Save and Format Command Line Reports](#)

command

Issue a command to a running collect action.

GUI Equivalent

Toolbar: Command

Window: Collection Log

Syntax

-command=<value>

Arguments

<value>	Description
mark	Place time-stamped mark in the data that can be referenced during analysis.
pause	Temporarily suspend the collection process. Use <code>-command resume</code> when you are ready to continue collection.
resume	Continue collection on a paused collection process.
status	Print collection status.
stop	Terminate a running collection process. Alternatively, use <code>ctrl + c</code> .

Modifiers

`result-dir, user-data-dir`

Description

This option performs one of the following actions on a running collect action: pause, resume, stop, status, or mark. Use with `result-dir` to specify the result directory for the running analysis.

Example

This example terminates the collect process in the default directory.

```
vtune -command stop
```

Run an unlimited duration collect process, which runs until stopped.

```
vtune -collect hotspots -knob sampling-mode=hw -duration unlimited -r ./results/r002hs
```

In another window, use `-command stop` to terminate the process running in the result directory `results/r002hs`, specified by the `-r` option (shortname of `result-dir`).

```
vtune -command stop -r ./results/r002hs
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

cpu-mask

Specify CPU(s) for a collect or collect-with action.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Advanced** section > **CPU mask** option

Syntax

`-cpu-mask=<cpu_mask1>,<cpu_mask3>-<cpu_mask5>...`

Arguments

CPU number or a range of numbers.

Default

ALL The hardware event-based sampling collector collects data on all CPUs in the system.

Actions Modified

[collect](#), [collect-with](#)

Description

This option specifies the CPU(s) for which data will be collected during hardware event-based sampling collection. Specify a list of comma-separated CPU IDs (with no spaces) and/or the range(s) of CPU IDs. A range is represented by a beginning and ending ID, separated by a dash.

Example

This example collects samples on four CPUs (1, 3, 4, and 5) for a Linux sample application.

```
vtune -collect hotspots -knob sampling-mode=hw -cpu-mask 1,3-5 -- /home/test/sample
```

See Also

csv-delimiter

Specify the delimiter for a tabular report.

Syntax

`-csv-delimiter=<delimiter>`

Arguments

<delimiter>

A character, keyword or string of characters to use as a delimiter when generating a tabular (CSV) report. Any character string may be used as a delimiter, but the most common values are one of these keywords: *comma* | *tab* | *semicolon* | *colon*

Default

comma

Actions Modified

The `report` action, used with the `format csv` action-option. To write the report to a file, also use the `report-output` option.

Description

Use this option to specify a delimiter when using `-format csv` to generate a report in CSV format.

Example

Generate a tabular hotspots report from the most recent result, using comma delimiters, and save the report as `MyReport.csv` in the current working directory.

```
vtune -R hotspots -format csv -csv-delimiter comma -report-output MyReport.csv
```

Sample output:

```
Module,Process,CPU Time  
worker3.so,main,10.735worker1.so,main,5.525worker2.so,main,3.612worker5.so,main,3.103worker4.so,main,1.679main,main,0.064
```

See Also

[Generate Command Line Reports](#)

[Save and Format Command Line Reports](#)

[vtune Command Syntax](#)

[vtune Actions](#)

cumulative-threshold-percent

Set a percent of the target CPU/Wait time to display only the hottest program units that exceed this threshold.

GUI Equivalent

[Window: Summary - Hotspots](#)

Syntax

`-cumulative-threshold-percent=<value>`

Arguments

`<value>` The percent of target CPU/Wait time consumed by the program units displayed.

Default

OFF all program units.

Actions Modified

`report`

Description

Use the `cumulative-threshold-percent` action-option to generate a performance detail report that focuses on program units that exceed the specified percentage of target CPU/Wait time. Functions below the specified threshold are filtered out, so your report includes just the hottest program units, and excludes those that are insignificant.

Example

Linux*: Generate a Performance Detail report from the `r001hs` Hotspots result that only includes functions that cumulatively account for 90% of target CPU time. Functions cumulatively representing less than 10% of target CPU time are excluded.

```
vtune -report perf-detail -r r001hs -cumulative-threshold-percent=90
```

Module	Function	CPU Time	Cumulative Percent
matrix	algorithm_2	3.136	70.415
matrix	algorithm_1	1.156	96.375

Windows*: Generate performance reports in `r001hs` and `r002hs` functions that account for 50% of the total difference. Positive and negative difference values are handled separately.

```
vtune -R perf -r r001hs -r r002hs -cumulative-threshold-percent=50
```

Module	Function	Result 1:CPU Time	Result 2:CPU Time	Difference:CPU Time
Cumulative Percent				
matrix.exe	algorithm_2	3.106	3.131	-0.025
100.000				
Module	Function	Result 1:CPU Time	Result 2:CPU Time	Difference:CPU Time
Cumulative Percent				
ntdll.dll	KiFastSystemCallRet	0.012	0	0.012
39.956				
ntdll.dll	NtWaitForSingleObject	0.113	0.110	0.003
				50.051

See Also

[Change Threshold Values](#)

[vtune Command Syntax](#)

[vtune Actions](#)

custom-collector

Launch an external collector to gather custom interval and counter statistics for your target in parallel with the VTune Profiler.

GUI Equivalent

Configure Analysis window > **WHAT** pane

Syntax

`-custom-collector=<string>`

Arguments

`<string>` Command line launching an external collection tool.

Actions Modified

`collect,collect-with`

Description

Your custom collector can be an application you analyze with the VTune Profiler or a collector that can be launched with the VTune Profiler.

Use the `-custom-collector` option to specify an external collector other than a target analysis application.

When you start a collection, the VTune Profiler does the following:

1. Launches the target application in the suspended mode.
2. Launches the custom collector in the attach (or system-wide) mode.
3. Switches the application to the active mode and starts profiling.

If your custom collector cannot be launched in the attach mode, the collection may produce incomplete data.

You can later import custom collection data (time intervals and counters) in a [CSV format](#) to the VTune Profiler result.

Example

This example runs Hotspots analysis in the default user-mode sampling mode and also launches an external script collecting custom statistics for the specified application:

Windows:

```
vtune -collect hotspots -custom-collector="python.exe C:\work\custom_collector.py" -- notepad.exe
```

Linux:

```
vtune -collect hotspots -custom-collector="python /home/my_collectors/custom_collector.py" -- my_app
```

This example runs VTune Profiler event-based sampling collector and also uses an external system collector to identify product environment variables:

Windows:

```
vtune -collect-with runsa -custom-collector="set | find \"AMPLXE\\"" -- notepad.exe
```

Linux:

```
vtune -collect-with runsa -custom-collector="set | find \"AMPLXE\\"" -- my_app
```

See Also

[Use a Custom Collector
in GUI](#)

[vtune Command Syntax](#)

[vtune Actions](#)

data-limit

Limit the amount of raw data (in MB) to be collected.

GUI Equivalent

Configure Analysis window > **WHAT** tab > **Result size from collection start, MB** option

Syntax

`-data-limit=<integer>`

Arguments

<code><integer></code>	Size of collected data (in MB)
------------------------------	--------------------------------

Default

<code><integer></code>	The default limit of collected data is set to 500 MB.
------------------------------	---

Actions Modified

`collect, collect-with`

Description

Use the `data-limit` action-option to limit the amount of raw data (in MB) to be collected. Zero data limit means no limit for data collection.

Alternate Options

<code>ring-buffer</code>	Limit the amount of raw data (in sec) to be collected.
--------------------------	--

Example

Perform a Hotspots analysis and limit the size of collected data to 200MB.

```
vtune -collect hotspots -data-limit=200 myApp
```

See Also

[Limit Data Collection](#)

`ring-buffer`
action-option

[vtune Command Syntax](#)

[vtune Actions](#)

discard-raw-data

Specify removal of raw collector data after finalization.

GUI Equivalent

Pane: Options - General

Syntax

-discard-raw-data

-no-discard-raw-data

Default

no-discard-raw-data Raw collector data is saved unless you specify the `discard-raw-data` option.

Actions Modified

`collect`, `collect-with`, `finalize`, `import`

Description

Use the `discard-raw-data` action-option if you want to remove raw collector data after the result is finalized. This makes the result files smaller.

NOTE

Keeping raw data enables result re-finalization. Do not use this option if you want to re-finalize the results in the future.

Example

This example runs the Hotspots analysis for the sample Linux* application, generates a default summary report, and removes raw collector data.

```
vtune -collect hotspots -discard-raw-data -- /home/test/sample
```

See Also

[Finalization](#)

[vtune Command Syntax](#)

[vtune Actions](#)

duration

Specify the duration for collection (in seconds).

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Advanced** section > **Automatically stop collection after** option

Syntax

`-duration=<value>`

Arguments

unlimited

Collection duration is unbounded.

Description

Use the `filter` option to include or exclude data from a report based on the specified `column_name`, the `=` or `!=` operator, and the value for that column.

To display a list of available filter attributes for a particular report, use `vtune -report <report_name> -r <result_dir> filter=?` option. If you do not specify a result directory, the latest result is used by default.

Examples

Generate a hotspots report on Linux* from the specified hotspots result that only includes data from the `appname` process. Data from other processes is excluded. This report is sent to `stdout`.

```
vtune -report hotspots -filter process=appname -result-dir /temp/test/r001hs
```

Generate a hotspots report from the most recent hotspots for all modules except `foo`, and save it as a text file in the specified directory on Windows*.

```
vtune -R hotspots -filter module!=foo -report-output C:\Test\report.txt
```

Obtain a list of filters available for the hw-events report for a Microarchitecture Exploration analysis result on Linux:

```
vtune -report hw-events -r /temp/test/r029ue/r029ue filter=?  
vtune: Using result path '/temp/test/r029ue/r029ue'
```

Available values for '`-filter`' option are:

```
basic-block : Basic Block  
basic-block-only : Basic Block  
function-only : Function  
function-mangled : Function  
module : Module  
module-path : Module Path  
process : Process  
thread-id : TID  
process-id : PID  
source-file : Source File  
source-line : Source Line  
source-file-path : Source File Path  
thread : Thread  
function-callstack : Function  
function-parent-callstack : Function  
callstack : Call Stack  
callstack-address : Call Stack  
no-attr-callstack : Call Stack  
cpuid : Logical Core  
address : Code Location  
function-start-address : Start Address  
function : Function  
source-function : Source Function  
package : Package
```

See Also

[Filter and Group Command Line Reports](#)

from CLI

[group-by](#)

action-option

[vtune Command Syntax](#)

[vtune Actions](#)

finalization-mode

Perform full finalization, fast finalization, deferred finalization or skip finalization.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Advanced** section > **Select finalization mode** option

Syntax

finalization-mode=<value>

Arguments

full	Perform full finalization on the target system.
fast	Reduce the number of loaded samples to speed up post-processing.
deferred	Only calculate the binary checksums for finalization on another machine.
none	Skip finalization.

Default

fast vtune performs fast finalization with the reduced number of loaded samples.

Actions Modified

`collect,collect-with,import,finalize`

Description

Use the `finalization-mode` option with the `collect`, `collect-with`, `import`, and `finalize` commands to define the finalization mode for the result.

Use the `full` finalization mode to perform the finalization on unchanged sampling data on the target system. This mode takes the most time and resources to complete, but produces the most accurate results.

Use the `fast` finalization mode to perform the finalization on the target system using algorithmically reduced sampling data. This greatly reduces the finalization time with a negligible impact on accuracy in most cases. If you discover inaccuracies in your finalization, you can always use the `finalize` action with the `full` finalization mode to re-finalize the result in `full` mode.

Use the `deferred` finalization mode to collect the sampling data and the binary checksums to perform the finalization on another machine. After data collection completes, you can finalize and open the analysis result on the host system. This mode may be useful for profiling applications on targets with limited computational resources, such as IoT devices, and finalizing the result later on the host machine.

NOTE

To have binaries successfully resolved during finalization, ensure that the host system has access to the binaries.

Use the `none` option to skip finalization entirely and to not collect the binary checksums. You can also finalize this result later, however, you may encounter certain limitations. For example, if the binaries on the target system have changed or have become unavailable since the sampling data collection, binary resolution may produce an inaccurate or missing result for the affected binary.

You can always repeat the finalization process in a different mode using the `finalize` action.

Example

The following command starts the Hotspots analysis on Windows and only calculates the binary checksums for finalization on another machine.

```
vtune -collect hotspots -knob sampling-mode=hw -finalization-mode=deferred -- C:\test\myApp.exe
```

See Also

[Intel® Xeon Phi™ Processor Targets](#)

[finalize](#)

option

[Run Command Line Analysis](#)

[Finalization](#)

[vtune Actions](#)

[vtune Command Syntax](#)

finalize

Perform symbol resolution to finalize or re-resolve a result.

GUI Equivalent

[Configure Analysis tab >](#)



Re-resolve button

Syntax

`-finalize -result-dir <PATH>`

`-I -result-dir <PATH>`

Arguments

The `finalize` action must be used with the `result-dir` action-option, which passes in the PATH/name of the result directory.

Default

Result finalization is performed automatically as part of the collection process.

Modifiers

`call-stack-mode, discard-raw-data, inline-mode, quiet, result-dir, search-dir, verbose`

Description

Use the `finalize` action when you need to finalize an un-finalized or improperly finalized result in the directory specified by the `result-dir` action-option. Use GUI tools to change search directories settings, or use the `search-dir` action-option with the `finalize` action to re-finalize the result and update symbol information.

Normally, finalization is performed automatically as part of a `collect` or `import` action. However, you may need to re-finalize a result if:

- Finalization was suppressed during collection or importation, for example when the `-finalization-mode=none` action-option was specified for a `collect` or `collect-with` action.
- Re-resolve a result that was not properly finalized because some of the source or symbol files were missing. When viewed in the GUI or reports, the word [Unknown] commonly appears.

Example

In this example, finalization is suppressed when generating a Hotspots analysis result `r001hs` on Linux*.

```
vtune -collect hotspots -finalization-mode=none -result-dir /tmp/test/r001hs
```

Finalize the unfinished Hotspots analysis result `r001hs` created previously.

```
vtune -finalize -result-dir /tmp/test/r001hs
```

Re-finalize a Hotspots analysis result `r004hs`, specifying search directories for symbol files.

```
vtune -finalize -search-dir /home/foo/system_modules -result-dir /tmp/test/r001hs
```

See Also

[Finalization](#)

[vtune Command Syntax](#)

[vtune Actions](#)

format

Specify output format for report.

Syntax

```
-format <value>
```

Arguments

<value>

Description

`text`

Text output format. File extension is `.txt`.

`csv`

CSV output format. File extension is `.csv`. Must be used with `csv-delimiter` option.

`xml`

XML output format. File extension is `.xml`. Available for *summary* report only.

`html`

HTML output format. File extension is `.html`. Available for *summary* report only.

Default

text

Actions Modified

report

Description

Use the `format` action-option to specify output format for report. To print to a file, use this with the `report-output` option. If you choose `csv`, you must also use the `csv-delimiter` option to specify the delimiter, such as comma.

NOTE

XML and HTML formats are available for the *summary* report only.

Example

Generate a Hotspots report in CSV file format using a comma delimiter and save it as `MyReport.csv` in the current working directory.

```
vtune -report hotspots -report-output MyReport.csv -format csv -csv-delimiter comma
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

group-by

Specify grouping in a report.

GUI Equivalent

Bottom-up tab > **Grouping** drop-down menu

Syntax

`-group-by <granularity1>,<granularity2>`

Arguments

Argument

`<granularity>`

Description

Grouping level that depends on the report type.

Default

Varies by report; `function` is the most common default.

Actions Modified

report

Description

Use the `group-by` action-option to group data in your report by your specified criteria. For multiple grouping levels, add arguments separated by commas (no spaces).

NOTE

For some reports (for example, top-down report) you can specify only a single grouping level.

To display a list of available groupings for a particular report, type: `vtune -report <report_name> -r <result_dir> group-by=?`. If you do not specify a result directory, the latest result is used by default.

NOTE

The `function value` groups the result data both by function and by module. To group just by the function, use `function-only`.

Example

Output a hotspots report for the latest result with data grouped by module:

```
vtune -report hotspots -group-by module
```

Output a hotspots report for the latest result with data grouped by thread and function:

```
vtune -report hotspots -group-by thread,function
```

Display all available hotspots report groupings for a Hotspots analysis result on Linux*:

```
vtune -R hotspots -r /temp/test/r029hs/r029hs group-by=?
vtune: Using result path '/temp/test/r029hs/r029hs'
```

Available values for '-group-by' option are:

```
basic-block : Basic Block
function : Function
function-mangled : Function
module : Module
module-path : Module Path
process : Process
thread-id : TID
process-id : PID
source-file : Source File
source-line : Source Line
source-file-path : Source File Path
thread : Thread
callstack : Call Stack
cpuid : Logical Core
address : Code Location
function-start-address : Start Address
source-function : Source Function
package : Package
source-function-stack : Source Function Stack
core : Physical Core
class : Class
cacheline : Cacheline
data-address : Data Address
tasks-and-interrupts : Task and Interrupt
context : Context
vcore : VCore
```

The following items can be specified only as the final grouping level: callstack, source-function-stack.

See Also

[Save and Format Command Line Reports](#)

[Filter and Group Command Line Reports from CLI](#)

[filter](#)

[Group and Filter Data](#)

from GUI

[vtune Command Syntax](#)

[vtune Actions](#)

help

Display brief explanations of command line arguments.

Syntax

```
-h, -help  
-help <action>  
-help collect <analysis_type>  
-help collect-with <collector_type>  
-help report <report_type>
```

Arguments

Argument	Description
None	List available action options for which help is available.
<action>	Output a help message for the specified action.

Description

Use the `help` action to access help for the `vtune` command. The help for each action includes explanations and usage examples.

Below is a list of available actions:

`help, version, import, finalize, report, collect, collect-with, command`

Examples

Display all available `vtune` actions.

```
vtune -help
```

Display help for the `collect` action, including all available options.

```
vtune -help collect
```

This example displays help for the threading analysis type, including knobs that are available on your system.

```
vtune -help collect threading
```

Display help for the hotspots report, including value for the group-by action-option.

```
vtune -help report hotspots
```

See Also

[Get Help](#)

[vtune Command Syntax](#)

[vtune Actions](#)

import

Import one or more collection data files/directories.

GUI Equivalent

VTune Profiler menu > **Import Result...**

Syntax

```
-import <PATH>
```

Arguments

A string containing the *PATH* of the data files to import. To import several files, make sure to use the `import` option for each path.

Modifiers

```
call-stack-mode, discard-raw-data, inline-mode, result-dir, search-dir, user-data-dir
```

Description

Use the `import` action to import one or more collection data files into the VTune Profiler. You may import the following formats:

- .tb6 or .tb7 with event-based sampling data. To import the files, use the `-result-dir` option and specify the name for a new directory you want to create for the imported data. If you do not use the `-result-dir` option, the VTune Profiler creates a new directory with the default name.
- .perf files with event-based sampling data collected by Linux* Perf tool. To ensure accurate data representation in the VTune Profiler, make sure to run the Perf collection with the predefined command line options:

- For application analysis:

```
perf record -o <trace_file_name>.perf --call-graph dwarf -e cpu-cycles,instructions <application_to_launch>
```

- For process analysis:

```
perf record -o <trace_file_name>.perf --call-graph dwarf -e cpu-cycles,instructions <application_to_launch> -p <PID> sleep 15
```

where the `-e` option is used to specify a list of events to collect as `-e <list of events>`; `--call-graph` option (optional) configures samples to be collected together with the thread call stack at the moment a sample is taken. See Linux Perf documentation on possible call stack collection options (for example, `dwarf`) and its availability in different OS kernel versions.

NOTE

The Linux kernel exposes Perf API to the Perf tool starting from version 2.6.31. Any attempts to run the Perf tool on kernels prior to this version lead to undefined results or even crashes. See Linux Perf documentation for more details.

- To import a [csv file](#), use the `-result-dir` option and specify the name of an existing directory of the result that was collected by the VTune Profiler in parallel with the [external data collection](#). VTune Profiler adds the externally collected statistics to the result and provides integrated data in the **Timeline** pane.

NOTE

Importing a [csv](#) file to the VTune Profiler result does not affect symbol resolution in the result. For example, you can safely import a [csv](#) file to a result located on a system where module and debug information is not available.

- *.pwr processed Intel SoC Watch files with [energy analysis](#) data

Example

This example imports the `sample_data.tb7` file into a VTune Profiler project and creates the result directory `r000hs`:

```
vtune -import sample_data.tb7 -result-dir r000hs
```

This example imports a trace file collected with the Linux Perf tool into a VTune Profiler project and creates a default result directory `r000` (since no result directory is specified from the command line):

```
vtune -import perf_trace.perf
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

[Import Results and Traces into VTune Profiler GUI](#)
in GUI

inline-mode

Exclude/include inline functions in the stack.

GUI Equivalent

Toolbar: **Filter > Inline Mode menu**

Syntax

`-inline-mode off | on`

Default

`on` Inline functions are included in the stack.

Actions Modified

`collect, finalize, import, report`

Description

Use `inline-mode off` with the `collect`, `finalize` or `import` actions if you want to exclude inline functions from the stack in results. You can also use this with the `report` action to exclude inline functions from reports.

By default, this option is enabled so that performance details for all inline functions used in the application are included in the stack in results and reports.

NOTE

This option is available if information about inline functions is available in debug information generated by compilers. See [View Data on Inline Functions](#) for supported compilers and options.

Example

Generate a hotspots report with inline mode disabled.

```
vtune -report hotspots -inline-mode off
```

See Also

[View Data on Inline Functions](#)

from GUI

[vtune Command Syntax](#)

[vtune Actions](#)

knob

Set configuration options for the specified analysis type or collector type.

GUI Equivalent

Configure Analysis window > **HOW** pane

Syntax

`-knob | -k <knob-name>=<knob-value>`

Arguments

knob-name

An analysis type or collector type may have one or more configuration options (*knobs*) that provide additional instructions for performing the specified type of analysis. To use a knob, you must specify the knob name and knob value.

Multiple *knob* options are allowed and can be followed by additional action-options, as well as global-options, if needed.

knob-value

There are values available for each knob. In most cases this is a Boolean value, so for Boolean knobs, specify `<knob-name>=true` to enable the knob.

NOTE

Knob behavior may vary depending on the analysis type or collector type.

<knob-name>	Description
accurate-cpu-time-detection=true false (Windows only)	Collect more accurate CPU time data. This option requires additional disk space and post-processing time. Administrator privileges are required. Supported analysis: runss
Default: true	
analyze-loops=true false	Extend loop analysis to collect advanced loops information such as instruction set usage and display analysis results by loops and functions.
Default: false	Supported analysis: runss, runsa
analyze-mem-objects=true false	Enable the instrumentation of memory allocation/de-allocation and map hardware events to memory objects. This option is supported only for Linux targets which run on the Intel microarchitectures code named Haswell (or newer).
Default: false	Supported analysis: memory-access
analyze-openmp=true false	Instrument the OpenMP* runtimes in your application to group performance data by regions/work-sharing constructs and detect inefficiencies such as imbalance, lock contention, or overhead on performing scheduling, reduction, and atomic operations. Using this option may cause higher overhead and increase the result size.
Default: true for the HPC Performance Characterization analysis; false for other analysis types.	Supported analysis: hotspots, threading, hpc-performance, memory-access, uarch-exploration, runsa
analyze-persistent-memory=true false	Collect performance information for Intel® Optane™ Persistent Memory modules.
Default: false	Supported analysis: platform-profiler
analyze-power-usage=true false	Collect information about energy consumed by CPU, DRAM, and discrete GPU.
Default: false	Supported analysis: gpu-hotspots,gpu-offload
analyze-throttling-reasons=true false	Collect information about factors that cause the CPU to throttle .
Default: false	Supported analysis: system-overview
analyze-xelink-usage=true false	Collect information about data traffic between GPU interconnects (Xe Link) in multi-GPU analysis.
Default: false	Supported analysis: gpu-hotspots,gpu-offload
atrace-config=<event>	Collect Android framework events from Systrace*.
	Supported analysis: runsa

<knob-name>	Description
Available events are gfx, input, view, webview, wm, am, audio, video, camera, hal, res, dalvik.	
characterization-mode=overview global-local-accesses compute-extended full-compute instruction-count Default: overview	<p>Monitor the Render and GPGPU engine usage (Intel Graphics only), identify which parts of the engine are loaded, and correlate GPU and CPU data.</p> <p>The Characterization mode uses platform-specific presets of the GPU metrics. All presets, except for the <code>instruction-count</code>, collect data about execution units (EUs) activity: EU Array Active, EU Array Stalled, EU Array Idle, Computing Threads Started, and Core Frequency; and each one introduces additional metrics:</p> <ul style="list-style-type: none"> • <code>overview</code> metric set includes additional metrics that track general GPU memory accesses such as Memory Read/Write Bandwidth, GPU L3 Misses, Sampler Busy, Sampler Is Bottleneck, and GPU Memory Texture Read Bandwidth. These metrics can be useful for both graphics and compute-intensive applications. • <code>global-local-accesses</code> metric group includes additional metrics that distinguish accessing different types of data on a GPU: Untyped Memory Read/Write Bandwidth, Typed Memory Read/Write Transactions, SLM Read/Write Bandwidth, Render/GPGPU Command Streamer Loaded, and GPU EU Array Usage. These metrics are useful for compute-intensive workloads on the GPU. • <code>compute-extended</code> metric group includes additional metrics targeted only for GPU analysis on the Intel processor code name Broadwell and higher. For other systems, this preset is not available. • <code>full-compute</code> metric group is a combination of the <code>overview</code> and <code>global-local-accesses</code> event sets. • <code>instruction-count</code> metric group counts the execution frequency of specific classes of instructions. <p>Supported analysis: gpu-hotspots, graphics-rendering, runsa</p>
chipset-event-config="event1, event2 , .."	Specify a comma-separated list of Android chipset events (up to 5 events) to monitor with the hardware event-based sampling collector.
source-analysis=bb-latency mem-latency Default value: bb-latency	<p>Supported analysis: runsa</p> <p>Collect data on performance-critical basic blocks and issues caused by memory accesses in the GPU kernels. Choose one of the following modes:</p> <ul style="list-style-type: none"> • <code>bb-latency</code> mode helps you identify issues caused by algorithm inefficiencies. In this mode, VTune Profiler measures the execution time of all basic blocks. Basic block is a straight-line code sequence that has a single entry point at the beginning of the sequence and a single exit point at the end of this sequence. During post-processing, VTune Profiler calculates the execution time for each instruction in the basic block. So, this mode helps understand which operations are more expensive.

<knob-name>	Description
	<ul style="list-style-type: none"> mem-latency mode helps identify latency issues caused by memory accesses. In this mode, VTune Profiler profiles memory read/synchronization instructions to estimate their impact on the kernel execution time. Consider using this option, if you ran the <code>gpu-hotspots</code> analysis in the Characterization mode, identified that the GPU kernel is throughput or memory-bound, and want to explore which memory read/synchronization instructions from the same basic block take more time.
	Supported analysis: <code>gpu-hotspots</code>
collect-bad-speculation=true false	Collect the minimum set of data required to compute top-level metrics and all Bad Speculation sub-metrics.
	Supported analysis: <code>uarch-exploration, runsa</code>
Default value: true	
collect-core-bound=true false	Collect the minimum set of data required to compute top-level metrics and all Core Bound sub-metrics.
	Supported analysis: <code>uarch-exploration, runsa</code>
Default: false	
collect-frontend-bound=true false	Collect the minimum set of data required to compute top-level metrics and all Front-End Bound sub-metrics.
	Supported analysis: <code>uarch-exploration, runsa</code>
Default value: true	
collect-cpu-gpu-bandwidth=true false	Collect DRAM bandwidth data for all hosts. Additionally, collect PCIe bandwidth for supported server hosts (Intel® micro-architectures code named Ice Lake and Sapphire Rapids). To view collected data in GUI, enable the Analyze CPU host-GPU bandwidth option.
Default: false	
	Supported analysis: <code>gpu-offload</code>
collect-cpu-gpu-pci-bandwidth=true false	Collect PCIe bandwidth for supported server hosts (Intel® micro-architectures code named Ice Lake and Sapphire Rapids). This knob is available for custom analyses only. To view collected data in GUI, enable the Analyze CPU host-GPU bandwidth option.
Default: false	
	Supported analysis: <code>runsa</code>
collect-io-waits=true false	Analyze the percentage of time each thread and CPU spends in I/O wait state.
Default: false	
	Supported analysis: <code>runsa</code>
collect-memory-bandwidth=true false	Collect data to identify where your application is generating significant bandwidth to DRAM. To view collected data in GUI, enable the Analyze memory bandwidth option.
Default: depends on analysis type	
	Supported analysis: <code>performance-snapshot, uarch-exploration, hpc-performance, gpu-hotspots, runsa</code>
collect-memory-bound=true false	Collect the minimum set of data required to compute top-level metrics and all Memory Bound sub-metrics.
Default value: true	
	Supported analysis: <code>uarch-exploration, hpc-performance</code>

<knob-name>	Description
collect-programming-api=true false Default for gpu-hotspots: true, for runss: false.	Analyze execution of SYCL apps, OpenCL™ kernels and Intel® Media SDK programs on Intel HD Graphics and Intel® Iris® Graphics. This option may affect the performance of your application on the CPU side.
collect-retiring=true false Default value: true	Supported analysis: gpu-hotspots, gpu-offload, runsa Collect the minimum set of data required to compute top-level metrics and all Retiring sub-metrics. Supported analysis: uarch-exploration, runsa
collecting-mode=hw-tracing hw-tracing Default value: hw-sampling	Specify the system-wide collection mode to either explore CPU, GPU, and I/O resources utilization with the default event-based sampling mode, or enable the low-overhead hardware tracing and identify a root cause of latency issues. Supported analysis: system-overview, runsa
computing-task-of-interest=computing_task_name[#start_idx#step#stop_idx] Default value: #1#1#4294967295	Specify a comma-separated list of GPU computing task names and invocations. Use a search string, if necessary (* and . are supported). Invocations happen in this format: computing_task_name[#start_idx#step#stop_idx] Default value:#1#1#4294967295 <ul style="list-style-type: none">• <i>computing_task_name</i> is the name of the GPU computing task (default value is *);• <i>start_idx</i> is the number of the first invocation to be profiled (default value is 1);• <i>step_idx</i> is the number of the step idx invocation (default value is 1);• <i>stop_idx</i> is the number of the last invocation to be profiled (default value is 4294967295, UNIT_MAX) Supported analysis: gpu-hotspots, runsa
counting-mode=true false Default: false	Choose between collecting detailed context data for each PMU event (such as code or hardware context) or the counts of events. Counting mode introduces less overhead but gives less information. Supported analysis: runsa
cpu-samples-mode=off stack nostack Default: false	Enable to periodically sample the application. Samples can be collected with or without stacks. Supported analysis: runss
dpdk=true false Default: false	Profile DPDK IO API. Supported analysis: io
dram-bandwidth-limits=true false Default: true for the HPC Performance Characterization and Microarchitecture Exploration analysis with	Evaluate maximum achievable local DRAM bandwidth before the collection starts. This data is used to scale bandwidth metrics on the timeline and calculate thresholds. Supported analysis: performance-snapshot, memory-access, uarch-exploration, hpc-performance, runsa

<knob-name>	Description
collect-memory-bandwidth knob enabled; true for the Memory Access and Microarchitecture Exploration analysis.	
enable-characterization-insights=true false	Get additional performance insights such as the efficiency of hardware usage, and learn next steps. Supported analysis: gpu-offload
enable-context-switches=true false Default: false	Analyze detailed scheduling layout for all threads <i>in your application</i> , explore time spent on a context switch and identify the nature of context switches for a thread (preemption or synchronization). Supported analysis: runsa
enable-driverless-collection=true false Default: false	Enable driverless Linux Perf collection when possible. Supported analysis: runsa
enable-gpu-usage=true false Default: false	Analyze frame rate and usage of Intel HD Graphics and Intel® Iris® Graphics engines and identify whether your application is GPU or CPU bound. Supported analysis: runss, runsa
enable-interrupt-collection=true false Default: false	Collect interrupt events that alter a normal execution flow of a program. Such events can be generated by hardware devices or by CPUs. Use this data to identify slow interrupts that affect your code performance. Supported analysis: system-overview.
enable-parallel-fs-collection=true false Default: false	Analyze Lustre* file system performance statistics, including Bandwidth, Package Rate, Average Packet Size, and others. Supported analysis: runsa
enable-stack-collection=true false Default: false	Enable Hardware Event-based Sampling Collection with Stacks . Supported analysis: hotspots, hpc-performance, gpu-offload, runsa
enable-system-cswitch=true false Default: false	Analyze detailed scheduling layout for all threads <i>on the system</i> and identify the nature of context switches for a thread (preemption or synchronization). Supported analysis: runsa
enable-thread-affinity=true false Default: false	Analyze thread pinning to sockets, physical cores, and logical cores. Identify incorrect affinity that utilizes logical cores instead of physical cores and contributes to poor physical CPU utilization.

<knob-name>	Description
NOTE	
enable-user-sync=true false	Affinity information is collected at the end of the thread lifetime, so the resulting data may not show the whole issue for dynamic affinity that is changed during the thread lifetime.
enable-user-sync=true false Default: false	Collect synchronization data via the User-Defined Synchronization API . Supported analysis: threading, runss
enable-user-tasks=true false Default: false	Analyze tasks, events and counters specified in your application via the Task API . This option causes higher overhead and increases result size. Supported analysis: hotspots, threading, uarch-exploration, runss, runsa
event-config=<event_name1>,<event_name2>,...	Configure PMU events to collect with the hardware event-based sampling collector. Multiple events can be specified as a comma-separated list (no spaces).
NOTE	
To display a list of events available on the target PMU, enter:	vtune -collect-with runsa -knob event-config=? <target>
The command returns names and short descriptions of available events. For more information on the events, use Intel Processor Events Reference .	
Supported analysis: runsa	
event-mode=all user os Default: all	Limit event-based sampling collection to OS or USER mode. Supported analysis: runsa
ftrace-config=<event_name> Available events are freq, idle, sched, disk, filesystem, irq, kvm, workq, softirq, sync.	Collect Linux Ftrace* framework events. <ul style="list-style-type: none">• This option is supported for Linux target systems only.• On some systems, Linux Ftrace events collection is possible only for the root user. Supported analysis: runsa, runss
Default for Linux targets: sched,freq,idle,workq,irq,softirq	
Default for Android targets: sched,freq,idle,workq,filesystem,irq,softirq,sync,disk	

<knob-name>	Description
gpu-sampling-interval=<number> between 0.1 and 1000ms	Specify an interval between GPU samples (in milliseconds).
Default: 1.	Supported analysis: gpu-hotspots, graphics-rendering, runss, runsa
io-mode=off stack nostack	Enable to identify where threads are waiting or compute thread concurrency. The collector instruments APIs, which causes higher overhead and increases result size.
Default: off	Supported analysis: runss, runsa
ipt-regions-to-load=<number> between 10 and 5000	Specify the maximum number (10-5000) of code regions to load for detailed analysis.
Default: 1000	Supported analysis: anomaly-detection
kernel-stack=true false	Profile system disk IO API.
Default: true	Supported analysis: io
max-region-duration=<number> between 0.001 and 1000 ms	Specify the maximum duration (0.001-1000ms) of analysis per code region.
Default: 100	Supported analysis: anomaly-detection
mem-object-size-min-thres=<number>	Specify a minimal size of memory allocations to analyze. This option helps reduce runtime overhead of the instrumentation.
Default: 1024 bytes	This option is supported only for Linux targets which run on Intel microarchitectures code named Haswell (or later).
metrics_set=NOC	Supported analysis: memory-access
Default: NOC	Specify the type of metrics set to collect.
mrte-type=java,dotnet java,dotnet,python python	Specify a type of managed runtime to analyze. Available values: combined .NET* and Java* analysis, combined Java, .NET and Python* analysis, and Python only.
Default: java,dotnet	Supported analysis: runss, runsa
no-altstack=true false	Disable using alternative stacks for signal handlers. Consider this option for profiling standard Python 3 code on Linux.
Default: false	Supported analysis: runss
pmu-collection-mode=detailed summary	Choose the detailed sampling-based collection mode to view data breakdown per function and other hotspots. Use the summary counting-based mode for an overview of the whole profiling run. This mode has a lower collection overhead and fast post-processing time.
Default: detailed	Supported analysis: uarch-exploration

<knob-name>	Description
profiling-mode=characterization (default) , code-level-analysis, query-based, time-based	<p>Select a profiling mode for these analyses:</p> <ul style="list-style-type: none"> GPU Compute/Media Hotspots analysis: Characterize GPU performance issues based on GPU hardware metric presets Custom analysis: Enable a source analysis to identify basic blocks latency due to algorithm inefficiencies or memory latency due to memory access issues NPU Exploration analysis: Select a data collection mode (time-based or query-based) <p>Supported analysis: gpu-hotspots, runsa, npu</p>
sampling-interval=<number> For user-mode sampling and tracing types: a number (in milliseconds) between 1 and 1000. Default: 10 For hardware event-based sampling types: a number (in milliseconds) between 0.01 and 1000. Default: 1. For NPU exploration: a number (in milliseconds) between 0.1 and 1000.	<p>Specify a sampling interval (in milliseconds) between CPU samples. For NPU Exploration analysis, specify the sampling interval for data collection in time-based mode.</p> <p>Supported analysis: hotspots, runss, threading, , runsa, system-overview, memory-access, hpc-performance, npu</p>
sampling-mode=sw hw Default: sw	<p>Specify a profiling mode.</p> <p>Use <code>sw</code> to identify CPU hotspots and explore a call flow of your program. This mode does not require sampling drivers to be installed but incurs more collection overhead.</p> <p>Use <code>hw</code> to identify application hotspots based on such basic hardware events as Clockticks and Instructions Retired. This is a low-overhead collection mode but it requires the sampling driver to be installed on your system.</p> <p>Supported analysis: hotspots, threading</p>
signals-mode=off objects stack nostack Default: off	<p>Enable to view synchronization transitions in the timeline and signalling call stacks for associated waits. The collector instruments signalling APIs, which causes higher overhead and increases result size.</p> <p>Supported analysis: runss</p>
spdk=true false Default: false	<p>Profile SPDK IO API.</p> <p>Supported analysis: io</p>
stack-size=<number> A number between 0 and 2147483647. Default is 0 (unlimited stack size).	<p>Reduce the collection overhead and limit the stack size (in bytes) processed by the VTune Profiler.</p> <p>Supported analysis: runsa</p>

<knob-name>	Description
stack-stitching=true false	For Intel® oneAPI Threading Building Blocks(oneTBB)-based applications, restructure the call flow to attach stacks to a point introducing a parallel workload.
Default: true	Supported analysis: runss
stack-type=software lbr	Choose between software stack and hardware LBR-based stack types. Software stacks have no depth limitations and provide more data while hardware stacks introduce less overhead. Typically, software stack type is recommended unless the collection overhead becomes significant. Note that hardware LBR stack type may not be available on all platforms.
Default: software	Supported analysis: runsa
stackwalk-mode=online offline	Choose between online (during collection) and offline (after collection) modes to analyze stacks. Offline mode reduces analysis overhead and is typically recommended.
Default: offline	Supported analysis: runss
target-gpu=<domain:bus:device.function>	Select a target GPU for profiling when you have multiple GPUs connected to your system. If unset, VTune Profiler selects the newest GPU architecture it can detect.
Default: The newest GPU architecture that VTune Profiler can detect	Example: target-gpu=0:0:2.0
uncore-sampling-interval=<number>	Supported analysis: gpu-offload, gpu-hotspots
For hardware event-based sampling types: a number (in milliseconds) between 1 and 1000. Default: 10.	Specify an interval (in milliseconds) between uncore event samples.
waits-mode=off stack nostack	Supported analysis: runsa
Default: off	Enable to identify where threads are waiting or compute thread concurrency. The collector instruments APIs, which causes higher overhead and increases result size.
	Supported analysis: runss

Actions Modified

`collect`, `collect-with`

Description

Use the knob action-option to configure knob settings for a `collect` (predefined analysis types) or `collect-with` (custom analysis types) action where the analysis type supports one or more knobs. Each analysis type or collector type supports a specific set of knobs, and each knob requires a value. In most cases the knob value is Boolean, so you would use `True` to enable the knob.

To see all knobs available for a predefined analysis type:

```
vtune -help collect <analysis_type>
```

To see knobs for a custom analysis type:

```
vtune -help collect-with <analysis_type>
```

Example

This example returns a list of knobs for the Threading analysis type:

```
vtune -help collect threading
```

This example runs a custom event-based sampling data collection on an Android system enabling collection of Android framework and chipset events.

```
vtune -collect-with runss -target-system=android -knob sampling-interval=2 -knob cpu-samples-mode=stack -knob ftrace-config=gfx,dalvik -knob chipset-event-config="GMCH_PARTIAL_WR_DRAM.ANY,GMCH_CORE_CLKS" --target-process com.intel.tbb.example.tachyon
```

This example configures and runs a custom event-based sampling data collection with the stack size limited to 8192 bytes:

```
vtune -collect-with runsa -knob enable-stack-collection=true -knob stack-size=8192 -knob enable-call-counts=true -knob event-config=CPU_CLK_UNHALTED.REF_TSC:sa=1800000,CPU_CLK_UNHALTED
```

See Also

[Custom Analysis Options
in GUI](#)

[Analyze Performance
from GUI](#)

[API Support](#)

[vtune Command Syntax](#)

[vtune Actions](#)

kvm-guest-kallsyms

Specify a local path to the /proc/kallsyms file copied from the guest system.

GUI Equivalent

Guest OS /proc/kallsyms option in the [WHAT](#) pane

Syntax

```
-kvm-guest-kallsyms=<string>
```

Arguments

A string containing the PATH, for example: /home/<user>/[guest]/<kvm kallsyms path>.

Actions Modified

`collect, collect-with`

Description

Specify a local path to the /proc/kallsyms file copied from the guest OS for proper [finalization](#).

Example

Enable a custom hardware event-based sampling collection for the KVM guest OS and collect irq, softirq, workq, and kvm FTrace* events:

```
vtune -collect-with runsa -knob event-
config=CPU_CLK_UNHALTED.REF_TSC:sa=3500000,CPU_CLK_UNHALTED.THREAD:sa=3500000,INST_RETIREDA.NY:sa
=3500000 -knob enable-stack-collection=true -knob ftrace-config=irq,softirq,workq,kvm -analyze-
kvm-guest -kvm-guest-kallsyms=/home/vtune/[guest]/kvm.kallsyms -kvm-guest-modules=/home/vtune/
[guest]/kvm.modules --search-dir sym:p=/home/vtune/ --target-pid 9791
```

See Also

[Profile KVM Kernel and User Space on the KVM System](#)

from GUI

[Targets in Virtualized Environments](#)

[Profile Targets on a KVM* Guest System](#)

[knob](#)

[ftrace-config](#)

[kvm-guest-modules](#)

[analyze-kvm-guest](#)

[vtune Actions](#)

[vtune Command Syntax](#)

kvm-guest-modules

Specify a local path to the /proc/modules file copied from the guest system.

GUI Equivalent

Guest OS /proc/modules option in the [WHAT](#) pane

Syntax

`-kvm-guest-modules=<string>`

Arguments

A string containing the PATH, for example: `/home/<user>/<guest mount path>/<kvm modules path>`.

Actions Modified

`collect, collect-with`

Description

Specify a local path to the /proc/modules file copied from the guest OS for proper [finalization](#).

Example

Enable a custom hardware event-based sampling collection for the KVM guest OS mounted to the /home/vtune/guest_mount directory:

```
vtune -collect-with runsa -knob event-
config=CPU_CLK_UNHALTED.REF_TSC:sa=3500000,CPU_CLK_UNHALTED.THREAD:sa=3500000,INST_RETIREDA.NY:sa
=3500000 -knob enable-stack-collection=true -knob ftrace-config=irq,softirq,workq,kvm-analyze-
kvm-guest -kvm-guest-kallsyms=/home/vtune/guest_mount/kvm.kallsyms -kvm-guest-modules=/home/
vtune/guest_mount/kvm.modules --search-dir sym:p=/home/vtune/ --target-pid 9791
```

See Also

[Profile KVM Kernel and User Space on the KVM System](#)

from GUI

[Targets in Virtualized Environments](#)

[Profile Targets on a KVM* Guest System](#)

[knob](#)

[ftrace-config](#)

[analyze-kvm-guest](#)

[kvm-guest-kallsyms](#)

[vtune Actions](#)

[vtune Command Syntax](#)

limit

Set the number of top items to include in a report.

Syntax

`-limit <value>`

Arguments

`<value>` Number of items to output

Default

Unlimited lines in output

Actions Modified

`report`

Description

Use the `limit` action-option when you only want to include the top items in a report, and specify the number of items (program units) to include.

Example

Output a Hotspots report on the ten modules with the highest CPU time values.

```
vtune -report hotspots -limit 10 -group-by module
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

loop-mode

Show or hide loops in the stack.

IDE Equivalent

Toolbar: Filter > **Loop Mode** drop-down menu

Syntax

loop-mode=<value>

Arguments

loop-only	Display loops as regular nodes in the tree. Loop name consists of:
	<ul style="list-style-type: none"> • start address of the loop • number of the code line where this loop is created • name of the function where this loop is created
loop-and-function	Display both loops and functions as separate nodes.
function-only	Display data by function with no loop information (default mode).

Default

function-only vtune reports show no loop data.

Actions Modified

[report](#)

Description

Use the `loop-mode` option when performing data collection, finalization or importation, to set loop view for the result or report. You can also use this option with the `report` action to override the project-level setting for viewing a hierarchy of the loops in your application call tree.

Example

This command displays the data collected during the Hotspots analysis in the callstack report that is filtered to show loops only:

```
vtune -R callstacks -loop-mode=loop-only
```

Function	Function Stack	Module
CPU Time:Self		
[Outside any loop]		
[Unknown]	0.009	
[Loop@0x7dea03b7 in func@0x7dea0392]		
ntdll.dll	0.002	
	[Loop@0x7dea03a6 in func@0x7dea0392]	

ntdll.dll	0.002	[Outside any loop]
[Unknown]	0	
[Loop@0x1400147f0 in func@0x140014782]		
mfeapfk.sys	0.001	[Outside any loop]
[Unknown]	0.001	
[Loop@0x14001a111 in func@0x14001a0c0]		
mfeapfk.sys	0.001	[Loop@0x14001a100 in func@0x14001a0c0]
mfeapfk.sys	0.001	[Outside any loop]
[Unknown]	0	
[Loop@0x1402d0329 in func@0x1402d02af]		
ntoskrnl.exe	0.001	[Outside any loop]
[Unknown]	0.001	

See Also

[Analyze Loops](#)

[Run Command Line Analysis](#)

mrte-mode

Specify managed profiling mode for Java, Python*, Go*, .NET*, and Windows* Store applications.*

GUI Equivalent

Configure Analysis window >**WHAT** pane > **Managed code profiling mode** option

Syntax

`-mrte-mode <value>`

Arguments

`<value>`

Profiling mode for the managed code. Possible values are:

- `auto` - Automatically detects the type of target executable, managed or native, and switches to the corresponding mode.
- `native` - Collects data on native code only, does not attribute data to managed source.
- `mixed` - Collects data on both native and managed code, and attributes data to managed source where appropriate. Consider using this option when analyzing a native executable that makes calls to the managed code.
- `managed` - Collects data on both native and managed code, resolves samples attributed to native code, attributes data to managed source only. The call stack in the analysis result displays data for managed code only.

Default

`auto` Mode is set automatically based on detected target type (executable, managed or native).

Actions Modified

`collect, collect-with`

Description

Use the `mrte-mode` option to specify one of the following Microsoft* run-time environment profiling modes: `auto`, `native`, `mixed`, or `managed`.

Example

Collect hotspots data on native code only for a Windows sample application:

```
vtune -collect hotspots -mrte-mode native -- C:\test\sample.exe
```

See Also

[Managed Code Targets
in GUI](#)

[Java* Code Analysis from the Command Line](#)

[vtune Command Syntax](#)

[vtune Actions](#)

no-follow-child

Specify whether child processes are included in collection results.

GUI Equivalent

Configure Analysis window > **WHAT** pane> **Analyze child processes** options

Syntax

`-no-follow-child`

`-follow-child`

Default

The default is `-follow-child`, so that child processes are included in the `collect` action.

Actions Modified

`collect`

Description

Use the `no-follow-child` action-option when you want to exclude child processes from `collect` action data collection and analysis. This option is recommended when profiling an application launched by a script.

Example

In this example, only the `myApp` Linux* application will be profiled. No information will be collected about any child processes initiated by `myApp`.

```
vtune -collect hotspots -no-follow-child myApp -- /home/test/sample
```

See Also

[Run Command Line Analysis](#)

[vtune Command Syntax](#)

[vtune Actions](#)

no-summary

SUPPRESS SUMMARY REPORT GENERATION.

GUI Equivalent

Window: Summary - Hotspots

Syntax

`-no-summary`

`-summary`

Default

A Summary report is generated and sent to `stdout` after performing a `collect` or `collect-with` action.

Actions Modified

`collect`

Description

When performing certain actions, such as `collect` or `collect-with`, a Summary is generated and sent to `stdout` by default. To suppress this, use the `no-summary` option when performing data collection. This can save time and system resources when analyzing large applications.

Example

This example runs the Hotspots analysis for the sample application without generating a summary report.

On Windows*:

```
vtune -collect hotspots -no-summary -- C:\test\sample.exe
```

On Linux*:

```
vtune -collect hotspots -no-summary -- /home/test/sample
```

See Also

[report](#)

option

[Summary Report](#)

[vtune Command Syntax](#)

[vtune Actions](#)

Generate Command Line Reports

no-unplugged-mode

Enable collection from an unplugged Android device to exclude ADB connection and power impact on the results .*

GUI Equivalent

Analyze detached device option in the [WHAT: Analysis Target](#) pane

Syntax

```
-no-unplugged-mode  
-unplugged-mode
```

Actions Modified

`collect, collect-with`

Description

The `unplugged-mode` option enables collection on an unplugged Android device to exclude ADB connection and power supply impact on the results. When this option is used, you configure and launch an analysis from the host but data collection starts after disconnecting the device from the USB cable or a network. Collection results are automatically transferred to the host as soon as you plug in the device back.

Example

This command configures Hotspots analysis for the application on an Android system that will be launched after disconnecting the device from the USB cable or a network:

```
host>./vtune --collect hotspots --target-system=android -unplugged-mode -r quadrant_r@@@ --  
target-process com.intel.fluid
```

See Also

[Android* Target Analysis from the Command Line](#)

[vtune Command Syntax](#)

[vtune Actions](#)

quiet

Limit the amount of information displayed by vtune.

Syntax

```
-quiet  
-q
```

Default

OFF	Standard amount of information is output.
-----	---

Actions Modified

`collect, finalize, report, version`

Description

Use the `quiet` option to limit the amount of information displayed by `vtune`. Only error, fatal error, and warning messages are displayed when this option is used.

Example

This example suppresses unimportant messages while running the Hotspots analysis of the Linux* sample application and generating the default summary report.

```
vtune -collect hotspots -quiet -- /home/test/sample
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

report

Generate a specified type of report from an analysis result.

GUI Equivalent

[Viewpoint](#)

Syntax

```
-report <report_name>  
-R <report_name>
```

Arguments

Argument	Description
<code><report_name></code>	Type of report to create.
<code>affinity</code>	Display binding of a thread to a range of sockets, physical, and logical cores.
<code>callstacks</code>	Report full stack data for each hotspot function; identify the impact of each stack on the function CPU or Wait time. You can use the <code>group-by</code> or <code>filter</code> options to sort the data by: <ul style="list-style-type: none">• callstack• function• function-callstack
<code>exec-query</code>	
<code>gprof-cc</code>	Report a call tree with the time (CPU and Wait time, if available) spent in each function and its children.
<code>hotspots</code>	Display collected performance metrics according to the selected analysis type and identify program units that took the most CPU time (hotspots).

hw-events	Display the total number of hardware events.
platform-power-analysis	Display CPU sleep time, wake-up reasons and CPU frequency scaling time.
summary	Report on the overall performance of your target .
timeline	Display metric data over time and distributed over time intervals.
top-down	Report call sequences (stacks) detected during collection phase, starting from the application root (usually, the main() function). Use this report to see the impact of program units together with their callees.
vectspots	Display statistics that help identify code regions to tracing on a HW simulator.

Modifiers

[call-stack-mode](#), [csv-delimiter](#), [cumulative-threshold-percent](#), [discard-raw-data](#), [filter](#), [format](#), [group-by](#), [inline-mode](#), [limit](#), [quiet](#), [report-output](#), [result-dir](#), [search-dir](#), [source-search-dir](#), [source-object](#), [verbose](#), [time-filter](#), [loop-mode](#), [column](#)

Description

Use the `report` action to generate a report from an existing result. The report type must be compatible with the analysis type used in the collection.

By default, your report is written to `stdout`. If you want to save it to a file, use the `report-output` action-option.

Both short names and long names are case-sensitive. For example, `-R` is the short name of the `report` action, and `-r` is the short name of the `result-dir` action-option.

NOTE

To get the list of available report types, use the `vtune -help report` command.

To display help for a specific report type, use `vtune -help report <report_name>`, where `<report_name>` is the type of report that you want to create.

Example

In this pair of examples, a `collect` action is used to perform a hotspots analysis for the Linux* sample target and write the result to the current working directory. The second command uses the `report` action to generate a `hotspots` report from the most recent result and write it to `stdout`.

```
vtune -collect hotspots -- /home/test/sample
```

```
vtune -R hotspots
```

Generate a hotspots report from a hotspots analysis and group data by module.

```
vtune -R hotspots -result-dir r001hs -group-by module
```

Open source view with the hotspots performance metrics for the `foo` function and use the Windows* `C:\test\my_sources` directory to search for source files.

```
vtune -R hotspots -source-object function=foo -r r001hs -source-search-dir C:\test\my_sources
```

Write stack information for all functions in the threading analysis result r003tr. The data is grouped by call stack.

```
vtune -R callstacks -r r003tr -group-by callstack
```

See Also

[report-output](#)

option

[Save and Format Command Line Reports](#)

[Filter and Group Command Line Reports](#)

[Generate Command Line Reports](#)

report-knob

Set configuration options for the specified report type.

Syntax

```
-report-knob<knobName>=<knobValue>
```

Arguments

<knobName>	<knobValue>	Supported Report	Description
show-issues	true false. Default: true	summary	<p>Skip issue descriptions in the generated report.</p> <hr/> <p>NOTE This knob is available only for the HPC Performance Characterization analysis report.</p>

Actions Modified

[report](#)

Description

Use the `-report-knob` action-option to configure knob settings for a `report` action.

Example

This example generates the summary report for the HPC Performance Characterization analysis result and skips issue descriptions.

```
vtune -report summary -r r001hpc -report-knob show-issues=false
vtune: Executing actions 75 % Generating a report
Elapsed Time: 23.182s
GFLOPS: 14.748
CPU Utilization: 58.0%
    Average CPU Usage: 13.920 Out of 24 logical CPUs
    Serial Time: 0.069s (0.3%)
    Parallel Region Time: 23.113s (99.7%)
    Estimated Ideal Time: 14.010s (60.4%)
```

```

    OpenMP Potential Gain: 9.103s (39.3%)
Memory Bound: 0.446
    Cache Bound: 0.175
    DRAM Bound: 0.216
    NUMA: % of Remote Accesses: 38.3%
FPU Utilization: 2.7%
    GFLOPS: 14.748
        Scalar GFLOPS: 4.801
        Packed GFLOPS: 9.947
Collection and Platform Info
    Application Command Line: ./sp.B.x
    User Name: vtune
    Operating System: 3.10.0-327.el7.x86_64 NAME="Red Hat Enterprise Linux Server" VERSION="7.2
(Maipo)" ID="rhel" ID_LIKE="fedora" VERSION_ID="7.2" P
    RETTY_NAME="Red Hat Enterprise Linux Server 7.2 (Maipo)" ANSI_COLOR="0;31" CPE_NAME="cpe:/o:redhat:enterprise_linux:7.2:GA:server" HOME_URL="https://www.redhat.com/" BUG_REPORT_URL="https://bugzilla.redhat.com/" REDHAT_BUGZILLA_PRODUCT="Red Hat Enterprise Linux 7" REDHAT_BUGZILLA_PRODUCT_VERSION=7.
    2 REDHAT_SUPPORT_PRODUCT="Red Hat Enterprise Linux" REDHAT_SUPPORT_PRODUCT_VERSION="7.2"
    Computer Name: test
    Result Size: 1 GB
    Collection start time: 19:04:30 13/07/2016 UTC
    Collection stop time: 19:04:53 13/07/2016 UTC
    Name: Intel(R) Xeon(R) E5/E7 v2 Processor code named Ivytown
    Frequency: 2.694 GHz
    Logical CPU Count: 24
    CPU
        Name: Intel(R) Xeon(R) E5/E7 v2 Processor code named Ivytown
        Frequency: 2.694 GHz
        Logical CPU Count: 24

```

This example generates the summary report for the HPC Performance Characterization analysis result and shows issue descriptions.

```

vtune -report summary -r r001hpc -report-knob show-issues=true
vtune: Executing actions 75 % Generating a report
Elapsed Time: 23.182s
GFLOPS: 14.748
CPU Utilization: 58.0%
| The metric value is low, which may signal a poor logical CPU cores
| utilization caused by load imbalance, threading runtime overhead, contended
| synchronization, or thread/process underutilization. Explore CPU Utilization
| sub-metrics to estimate the efficiency of MPI and OpenMP parallelism or run
| the Threading analysis to identify parallel bottlenecks for other
| parallel runtimes.
|
Average CPU Usage: 13.920 Out of 24 logical CPUs
Serial Time: 0.069s (0.3%)
Parallel Region Time: 23.113s (99.7%)
    Estimated Ideal Time: 14.010s (60.4%)
OpenMP Potential Gain: 9.103s (39.3%)
    | The time wasted on load imbalance or parallel work arrangement is
    | significant and negatively impacts the application performance and
    | scalability. Explore OpenMP regions with the highest metric values.
    | Make sure the workload of the regions is enough and the loop schedule
    | is optimal.
|
...

```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

report-output

Write a generated report to a file.

Syntax

`-report-output <pathname>`

Arguments

Argument	Description
<code><dir></code>	Name of the directory if you are writing multiple report files
<code><pathname></code>	Directory, filename and extension of a single report file.

Default

The report is written to stdout.

Actions Modified

`report`

Description

Use the `report-output` action-option to write a report to a file.

- If the filename includes a file extension, it is used unchanged.
- If the file extension is not included in the filename, the value specified for the `format` option is used (.txt for text or .csv for csv).

NOTE

If you specify a .csv file, use the `csv-delimiter` option to specify which delimiter you want to use in the report.

Example

This example generates a wait-time report for the `r001tr` Threading analysis result and saves it in the /home/text/report.txt file.

```
vtune -report wait-time -r r001tr -format text -report-output /home/test/report.txt
```

This example creates a hotspots report from the most recent hotspot result and saves it as a .csv file with tab delimiters.

```
vtune -R hotspots -report-output MyReport.csv -format csv -csv-delimiter tab
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

report-width

Set the maximum width for a report

Syntax

```
-report-width <double>
```

Arguments

<double>	The maximum number of characters per line in a report.
----------	--

Default

None

Actions Modified

report

Description

If a report is too wide to view or print properly, use the `report-width` option to limit the number of characters per line.

Example

Output a hotspots report from the most recent result as a text file with a maximum width of 60 characters per line.

```
vtune -report hotspots -report-width 60 -report-output MyHotspotsReport.txt
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

result-dir

Specify the result directory.

GUI Equivalent

Configure Analysis window > **WHAT** pane

Syntax

```
-result-dir <PATH>  
-r <PATH>
```

Arguments

Argument

Description

<PATH>	The PATH/name of a directory where a result is stored. This may be an absolute pathname, or a pathname relative to the current working directory. If the final component of the pathname does not exist, it is created.
--------	---

Default

If not specified, the default result name and directory are used. If not specified for a `collect` or `collect-with` action, a new result directory is created in the current working directory. If not specified when generating a report, the report uses the highest numbered compatible result in the current working directory. The default name for a result directory is `r@@@{at}`, where `@@@` is the incremented number of the result, and `{at}` is a two- or three-letter abbreviation for the analysis type.

Actions Modified

`collect, collect-with, finalize, import, report`

Description

Use the `result-dir` option to specify the result directory. If you specify the result directory for collection or to import results from other projects, you should also specify the result directory for any actions that use this result, such as `report..`. Specifying the result directory when using the `finalize` action is highly recommended.

If you want to specify the result directory name, you can use the auto-incremented counter pattern `@@@` with a prefix and/or suffix.

For example, you could use the prefix `myResult-` and the usual analysis type suffix like this: `myResult-@@@{at}`. If you then perform a memory error analysis, followed by a threading error analysis, specifying `-result-dir myResult-@@@{at}` each time, the result directories would be assigned the following names: `myResult-000mil` and `myResult-001ti2`.

Both short names and long names are case-sensitive. For example, `-R` is the short name of the `report` action, and `-r` is the short name of the `result-dir` action-option.

Alternate Options

The `user-data-dir` global-option can be used to specify the base directory for results. Result directories created under this base directory would use the default naming conventions unless specified using the `result-dir` action-option.

Example

This example starts the Threading analysis of the `myApplication` application and saves the results in the baseline `result` directory.

On Linux*:

```
vtune -collect threading -result-dir /temp/test/baseline -- /temp/test/myApplication
```

On Windows*:

```
vtune -collect threading -result-dir C:\test\baseline -- C:\test\myApplication.exe
```

See Also

[Specify Result Directory from Command Line](#)

[Manage Result Files
from GUI](#)

[vtune Command Syntax](#)

[vtune Actions](#)

resume-after

Resume collection after the specified number of seconds.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Advanced** section > **Automatically resume collection after** option

Syntax

`-resume-after <value>`

Arguments

Argument	Description
<code><value></code>	The number of seconds that should elapse before data collection is resumed. Fractions of seconds are possible, for example: 1.56 for 1 sec 560 msec.

Default

OFF	Data collection started in the paused mode is not resumed unless this option is specified or the pause/resume API call in the target code is reached.
-----	---

Actions Modified

`collect`

Description

Use the `resume-after` option with the `start-paused` option to automatically exit paused mode after the specified number of seconds has elapsed.

Example

This example starts a Linux* sample application in paused mode and resumes the Hotspots analysis in 5 seconds.

```
vtune -collect hotspots -resume-after 5 -- /home/test/sample
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

[Pause Data Collection](#)

in GUI

[return-app-exitcode](#)

Return the exit code of the target.

Syntax

`-return-app-exitcode`

Default

OFF	By default, the vtune exit code is returned.
-----	--

Actions Modified

collect

Description

Use the `return-app-exitcode` option to return the exit code of the target rather than the vtune tool.

Example

This example runs the Threading analysis for the sample Linux* application, generates a default summary report, and returns the exit code of the sample application.

```
vtune -collect threading -return-app-exitcode -- /home/test/sample
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

ring-buffer

Limit the amount of raw data to be collected by setting the timer that enables the analysis only for the last seconds before the target or collection is terminated.

GUI Equivalent

Configure Analysis window >**WHAT** pane > **Advanced** > **Limit collected data by: Time from collection end, sec** option

Syntax

`-ring-buffer=<integer>`

Arguments

<code><integer></code>	Timer (in sec)
------------------------------	----------------

Actions Modified

collect, collect-with

Description

Use the `ring-buffer` action-option to limit the amount of raw data to be collected. The option sets the timer (in sec) that enables the analysis only for the last seconds before the target or collection is terminated.

Alternate Options

<code>data-limit</code>	Limit the amount of raw data (in MB) to be collected.
-------------------------	---

Example

Enable a Hotspots analysis for the last 10 seconds before the collection is terminated.

```
vtune -collect hotspots -ring-buffer=10 myApp
```

See Also

Limit Data Collection

data-limit

action-option

vtune Command Syntax

vtune Actions

search-dir

Specify a search directory for binary and symbol files.

GUI Equivalent

Binary/Symbol Search dialog box

Syntax

-search-dir *DIR*

Arguments

DIR Specify the name of the search directory to add.

Default

Only default search directories are used.

Actions Modified

collect, finalize, import

Description

This option specifies search directories for binary and symbol files. It is often used in conjunction with the `finalize` action to re-finalize a result when a symbol file is missed during collection. It is also used if you import results from another system.

During data collection, the result directory is set as the default search directory for the collected result.

If you import results from another system, specify additional search directories for system modules. To show correct results, the `vtune` tool requires the same modules that were used for data collection. To ensure the Intel® VTune™ Profiler takes the right module, copy the original system modules to your system.

Alternate Options

source-search-dir

Specify a search directory for source files.

Examples

When your source files are in multiple directories, use the `search-dir` option multiple times so that all the necessary directories are searched.

```
vtune -collect hotspots -knob sampling-mode=hw -search-dir /home/my_system_modules -search-dir /home/other_system_modules -- /home/test/myApplication
```

This example finalizes the `r001hs` result searching for symbol files in the `C:\Import\system_modules` directory.

```
vtune -finalize -search-dir C:\Import\system_modules -r C:\Import\r001hs
```

See Also

[source-search-dir](#)

action-option

[vtune Command Syntax](#)

[vtune Actions](#)

[Search Directories](#)

[Finalization](#)

show-as

Specify report values as events, samples, or percentage.

GUI Equivalent

Context Menu: Grid

Syntax

```
-show-as samples | events | percent
```

Arguments

Argument	Description
<code>samples</code>	Show the total number of samples collected for each event in the viewpoints provided for the hardware event-based sampling data collection.
<code>events</code>	Show the number of times the event occurred during sampling data collection. VTune Profiler determines this value by applying the following formula for each event: < Event name > samples * Sample After value .
<code>percent</code>	Show the percentage of samples collected for the event. This value is calculated using the following formula: (Number of samples collected for the event/ Total number of samples collected for the event) x 100 .

Default

<code>samples</code>	Performance data collected during the hardware event-based analysis are displayed as samples.
----------------------	---

Actions Modified

`report`

Description

Choose the data format for displaying results collected during hardware event-based sampling.

Example

Generate a hardware events report for the result collected during a hotspots analysis and show as a percentage of events.

```
vtune -report hw-events -result-dir r001hs -show-as percent
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

[Choose Data Format](#)

from GUI

[Hardware Event-based Sampling Collection](#)

sort-asc

Sort data in ascending order by the specified column name.

GUI Equivalent

[Context Menu: Grid](#)

Syntax

`-sort-asc <string>`

`-s <string>`

Arguments

Argument

`<string>`

Description

Column name that corresponds to a performance metric or event name. Multiple values are possible.

Actions Modified

`report`

Description

Use the `sort-asc` option with the `report` action to sort data by the specified column name in ascending order. Each column name corresponds to a performance metric or event.

You can specify multiple values as a comma-separated string (no spaces).

Alternate Options

Opposite: `sort-desc`

Example

This example sorts the data collected in the r001ue result and displayed in the Hardware Events report in the ascending order by the INST_RETIREDA.NY and CPU_CLK_UNHALTED.CORE event columns.

```
vtune -r r001ge -report hw-events -sort-asc=INST_RETIREDA.NY,CPU_CLK_UNHALTED.CORE
```

See Also

[Generate Command Line Reports](#)

[Reference](#)

sort-desc

Sort data in descending order by the specified column name.

GUI Equivalent

Context Menu: Grid

Syntax

```
-sort-desc <string>  
-S <string>
```

Arguments

Argument	Description
<string>	Column name that corresponds to a performance metric or event name. Multiple values are possible.

Default

Varies by report.

Actions Modified

report

Description

Use the sort-desc option with the report action to sort data by the specified column name in descending order. Each column name corresponds to a performance metric or event.

You can specify multiple values as a comma-separated string (no spaces).

Alternate Options

Opposite: [sort-asc](#)

Example

Sort the data collected in the r001ue result and displayed in the Hardware Events report in the descending order by the INST_RETIREDA.NY and CPU_CLK_UNHALTED.CORE event columns.

```
vtune -r r001ue -report hw-events -sort-desc=INST_RETIREDA.NY,CPU_CLK_UNHALTED.CORE
```

See Also

[Generate Command Line Reports](#)

Reference

source-object

Type of source object to display in a report for source or assembly data.

GUI Equivalent

Context Menus: Source/Assembly Window

Syntax

```
-source-object <object_type> [=] <value>
```

Arguments

Argument	Description
<object_type>	Application unit for which source or assembly data should be displayed. Possible values are: module, source-file, function.

Actions Modified

`report` with either `hw-events` or `hotspots` report type.

Description

Use the `source-object` option to switch report to source or assembly view mode, including associated performance data. To define a particular object, you can specify this option more than once. For example, if two modules each have a function named `foo`, VTune Profiler will throw an error unless you specify both the module and function.

Tip

By default, source view is displayed. Specify `group-by address` to see disassembly view with associated performance data.

Examples

Generate a hardware events report that displays source data for the `foo` function. Since the result directory is not specified, the most recent hardware analysis result in the current working directory is used.

```
vtune -report hw-events -source-object function=foo
```

This example specifies the object as the function `foo` in `module1`. This would avoid a conflict if there was a second function named `foo` in some other module.

```
vtune -report hw-events -source-object module=module1 -source-object function=foo
```

Generate a hardware events report that displays assembly data for the `foo` function.

```
vtune -R hw-events -source-object function=foo -group-by address
```

Generate a hardware events report that displays assembly data grouped by basic block and then address.

```
vtune -R hw-events -source-object function=foo -group-by basic-block,address
```

Generate a hardware events report that displays assembly data grouped by function-range, then basic block, and then by address.

```
vtune -R hw-events -source-object function=foo -group-by function-range,basic-block,address
```

See Also

[filter](#)

[vtune Command Syntax](#)

[vtune Actions](#)

source-search-dir

Specify a search directory for source files.

GUI Equivalent

Dialog Box: Source Search

Syntax

```
-source-search-dir DIR
```

Arguments

Argument	Description
<i>DIR</i>	Specify the name of the search directory to add.

Default

Only default search directories are used.

Actions Modified

[report](#)

Description

This option specifies search directories for source files. Use this option to specify the location of source files required to provide correct source view report with the `source-object` option.

During data collection, the result directory is set as the default search directory for the collected result.

Alternate Options

<code>search-dir</code>	Specify search directories for symbol and binary files.
-------------------------	---

Example

This command opens the source view with the hotspots performance metrics for the foo function and uses the directory to search for source files.

```
vtune -report hotspots -source-object function=foo -r r001hs -source-search-dir /home/my_sources
```

See Also

[search-dir](#)

action-option

[source-object](#)

action-option

vtune Command Syntax

vtune Actions

Search Directories

stack-size

Specify the size of a raw stack (in bytes) to process.

GUI Equivalent

Custom Analysis

Syntax

`-stack-size=<value in bytes>`

Arguments

Possible `<value>`: numbers between 0 and 2147483647

Default

0

The stack size is unlimited.

NOTE

For [driverless sampling collection](#), the default value is 1024 bytes.

Actions Modified

`collect-with`

Description

When you configure a custom[hardware event-based sampling collection](#), you may reduce the collection overhead and limit the stack size (in bytes) processed by the VTune Profiler by using the `-stack-size` option.

Example

This example configures and runs a custom event-based sampling data collection with the stack size limited to 8192 bytes:

```
vtune -collect-with runsa -knob enable-stack-collection=true -knob stack-size=8192 -knob enable-call-counts=true -knob event-config=CPU_CLK_UNHALTED.REF_TSC:sa=1800000,CPU_CLK_UNHALTED.THREAD:sa=1800000,INST_RETIRE...
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

start-paused

Start data collection in the paused mode.

GUI Equivalent

Toolbar: Command

Syntax

-start-paused

Default

OFF Data collection starts without pausing.

Actions Modified

`collect` with one of the user-mode sampling analysis types

Description

This option starts the data collection in the paused mode.

Collection resumes when pause/resume API calls in the target code are reached, when the `command` action is used with the `resume` argument, or if the `resume-after` option is used, when the specified time has elapsed.

Example

This example starts the hotspots analysis of the `sample` application in the paused mode.

```
vtune -collect hotspots -start-paused -- /home/test/sample
```

See Also

`resume-after`
option

[vtune Command Syntax](#)

[vtune Actions](#)

[Pause Data Collection](#)
in GUI

strategy

Specify which processes to analyze.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Analyze child processes** check box > **Per-process Configuration**

Syntax

`-strategy <process_name1>:<profiling_mode>,<process_name2>:<profiling_mode>,...`

Arguments

Argument	Description
----------	-------------

<code><process_name></code>	The name of the process to which the strategy configuration applies. If <code><process_name></code> is empty, the strategy configuration applies by default to all processes for which a profiling strategy is not specified.
<code><profiling_mode></code>	The strategy for profiling the specified process. Possible values are:
Value	Description
<code>trace:trace</code>	Collect data on the process, and its child processes.
<code>notrace:trace</code>	Do not analyze the process, but collect data on its child processes.
<code>notrace:notrace</code>	Ignore the process, and its child processes, while collecting data.
<code>trace:notrace</code>	Analyze the process, but do not collect data on its child processes.

Default

`:trace:trace` Analyze both parent and child processes.

Actions Modified

`collect, collect-with`

Description

Use the `strategy` action-option to specify which processes to analyze, and which to ignore.

This option is not applicable to hardware event-based analysis types.

Example

This example performs a Hotspots analysis where the strategy configuration limits data collection to the `example` process, and ignores its child processes.

```
vtune -collect hotspots -strategy :notrace:trace,example:trace:notrace /home/test/run_example.sh
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

[Set Up Analysis Target
from GUI](#)

target-install-dir

Specify a path to the VTune Profiler target package installed on the remote system.

GUI Equivalent

Configure Analysis window > **WHERE** pane > **Remote Linux (SSH)** target system > **VTune Profiler installation directory on the remote system** option

Syntax

`-target-install-dir=<string>`

Arguments

`<string>` Path to the product installed on a remote Linux system. If the product is installed to the default location, this option is configured automatically.

Default

`/opt/intel/vtune_profiler_<version>`

Actions Modified

`collect, collect-with`

Description

VTune Profiler supports command line analysis of applications running on a remote Linux or Android system (*target*) using the following product components installed:

- Host: VTune Profiler command line interface (vtune)
- Remote embedded Linux or Android target: target package with data collectors

To enable remote analysis, make sure the path to the VTune Profiler installed on the remote target system is specified. If you use the default installation directory, the VTune Profiler on the host system automatically detects the location of the remote components. Otherwise, use the `-target-install-dir` to specify the correct path.

Example

This command runs Hotspots analysis with stacks for a Linux application and specifies a path to the remote version of the VTune Profiler installed to a non-default location:

```
host>./vtune --target-system=ssh:user1@172.16.254.1 -target-install-dir=/home/vtune_2020.123456 -  
collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -- /home/samples/  
matrix
```

See Also

[Set Up Remote Linux* Target](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

[vtune Command Syntax](#)

[vtune Actions](#)

target-system

Collect data on a remote machine using SSH/ADB connection.

GUI Equivalent

Configure Analysis window > **WHERE** pane

Syntax

`-target-system=<string>`

Arguments

`<string>`

Target system for remote collection. Supported values are:

- `ssh:username@hostname[:port]` - for Linux* systems, where you specify a user name, network name of the remote system accessed via SSH (usually IP address), and a port to connect (if required).
- `get-perf-cmd:pmuName` - for Linux* systems. When you specify the target PMU name, this argument displays on the command line the parameters for the `perf` driverless collector for a specific analysis. To see a list of available PMUs, type:

```
sep -platform-list
```

Use this argument when:

- You do not have an SSH connection to the target machine.
- You cannot install VTune Profiler on the target machine, for security reasons.

NOTE

The Linux Perf* tool (driverless collection) supports complex event names that contain `.:=` symbols in v4.18 and newer versions. For example,

```
perf record -e cpu/
period=0x98968f,event=0xc7,umask=0x20,name=
\'FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE\'/uk ./a.out
```

Complex names like this example are not necessary for the Perf tool itself. You can replace these symbols for a simpler name.

```
perf record -e cpu/
period=0x98968f,event=0xc7,umask=0x20,name=
\'FP_ARITH_INST_RETIRED_256B_PACKED_SINGLE\'/uk ./a.out
```

Actions Modified

`collect`, `collect-with`

Description

Intel® VTune Profiler enables you to analyze applications running on a remote Linux system or Android device (*target system*) using the VTune Profiler command line interface (`vtune`) installed on the host system (remote usage mode). Use the `target-system` option to specify your target system and enable remote data collection.

For details, see [Linux* System Setup for Remote Analysis](#) and [Android* System Setup](#).

Example

This command runs Hotspots analysis in the hardware event-based mode for the application on a Linux embedded system:

```
host>./vtune --target-system=ssh:user1@172.16.254.1 -collect hotspots -knob collection-type:hw-
events -- /target-system-path/app
```

This example shows a list of available PMU names and the command for driverless collection for a Linux system:

```
$sep -platform-list
...
Platform: 111, PMU: skylake_server, Signature: 0x50650, CPU name: Intel(R) Xeon(R) Processor
code named Skylake
...
$ vtune --collect uarch-exploration --target-system=get-perf-cmd:skylake_server
```

This command runs Hotspots analysis in the user-mode sampling mode for the application on an Android system:

```
host>./vtune --collect hotspots --target-system=android -r quadrant_r@@@ --target-process
com.intel.fluid
```

This command runs Hotspots analysis in the hardware event-based mode for the application on an Android system:

```
host>./vtune --collect hotspots -knob collection-type:hw-events --target-system=android -r
quadrant_r000 --target-process com.intel.fluid
```

See Also

[Set Up Remote Linux* Target](#)

[Android* Targets](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

[Android* Target Analysis from the Command Line](#)

[vtune Command Syntax](#)

[vtune Actions](#)

target-tmp-dir

Specify a path to the temporary directory on the remote system where performance results are temporarily stored.

GUI Equivalent

Configure Analysis window > **WHERE** pane > **Remote Linux (SSH)** target system > **Temporary directory on the remote system** option

Syntax

`-target-tmp-dir=<string>`

Arguments

`<string>` Path to a directory on the remote Linux system where performance results are temporarily stored.

Default

`/tmp`

Actions Modified

`collect, collect-with`

Description

VTune Profiler supports command line analysis of applications running on a remote Linux system (*target*) using the following product components installed:

- Host: VTune Profiler command line interface (`vtune`)
- Remote embedded Linux or Android target: target package with data collectors

When the VTune Profiler collects data remotely, performance data is temporarily saved to the default `/tmp` directory on the remote system. If, for some reason, you changed the default temporary directory, make sure to specify this path with the `-target-tmp-dir` option.

This command runs Hotspots analysis with stacks for a Linux application and specifies a non-default temporary location on the remote system:

```
host>./vtune --target-system=ssh:vtune@10.125.21.170 -target-tmp-dir=/home/tmp -collect hotspots
-knob sampling-mode=hw -knob enable-stack-collection=true -- /home/samples/matrix
```

See Also

[Temporary Directory for Performance Results on Linux* Targets](#)

[Set Up Remote Linux* Target](#)

[Collect Data on Remote Linux* Systems from Command Line](#)

[vtune Command Syntax](#)

[vtune Actions](#)

target-duration-type

Adjust the sampling interval for longer-running targets.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Duration time estimate** option

Syntax

`-target-duration-type veryshort | short | medium | long`

Arguments

<code>veryshort</code>	Target takes less than 1 minute to run.
<code>short</code>	Target takes between 1 and 15 minutes to run.
<code>medium</code>	Target takes 15 minutes to 3 hours to run.
<code>long</code>	Target takes more than 3 hours to run.

Default

<code>short</code>	This is appropriate for a target that runs for 1 to 15 minutes.
--------------------	---

Actions Modified

`collect, collect-with`

Description

If your target runs 15 minutes or longer, or if it runs less than one minute, use the `target-duration` action-option to set a different duration type. The `collect` or `collect-with` action uses this value to adjust the sampling interval, which determines how much data is collected. For longer-running targets, the sampling interval is greater (less frequent) to reduce the amount of collected data. For very short-running targets, the sampling interval is smaller (more frequent). For hardware event-based analysis types, a multiplier applies to the configured Sample After value.

NOTE

This option is deprecated. Use the `-knob sampling-interval` option instead.

Example

Perform a Hotspots analysis using a medium sampling interval that is appropriate for targets with a duration of 15 minutes to 3 hours.

```
vtune -collect hotspots -knob sampling-mode=hw -target-duration-type medium -- MyApp
```

See Also

[Manage Analysis Duration from Command Line](#)

[Sample After Value](#)

[Sampling Interval](#)

[vtune Actions](#)

[vtune Command Syntax](#)

target-pid

Attach a collection to a running process specified by the process ID.

GUI Equivalent

Configure Analysis window> **WHAT** pane > **Attach To Process** > **PID**

Syntax

`-target-pid <value>`

Arguments

ID of process that you want to analyze.

Actions Modified

`collect, collect-with`

Description

Use the `target-pid` option to attach a `collect` or `collect-with` action to a running process specified by its process ID (pid).

Alternate Options

The `target-process` option provides the same capabilities, but uses the process name to specify the process.

Example

Attach a hotspots collection to a running process whose ID is 1234.

```
vtune -collect hotspots -target-pid 1234
```

See Also

[vtune Actions](#)

[vtune Command Syntax](#)

target-process

Attach a collection to a running process specified by the process name.

GUI Equivalent

Configure Analysis window > **WHAT** pane > **Attach To Process** > **Process name**

Syntax

```
-target-process <string>
```

Arguments

A string containing the name of the process to profile.

Actions Modified

`collect`, `collect-with`

Description

Use the `target-process` option to attach a `collect` or `collect-with` action to a running process specified by the process name.

Alternate Options

The `target-pid` option provides the same capabilities, but uses the process ID to specify the process.

Example

In this example, a Hotspots analysis is attached to the `myApp` process, which is already running on the system.

```
vtune -collect hotspots -target-process myApp
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

time-filter

Filter reports by a time range.

IDE Equivalent

Pane: Timeline

Syntax

`time-filter=<value>`

Arguments

`<value>` Specify filtered time range (in seconds) using format
`<begin_time>:<end_time>.`

Default

`OFF` By default, `vtune-cl` reports display data for the full analysis duration.

Actions Modified

`report`

Description

Use the `time-filter` option to filter the report and display data for the specified time range only. For example, `-time-filter=2.3:5.4` reports data collected from 2.3 seconds to 5.4 seconds of [Elapsed Time](#).

Examples

```
vtune-cl -R hotspots -time-filter=2.3:5.4
```

See Also

[Run Command Line Analysis](#)

[vtune Command Syntax](#)

[Filter and Group Command Line Reports](#)

trace-mpi

*For message passing interface (MPI) analysis ,
configure collectors to determine MPI rank ID in case
of a non-Intel MPI library implementation.*

Syntax

`-trace-mpi | -no-trace-mpi`

Default

`-no-trace-mpi`

Actions Modified

`collect, collect-with`

Description

Based on the `PMI_RANK` or `PMI_ID` MPI analysis environment variable (whichever is set), the VTune Profiler extends a process name with the captured rank number that is helpful to differentiate ranks in a VTune Profiler result with multiple ranks. The process naming schema in this case is `<process_name>` (rank `<N>`). Use the `-trace-mpi` option to enable detecting an MPI rank ID for MPI implementations that do not provide the environment variable.

Examples

This command runs the Hotspots analysis type (hardware event-based sampling mode) with enabled MPI rank ID detection.

```
mpirun -n 4 vtune -result-dir my_result -trace-mpi -collect hotspots -knob sampling-mode=hw -- ./test.x
```

See Also

[MPI Code Analysis](#)

user-data-dir

Specify the base directory for result paths.

GUI Equivalent

Configure Analysis window > **WHAT** pane

Syntax

```
-user-data-dir <PATH>
```

Arguments

A string containing the `PATH/name` of the user data directory.

Default

The current working directory.

Actions Modified

`collect, finalize, import`

Description

Use the `user-data-dir` action-option with the `result-dir` action-option when you want to specify a base directory for results.

Example

This example runs a Threading analysis of the `sample` Linux application and creates the default-named result directories under the `myresults` directory.

```
vtune -collect threading -user-data-dir /root/intel/myresults -- /home/test/sample
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

[result-dir](#)

option

[Manage Result Files from GUI](#)

verbose

Display detailed information on actions performed by the vtune tool.

Syntax

-verbose

-v

Default

OFF Standard amount of information is displayed.

Description

Use the `verbose` option when you want to see detailed information on the actions performed by the `vtune` command.

Example

This example displays detailed information while running a Hotspots analysis.

```
vtune -collect hotspots -verbose -- /home/test/sample
```

See Also

[quiet](#)

option

[vtune Command Syntax](#)

[vtune Actions](#)

version

Display version information for the vtune tool.

Syntax

-version

-V

Description

This action displays version information for the Intel® VTune™ Profiler and the `vtune` command.

Example

This example shows version information for the Intel® VTune™ Profiler and the `vtune` command.

```
vtune -version
```

See Also

[vtune Command Syntax](#)

[vtune Actions](#)

Introduction

Report Problems from Command Line

If the product crashes, you can use the `amplxe-feedback` command tool to package relevant information and send a report to the Intel Customer Support Team.

Basic Crash Report Process

1. Create a bug report package using the `create-bug-report` action and desired options.
2. Use the `send-crash-report` action to send the report to the Intel Customer Support Team.

Crash Report Actions

Action	Argument	Description
<code>create-bug-report</code>	<PATH> Pathname for the bug report.	Package the following into a bug report package: product log files, system information, crash reports for each running product process, and product installation details.
<code>list-crash-report</code>	None	Output a list of existing bug reports.
<code>report-system-info</code>	None	Output system information.
<code>send-crash-report</code>	<PATH> Pathname for the bug report.	Email the specified bug/crash report(s) to the Intel Customer Support Team.

Options for `create-bug-report`

Option	Argument	Description
<code>dump-stack</code>	<PID> Process identifier	Create crash report for process with specified process identifier (PID) when using <code>create-bug-report</code> .
<code>no-dump</code>	None	Disable crash reports to conserve system resources.
<code>dump-memory</code>	None	Use with <code>dump-stack</code> to include crash report for process with specified process identifier (PID).
<code>no-system-info</code>	None	Use when creating a report to speed up the bug report creation process by disabling system information collection.

Examples

This command generates a bug report package and stores it in a compressed file under the name you specify, such as `001bug`.

```
amplxe-feedback -create-bug-report=001bug
```

This command creates a list of crash report filenames.

```
amplxe-feedback -list-crash-report
```

This command outputs system information so you can provide this information to support.

```
amplxe-feedback -report-system-info
```

This command forwards the specified bug report to the Intel Customer Support Team.

```
amplxe-feedback -send-bug-report=r0001b
```

API Support

Intel® VTune™ Profiler supports two kinds of APIs:

- The Instrumentation and Tracing Technology API (ITT API) provided by the Intel®VTune™ Profiler enables your application to generate and control the collection of trace data during its execution.
- The JIT (Just-In-Time) Profiling API provides functionality to report information about just-in-time generated code that can be used by performance tools. You need to insert JIT Profiling API calls in the code generator to report information before JIT-compiled code goes to execution. This information is collected at runtime and used by tools like Intel® VTune™ Profiler to display performance metrics associated with JIT-compiled code.

Instrumentation and Tracing Technology APIs

NOTE

The Instrumentation and Tracing Technology API (ITT API) and the Just-in-Time Profiling API (JIT API) are open source components. Visit the [GitHub* repository](#) to access source code and contribute.

The Instrumentation and Tracing Technology API (ITT API) provided by the Intel®VTune™ Profiler enables your application to generate and control the collection of trace data during its execution.

ITT API has the following features:

- Controls application performance overhead based on the amount of traces that you collect.
- Enables trace collection without recompiling your application.
- Supports applications in C/C++ and Fortran environments on Windows*, Linux*, FreeBSD*, or Android* systems.
- Supports instrumentation for tracing application code.

To use the APIs, add API calls in your code to designate logical tasks. These markers will help you visualize the relationship between tasks in your code relative to other CPU and GPU tasks. To see user tasks in your performance analysis results, enable the **Analyze user tasks** checkbox in analysis settings.

NOTE

The ITT API is a set of pure C/C++ functions. There are no Java* or .NET* APIs. If you need runtime environment support, you can use a JNI, or C/C++ function call from the managed code. If the collector causes significant overhead or data storage, you can pause the analysis to reduce the overhead.

See Also

[Task Analysis](#)

[View Instrumentation and Tracing Technology \(ITT\) API Task Data in Intel® VTune™ Profiler](#)

Basic Usage and Configuration

You can control performance data collection for your application by adding basic instrumentation to your application and by configuring your environment and your build system to use the instrumentation and tracing technology (ITT) APIs.

User applications/modules linked to the static ITT API library do not have a runtime dependency on a dynamic library. Therefore, they can be executed without Intel®VTune™ Profiler.

To use the ITT APIs, set up your C/C++ or Fortran application using the steps provided in [Configuring Your Build System](#).

Unicode Support

All API functions that take parameters of type `_itt_char` follow the Windows OS unicode convention. If `UNICODE` is defined when compiling on a Windows OS, `_itt_char` is `wchar_t`, otherwise it is `char`. The actual function names are suffixed with `A` for the ASCII APIs and `W` for the unicode APIs. Both types of functions are defined in the DLL that implements the API.

Strings that are all ASCII characters are internally equivalent for both the unicode and the ASCII API versions. For example, the following strings are equivalent:

```
_itt_sync_createA( addr, "OpenMP Scheduler", "CriticalSection", 0);
_itt_sync_createW( addr, L"OpenMP Scheduler", LCriticalSection", 0);
```

See Also

[Minimize ITT API Overhead](#)

[Configure Your Build System](#)

[Task Analysis](#)

Configure Your Build System

NOTE

ITT API usage is supported on Windows*, Linux*, FreeBSD*, and Android* systems. It is not supported for QNX* systems.

Before instrumenting your application, you need to configure your build system to be able to reach the API headers and libraries.

For Windows* and Linux* systems:

- Add `<install_dir>/sdk/include` to your INCLUDE path for C/C++ applications or `<install_dir>/sdk/[lib32 or lib64]` to your INCLUDE path for Fortran applications
- Add `<install_dir>/sdk/lib32` to your 32-bit LIBRARIES path
- Add `<install_dir>/sdk/lib64` to your 64-bit LIBRARIES path

NOTE

On Linux* systems, you have to link the `d1` and `pthread` libraries to enable ITT API functionality. Not linking these libraries will not prevent your application from running, but no ITT API data will be collected.

For FreeBSD* systems:

NOTE

Header and library files are available from the `vtune_profiler_target_x86_64.tgz` FreeBSD target package. See [Set Up FreeBSD* System](#) for more information.

- Add `<target-package>/sdk/include` to your INCLUDE path for C/C++ applications or `<install_dir>/sdk/[lib32 or lib64]` to your INCLUDE path for Fortran applications
- Add `<target-package>/sdk/lib64` to your 64-bit LIBRARIES path

For the Android* system, add the following libraries to your LIBRARIES path depending on your device architecture:

- Add <install_dir>/target/android_v5_x86_64/lib-x86_64 for the Intel® 64 architecture
- Add <install_dir>/target/android_v5/lib-x86 for the IA-32 architecture
- Add <install_dir>/target/android_arm/lib-arm for the ARM* architecture

<install_dir> is the Intel® VTune™ Profiler installation directory. The default installation path for the VTune Profiler varies with the product shipment.

NOTE

The ITT API headers, static libraries, and Fortran modules previously located at <install_dir>/include and <install_dir>/lib32 [64] folders were moved to the <install_dir>/sdk folder starting the VTune Profiler 2021.1-beta08 release. Copies of these files are retained at their old locations for backwards compatibility and these copies should not be used for new projects.

Include the ITT API Header or Module in Your Application

For C/C++ Applications

Add the following #include statements to every source file that you want to instrument:

```
#include <ittnotify.h>
```

The ittnotify.h header contains definitions of ITT API routines and important macros which provide the correct logic of API invocation from a user application.

The ITT API is designed to incur almost zero overhead when tracing is disabled. But if you need fully zero overhead, you can compile out all ITT API calls from your application by defining the INTEL_NO_ITTNODE_API macro in your project at compile time, either on the compiler command line, or in your source file, prior to including the ittnotify.h file.

For Fortran Applications

Add the ITTNODE module to your source files with the following source line:

```
USE ITTNODE
```

Insert ITT Notifications in Your Application

Insert __itt_* (C/C++) or ITT_* (Fortran) notifications in your source code.

C/C++ example:

```
__itt_pause();
```

Fortran example:

```
CALL ITT_PAUSE()
```

For more information, see [Instrumenting Your Application](#).

Link the libittnode.a (Linux*, Android*, FreeBSD*) or libittnode.lib (Windows*) Static Library to Your Application

You need to link the static library, libittnode.a (Linux*, FreeBSD*, Android*) or libittnode.lib (Windows*), to your application. If tracing is enabled, this static library loads the ITT API implementation and forwards ITT API instrumentation data to VTune Profiler. If tracing is disabled, the static library ignores ITT API calls, causing nearly zero instrumentation overhead.

After you instrument your application by adding ITT API calls to your code and link the libittnotify.a (Linux*, FreeBSD*, Android*) or libittnotify.lib (Windows*) static library, your application will check the INTEL_LIBITNOTIFY32 or the INTEL_LIBITNOTIFY64 environment variable depending on your application's architecture. If that variable is set, it will load the libraries defined in the variable.

Make sure to set these environment variables for the ittnotify_collector to enable data collection:

On Windows*:

```
INTEL_LIBITNOTIFY32=<install-dir>\bin32\runtime\ittnotify_collector.dll
```

```
INTEL_LIBITNOTIFY64=<install-dir>\bin64\runtime\ittnotify_collector.dll
```

On Linux*:

```
INTEL_LIBITNOTIFY32=<install-dir>/lib32/runtime/libittnotify_collector.so
```

```
INTEL_LIBITNOTIFY64=<install-dir>/lib64/runtime/libittnotify_collector.so
```

On FreeBSD*:

```
INTEL_LIBITNOTIFY64=<target-package>/lib64/runtime/libittnotify_collector.so
```

See Also

[Basic Usage and Configuration](#)

[Minimizing ITT API Overhead](#)

Attach ITT APIs to a Launched Application

You can use the Intel® VTune™ Profiler to attach to a running application instrumented with ITT API. But before launching the application, make sure to set up the following environment variable for the ittnotify_collector:

On Windows*:

```
INTEL_LIBITNOTIFY32=<install-dir>\bin32\runtime\ittnotify_collector.dll
```

```
INTEL_LIBITNOTIFY64=<install-dir>\bin64\runtime\ittnotify_collector.dll
```

On Linux*:

```
INTEL_LIBITNOTIFY32=<install-dir>/lib32/runtime/libittnotify_collector.so
```

```
INTEL_LIBITNOTIFY64=<install-dir>/lib64/runtime/libittnotify_collector.so
```

On FreeBSD:

NOTE

Header and library files are available from the vtune_profiler_target_x86_64.tgz FreeBSD target package. See [Set Up FreeBSD* System](#) for more information.

```
INTEL_LIBITNOTIFY64=<target-package>/lib64/runtime/libittnotify_collector.so
```

NOTE

The variables should contain the full path to the library without quotes.

Example

On Windows:

```
set INTEL_LIBITNOTIFY32=C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin32\runtime
\ittnotify_collector.dll
set INTEL_LIBITNOTIFY64=C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\runtime
\ittnotify_collector.dll
```

On Linux:

```
export INTEL_LIBITNOTIFY32=/opt/intel/oneapi/vtune/latest/lib32/runtime/
libittnotify_collector.so
export INTEL_LIBITNOTIFY64=/opt/intel/oneapi/vtune/latest/lib64/runtime/
libittnotify_collector.so
```

On FreeBSD:

NOTE You may need to change the path to reflect the placement of the FreeBSD target package on your target system.

```
setenv INTEL_LIBITNOTIFY64 /tmp/vtune_profiler_2021.9.0/lib64/runtime/libittnotify_collector.so
```

After you complete the configuration, you can start the instrumented application in the correct environment and Intel® VTune™ Profiler will collect user API data even if the application was launched before the VTune Profiler.

See Also

[Set Up Analysis Target](#)

Instrument Your Application

To get the most out of the ITT APIs when collecting performance data with Intel® VTune™ Profiler, you need to add API calls in your code to designate logical tasks. This will help you visualize the relationship between tasks in your code, including when they start and end, relative to other CPU and GPU tasks.

At the highest level a task is a logical group of work executing on a specific thread, and may correspond to any grouping of code within your program that you consider important. You can mark up your code by identifying the beginning and end of each logical task with `_itt_task_begin` and `_itt_task_end` calls. For example, to track "smoke rendering" separately from "detailed shadows", you should add API tracking calls to the code modules for these specific features.

To get started, use the following API calls:

- `_itt_domain_create()` creates a domain required in most ITT API calls. You need to define at least one domain.
- `_itt_string_handle_create()` creates string handles for identifying your tasks. String handles are more efficient for identifying traces than strings.
- `_itt_task_begin()` marks the beginning of a task.
- `_itt_task_end()` marks the end of a task.

Example

The following sample shows how four basic ITT API functions are used in a multi threaded application:

- [Domain API](#)
- [String Handle API](#)
- [Task API](#)

- [Thread Naming API](#)

```
#include <windows.h>
#include <ittnotify.h>

// Forward declaration of a thread function.
DWORD WINAPI workerthread(LPVOID);
bool g_done = false;
// Create a domain that is visible globally: we will use it in our example.
__itt_domain* domain = __itt_domain_create("Example.Domain.Global");
// Create string handles which associates with the "main" task.
__itt_string_handle* handle_main = __itt_string_handle_create("main");
__itt_string_handle* handle_createthread = __itt_string_handle_create("CreateThread");
void main(int, char* argv[])
{
// Create a task associated with the "main" routine.
__itt_task_begin(domain, __itt_null, __itt_null, handle_main);
// Now we'll create 4 worker threads
for (int i = 0; i < 4; i++)
{
// We might be curious about the cost of CreateThread. We add tracing to do the measurement.
__itt_task_begin(domain, __itt_null, __itt_null, handle_createthread);
::CreateThread(NULL, 0, workerthread, (LPVOID)i, 0, NULL);
__itt_task_end(domain);
}

// Wait a while,...
::Sleep(5000);
g_done = true;
// Mark the end of the main task
__itt_task_end(domain);
}
// Create string handle for the work task.
__itt_string_handle* handle_work = __itt_string_handle_create("work");
DWORD WINAPI workerthread(LPVOID data)
{
// Set the name of this thread so it shows up in the UI as something meaningful
char threadname[32];
wsprintf(threadname, "Worker Thread %d", data);
__itt_thread_set_name(threadname);
// Each worker thread does some number of "work" tasks
while(!g_done)
{
__itt_task_begin(domain, __itt_null, __itt_null, handle_work);
::Sleep(150);
__itt_task_end(domain);
}
return 0;
}
```

See Also

[Basic Usage and Configuration](#)

[Domain API](#)

[String Handle API](#)

[Task API](#)

Minimize ITT API Overhead

The ITT API overhead and its impact on the overall application performance depends on the amount of instrumentation code added to the application. When instrumenting an application with ITT API, you should balance between application performance and the amount of performance data that you need to collect, in order to minimize API overhead while collecting sufficient performance data.

Follow these guidelines to achieve good balance between overall performance of the instrumented application and instrumentation detail:

- Instrument only those paths within your application that are important for analysis.
- Create ITT domains and string handles outside the critical paths.
- Filter data collection by different aspects of your application that can be analyzed separately. The overhead for a disabled API call (thus filtering out the associated call) is always less than 10 clock ticks, regardless of the API.

Conditional Compilation

For best performance in the release version of your code, use conditional compilation to turn off annotations. Define the macro `INTEL_NO_ITTNOTIFY_API` before you include `ittnotify.h` during compilation to eliminate all `__itt_*` functions from your code.

You can also remove the static library from the linking stage by defining this macro.

Usage Example: Using Domains and String Handles

The ITT APIs include a subset of functions which create domains and string handles. These functions always return identical handles for the same domain names and strings. This requires these functions to perform string comparisons and table lookups, which can incur serious performance penalties. In addition, the performance of these functions is proportional to the log of the number of created domains or string handles. It is best to create domains and string handles at global scope, or during application startup.

The following code section creates two domains in the global scope. You can use these domains to control the level of detail that is written to the trace file.

```
__itt_domain* basic = __itt_domain_create(L"MyFunction.Basic");
__itt_domain* detailed = __itt_domain_create(L"MyFunction.Detailed");
// Create string handles at global scope.
__itt_string_handle* h_my_funcion = __itt_string_handle_create(L"MyFunction");
void MyFunction(int arg)
{
    __itt_task_begin(basic, __itt_null, __itt_null, h_my_function);
    Foo(arg);
    FooEx();
    __itt_task_end(basic);
}
__itt_string_handle* h_foo = __itt_string_handle_create(L"Foo");
void Foo(int arg)
{
    // Skip tracing detailed data if the detailed domain is disabled.
    __itt_task_begin(detailed, __itt_null, __itt_null, h_foo);
    // Do some work here...
    __itt_task_end(detailed);
}
__itt_string_handle* h_foo_ex = __itt_string_handle_create(L"FooEx");
void FooEx()
{
    // Skip tracing detailed data if the detailed domain is disabled.
    __itt_task_begin(detailed, __itt_null, __itt_null, h_foo_ex);
    // Do some work here...
```

```

    __itt_task_end(detailed);
}
// This is my entry point.
int main(int argc, char** argv)
{
    if(argc < 2)
        //Disable detailed domain if we do not need tracing from that in this
        //application run
detailed ->flags = 0;
    MyFunction(atoi(argv[1]));
}

```

See Also

[Basic Usage and Configuration](#)

[Instrument Your Application](#)

[Configure Your Build System](#)

View Instrumentation and Tracing Technology (ITT) API Task Data in Intel® VTune™ Profiler

User task and API data can be visualized in Intel® VTune™ Profiler performance analysis results.

After you have added basic annotations to your application to control performance data collection, you can view these annotations in the Intel VTune Profiler timeline. All supported instrumentation and tracing technology (ITT) API tasks can be visualized in VTune Profiler.

Use the following steps to include ITT API tasks in your performance analysis collection:

1. Click the



(standalone GUI)/

(Visual Studio IDE) **Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

The **Configure Analysis** window opens.

2. Set up the analysis target in the **WHERE** and **WHAT** panes.
3. From **HOW** pane, click the



Browse button and select an analysis type. For more information about each analysis type, see [Performance Analysis Setup](#).

4. Select the **Analyze user tasks, events, and counters** checkbox to view the API tasks, counters, and events that you added to your application code.

NOTE

In some cases, the **Analyze user tasks, events, and counters** checkbox is in the expandable **Details** section. To enable the checkbox, use the **Copy** button at the top of the tab to create an editable version of the analysis type configuration. For more information, see [Custom Analysis](#).

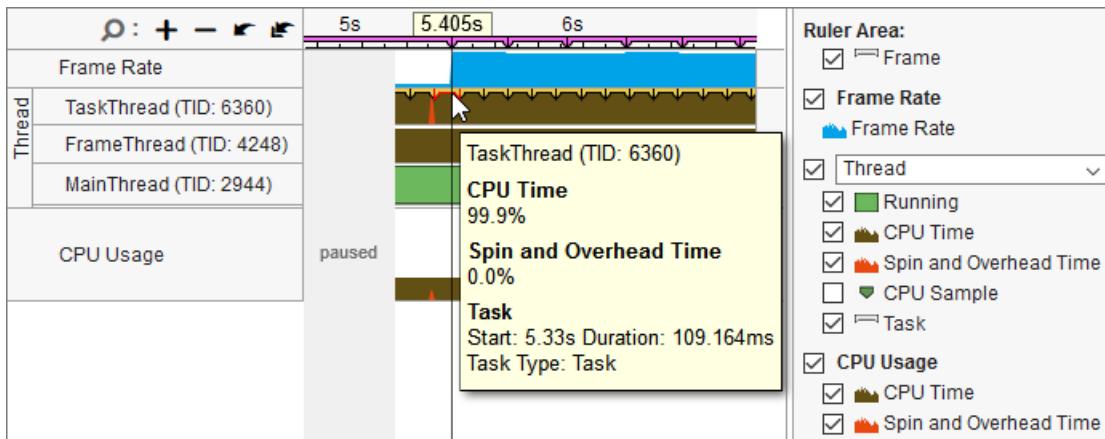
5. Click the



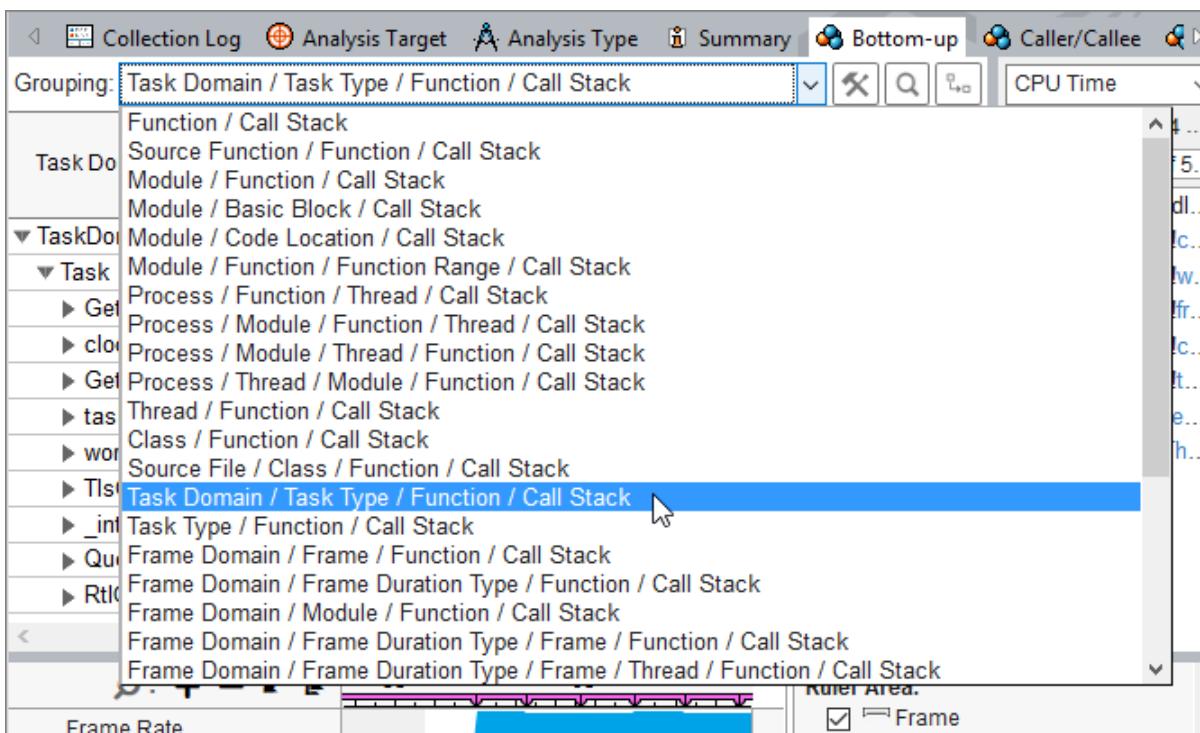
Start button to [run the analysis](#).

After collection completes, the analysis results appear in a viewpoint specific to the analysis type selected. The API data collected is available in the following locations:

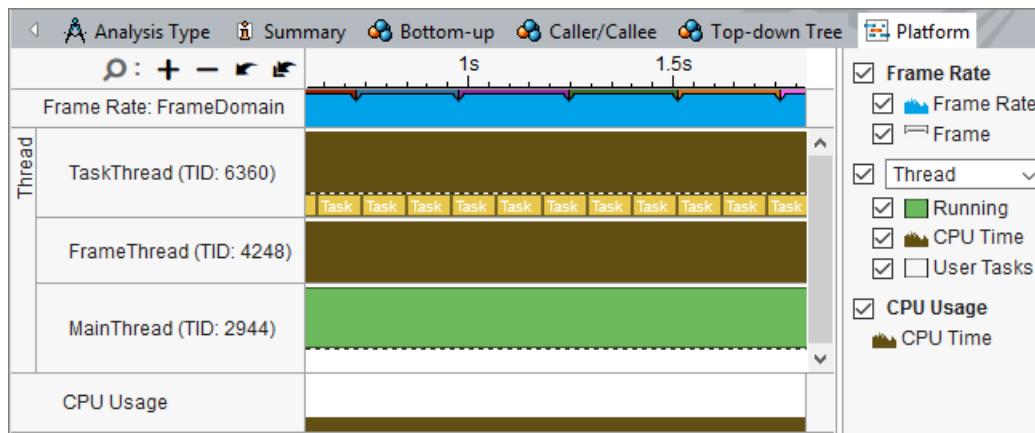
- Timeline view: Each API type appears differently on the timeline view. In the example below, the code was instrumented with the task API, frame API, event API, and collection control API. Tasks appear as yellow bars on the task thread. Frames appear at the top of the timeline in pink. Events appear on the appropriate thread as a triangle at the event time. Collection control events span the entire timeline. Hover over a task, frame, or event to view the type of API task.



- Grid view: Set the **Grouping** to **Task Domain / Task Type / Function / Call Stack** or **Task Type / Function / Call Stack** to view task data in the grid pane.



- Platform tab: Individual tasks are available in a larger view on the **Platform** tab. Hover over a task to get more information.



See Also

[Instrumentation and Tracing Technology APIs](#)

[Basic Usage and Configuration](#)

[Instrument Your Application](#)

[Task Analysis](#)

Instrumentation and Tracing Technology API Reference

These are the available Instrumentation and Tracing Technology API tools:

Domain API

A *domain* enables tagging trace data for different modules or libraries in a program. Domains are specified by unique character strings, for example `TBB.Internal.Control`.

Each domain is represented by an opaque `_itt_domain` structure, which you can use to tag each of the ITT API calls in your code.

You can selectively enable or disable specific domains in your application, in order to filter the subsets of instrumentation that are collected into the output trace capture file. To disable a domain set its flag field to 0 value. This disables tracing for a particular domain while keeping the rest of the code unmodified. The overhead of a disabled domain is a single `if` check.

To create a domain, use the following primitives:

```
_itt_domain *ITTAPIO_itt_domain_create ( const char *name)
```

For a domain name, the URI naming style is recommended, for example, `com.my_company.my_application`. The set of domains is expected to be static over the application's execution time, therefore, there is no mechanism to destroy a domain.

Any domain can be accessed by any thread in the process, regardless of which thread created the domain. This call is thread-safe.

Parameters of the primitives:

[in]	<i>name</i>	Name of domain
------	-------------	----------------

Usage Example

```
#include "ittnotify.h"

__itt_domain* pD = __itt_domain_create(L"My Domain" );pD->flags = 0; /* disable domain */
```

See Also

[Basic Usage and Configuration](#)
[Instrument Your Application](#)
[Minimize ITT API Overhead](#)

String Handle API

Many API calls require names to identify API objects. String handles are pointers to names. They enable efficient handling of named objects in run time and make collected traces data more compact.

To create and return a handle value that can be associated with a string, use the following primitive:

```
__itt_string_handle *ITTAPI__itt_string_handle_create ( const char *name)
```

Consecutive calls to `__itt_string_handle_create` with the same name return the same value. The set of string handles is expected to remain static during the application's execution time, therefore, there is no mechanism to destroy a string handle. Any string handle can be accessed by any thread in the process, regardless of which thread created the string handle. This call is thread-safe.

Parameters of the primitive:

[in]	<i>name</i>	The input string
------	-------------	------------------

See Also

[Basic Usage and Configuration](#)
[Minimize ITT API Overhead](#)

Collection Control API

You can use the collection control APIs in your code to control the way the Intel® VTune™ Profiler collects data for applications.

Use This Primitive	To Do This
<code>void __itt_pause (void)</code>	Run the application without collecting data. VTune Profiler reduces the overhead of collection, by collecting only critical information, such as thread and process creation.
<code>void __itt_resume (void)</code>	Resume data collection. VTune Profiler resumes collecting all data.
<code>void __itt_detach (void)</code>	Detach data collection. VTune Profiler detaches all collectors from all processes. Your application continues to work but no data is collected for the running collection.

Pausing the data collection has the following effects:

- Data collection is paused for the whole program, not only within the current thread.
- Some runtime analysis overhead reduction.
- The following APIs are not affected by pausing the data collection:
 - Domain API
 - String Handle API
 - Thread Naming API
- The following APIs are affected by pausing the data collection. Data is not collected for these APIs while in paused state:

- Task API
- Frame API
- Event API
- User-Defined Synchronization API

NOTE

The Pause/Resume API call frequency is about 1Hz for a reasonable rate. Since this operation pauses and resumes data collection in all processes in the analysis run with the corresponding collection state notification to GUI, you are not recommended to call it on frequent basis for small workloads. For small workloads, consider using the [Frame APIs](#).

Usage Example: Focus on Specific Code Section

The `pause/resume` calls shown in the following code snippet enable you to focus the collection on a specific section of code, and start the application run with collection paused.

```
int main(int argc, char* argv[])
{
    // Do initialization work here
    __itt_resume();
    // Do profiling work here
    __itt_pause();
    // Do finalization work here
    return 0;
}
```

Usage Example: Hide Sections of Code

The `pause/resume` calls shown in the following code snippet enable you to hide some intensive work that you are not currently focusing on:

```
int main(int argc, char* argv[])
{
    // Do work here
    __itt_pause();
    // Do uninteresting work here
    __itt_resume();
    // Do work here
    __itt_detach();
    // Do uninteresting work here
    return 0;
}
```

See Also

[Basic Usage and Configuration](#)

[Frame API](#)

[View Instrumentation and Tracing Technology \(ITT\) API Task Data in Intel® VTune™ Profiler](#)

Thread Naming API

By default, each thread in your application is displayed in the **timeline** track with a default label generated from the process ID and the thread ID, or with the OS thread name. You can use the Thread Naming API in your code to give threads meaningful names.

Thread Naming API is a per-thread function that works in all states (paused or resumed).

To set thread name using a char or Unicode string, use the primitive:

```
void __itt_thread_set_name (const __itt_char *name)
```

Parameters of the primitive:

[in]	name	The thread name
------	------	-----------------

To indicate that this thread should be ignored from analysis:

```
void __itt_thread_ignore (void)
```

It does not affect the concurrency of the application. It does not be visible in the **Timeline** pane.

If the thread name is set multiple times, only the last name is used.

Usage Example

You can use the following thread naming example to give a meaningful name to the thread you wish to focus on and ignore the service thread.

```
DWORD WINAPI service_thread(LPVOID lpArg)
{
    __itt_thread_ignore();
    // Do service work here. This thread will not be displayed.
    return 0;
}

DWORD WINAPI thread_function(LPVOID lpArg)
{
    __itt_thread_set_name("My worker thread");
    // Do thread work here
    return 0;
}

int main(int argc, char* argv[])
{
    ...
    CreateThread(NULL, 0, service_thread, NULL, 0, NULL);
    CreateThread(NULL, 0, thread_function, NULL, 0, NULL);
    ...
    return 0;
}
```

See Also

[Basic Usage and Configuration](#)

Task API

A *task* is a logical unit of work performed by a particular thread. Tasks can nest; thus, tasks typically correspond to functions, scopes, or a case block in a switch statement. You can use the Task API to assign tasks to threads.

Task API is a per-thread function that works in resumed state. This function does not work in paused state.

The Task API does not enable a thread to suspend the current task and switch to a different task (task switching), or move a task to a different thread (task stealing).

A task instance represents a piece of work performed by a particular thread for a period of time. The task is defined by the bracketing of `__itt_task_begin()` and `__itt_task_end()` on the same thread.

NOTE

To be able to see user tasks in your results, enable the **Analyze user tasks** checkbox in analysis settings.

Task API Functions

Create a task instance on a thread. This becomes the current task instance for that thread. A call to `__itt_task_end()` on the same thread ends the current task instance.

```
void __itt_task_begin (const __itt_domain *domain, __itt_id taskid, __itt_id parentid,
__itt_string_handle *name)
```

Trace the end of the current task.

```
void __itt_task_end (const __itt_domain *domain)
```

ITTAPI __itt_task_* Function Parameters

The following table defines the parameters used in the Task API primitives.

Type	Parameter	Description
[in]	<code>__itt_domain</code>	The domain of the task.
[in]	<code>__itt_id taskid</code>	This is a reserved parameter.
[in]	<code>__itt_id parentid</code>	This is a reserved parameter.
[in]	<code>__itt_string_handle</code>	The task string handle.

Enable Task APIs

The following steps are required to begin using task APIs:

1. Include `ittnotify.h` header.
2. Create domain and string handles for your tasks.
3. Insert task begin and task end marks in your code.
4. Link to `libittnotify.lib` (Windows*) or `libittnotify.a` (Linux*).
5. Enable the **Analyze user tasks, events, and counters** option before profiling. For more information, see [Task Analysis](#) topic.

Usage Example

The following code snippet creates a domain and a couple of tasks at global scope.

```
#include "ittnotify.h"

void do_foo(double seconds);

__itt_domain* domain = __itt_domain_create("MyTraces.MyDomain");
__itt_string_handle* shMyTask = __itt_string_handle_create("My Task");
__itt_string_handle* shMySubtask = __itt_string_handle_create("My SubTask");

void BeginFrame() {
    __itt_task_begin(domain, __itt_null, __itt_null, shMyTask);
    do_foo(1);
}
```

```

void DoWork() {
    __itt_task_begin(domain, __itt_null, __itt_null, shMySubtask);
    do_foo(1);
    __itt_task_end(domain);
}
void EndFrame() {
    do_foo(1);
    __itt_task_end(domain);
}

int main() {
    BeginFrame();
    DoWork();
    EndFrame();
    return 0;
}

#ifndef _WIN32
#include <ctime>

void do_foo(double seconds) {
    clock_t goal = (clock_t)((double)clock() + seconds * CLOCKS_PER_SEC);
    while (goal > clock());
}
#else
#include <time.h>

#define NSEC 1000000000
#define TYPE long

void do_foo(double sec) {
    struct timespec start_time;
    struct timespec current_time;

    clock_gettime(CLOCK_REALTIME, &start_time);
    while(1) {
        clock_gettime(CLOCK_REALTIME, &current_time);
        TYPE cur_nsec=(long)((current_time.tv_sec-start_time.tv_sec)*NSEC +
current_time.tv_nsec - start_time.tv_nsec);
        if(cur_nsec>=0)
            break;
    }
}
#endif

```

See Also

[Basic Usage and Configuration](#)

[Minimize ITT API Overhead](#)

[Task Analysis](#)

[View Instrumentation and Tracing Technology \(ITT\) API Task Data in Intel® VTune™ Profiler](#)

Frame API

Use the frame API to insert calls to the desired places in your code and analyze performance per frame, where frame is the time period between frame begin and end points. When frames are displayed in Intel®VTune™ Profiler, they are displayed in a separate track, so they provide a way to visually separate this data from normal task data.

Frame API is a per-process function that works in resumed state. This function does not work in paused state.

You can run the frame analysis to:

- Analyze Windows OS game applications that use DirectX* rendering.
- Analyze graphical applications performing repeated calculations.
- Analyze transaction processing on a per transaction basis to discover input cases that cause bad performance.

Frames represent a series of non-overlapping regions of Elapsed time. Frames are global in nature and not connected with any specific thread. ITT APIs that enable analyzing code frames and presenting the analysis data.

Adding Frame API to Your Code

Use This Primitive	To Do This						
<pre><code>__itt_domain *ITTAPI __itt_domain_create (const char *name)</code></pre>	<p>For a domain name, the URI naming style is recommended, for example, com.my_company.my_application. The set of domains is expected to be static over the application's execution time, therefore, there is no mechanism to destroy a domain.</p> <p>Any domain can be accessed by any thread in the process, regardless of which thread created the domain. This call is thread-safe.</p> <table border="1" data-bbox="763 1178 1457 1273"> <tr> <td data-bbox="780 1189 918 1262">[in] na] me</td><td data-bbox="975 1189 1171 1220">Name of domain</td></tr> </table>	[in] na] me	Name of domain				
[in] na] me	Name of domain						
<pre><code>void __itt_frame_begin_v3(const __itt_domain *domain, __itt_id *id);</code></pre>	<p>Define the beginning of the frame instance. An <code>__itt_frame_begin_v3</code> call must be paired with an <code>__itt_frame_end_v3</code> call.</p> <p>Successive calls to <code>__itt_frame_begin_v3</code> with the same ID are ignored until a call to <code>__itt_frame_end_v3</code> with the same ID.</p> <table border="1" data-bbox="763 1526 1457 1924"> <tr> <td data-bbox="780 1537 837 1569">[in]</td><td data-bbox="1008 1537 1106 1569">domain</td><td data-bbox="1237 1537 1432 1643">The domain for this frame instance</td></tr> <tr> <td data-bbox="780 1664 837 1695">[in]</td><td data-bbox="1008 1664 1041 1695">id</td><td data-bbox="1237 1664 1432 1913">The instance ID for this frame instance. Can be NULL, in which case the next call to <code>__itt_frame_</code></td></tr> </table>	[in]	domain	The domain for this frame instance	[in]	id	The instance ID for this frame instance. Can be NULL, in which case the next call to <code>__itt_frame_</code>
[in]	domain	The domain for this frame instance					
[in]	id	The instance ID for this frame instance. Can be NULL, in which case the next call to <code>__itt_frame_</code>					

Use This Primitive	To Do This						
<pre data-bbox="164 578 719 650">void __itt_frame_end_v3(const __itt_domain *domain, __itt_id *id);</pre>	<p data-bbox="1241 283 1426 536">end_v3.htm" >__itt_frame_end_v3 with NULL as the <i>id</i> parameter designates the end of the frame.</p> <p data-bbox="763 578 1432 811">Define the end of the frame instance. A <code>__itt_frame_end_v3</code> call must be paired with a <code>__itt_frame_begin_v3</code> call. The first call to <code>__itt_frame_end_v3</code> with a given ID ends the frame. Successive calls with the same ID are ignored, as are calls that do not have a matching <code>__itt_frame_begin_v3</code> call.</p> <table border="1" data-bbox="763 819 1472 1186"> <tr> <td data-bbox="768 825 866 889">[in]</td> <td data-bbox="866 825 1144 889"><i>domain</i></td> <td data-bbox="1144 825 1472 931">The domain for this frame instance</td> </tr> <tr> <td data-bbox="768 931 866 1186">[in]</td> <td data-bbox="866 931 1144 1186"><i>id</i></td> <td data-bbox="1144 931 1472 1186">The instance ID for this frame instance, or NULL for the current instance</td> </tr> </table>	[in]	<i>domain</i>	The domain for this frame instance	[in]	<i>id</i>	The instance ID for this frame instance, or NULL for the current instance
[in]	<i>domain</i>	The domain for this frame instance					
[in]	<i>id</i>	The instance ID for this frame instance, or NULL for the current instance					

NOTE

The analysis types based on the hardware event-based sampling collection are limited to 64 distinct frame domains.

Guidelines for Frame API Usage

- Use the frame API to denote the frame begin point and end point. Consider a frame as the time period between frame begin and end points.
- VTune Profiler does not attribute the time/samples between `__itt_frame_end_v3()` and `__itt_frame_begin_v3()` to any program unit and displays it as [Unknown] in the viewpoint.
- If there are consecutive `__itt_frame_begin_v3` calls in the same domain, treat it as a `__itt_frame_end_v3/__itt_frame_begin_v3` pair.
- Recursive/nested/overlapping frames for the same domain are not allowed.
- The `__itt_frame_begin_v3()` and `__itt_frame_end_v3()` calls can be made from different threads.
- The recommended maximum rate for calling the frame API is 1000 frames per second. A higher rate may result in large product memory consumption and slow finalization.

Usage Example

The following example uses the frame API to capture the Elapsed times for the specified code sections.

```
#include "ittnotify.h"

__itt_domain* pD = __itt_domain_create( L"My Domain" );

pD->flags = 1; /* enable domain */

for (int i = 0; i < getItemCount(); ++i)
{
    __itt_frame_begin_v3(pD, NULL);
    do_foo();
    __itt_frame_end_v3(pD, NULL);
}

...

__itt_frame_begin_v3(pD, NULL);
do_foo_1();
__itt_frame_end_v3(pD, NULL);

...

__itt_frame_begin_v3(pD, NULL);
do_foo_2();
__itt_frame_end_v3(pD, NULL);
```

See Also

[Basic Usage and Configuration](#)

[Viewing ITT API Task Data in Intel VTune Profiler](#)

Histogram API

Use the Histogram API to define histograms that display arbitrary data in histogram form in Intel® VTune™ Profiler.

The Histogram API enables you to define custom histogram graphs in your code to display arbitrary data of your choice in VTune Profiler.

Histograms can be especially useful for showing statistics that can be split by individual units for cross-comparison.

For example, you can use this API in your workload to:

- Track load distribution
- Track resource utilization
- Identify oversubscribed or underutilized worker nodes

Any histogram instance can be accessed by any thread in the process, regardless of which thread created the histogram. The Histogram API call is thread-safe.

NOTE

By default, Histogram API data collection and visualization are available in the [Input and Output analysis](#) only. To see the histogram in the result of other analysis types, [create a custom analysis](#) based on the pre-defined analysis type you are interested in, and enable the **Analyze user histogram** checkbox in the [custom analysis options](#).

Define and Create Histogram

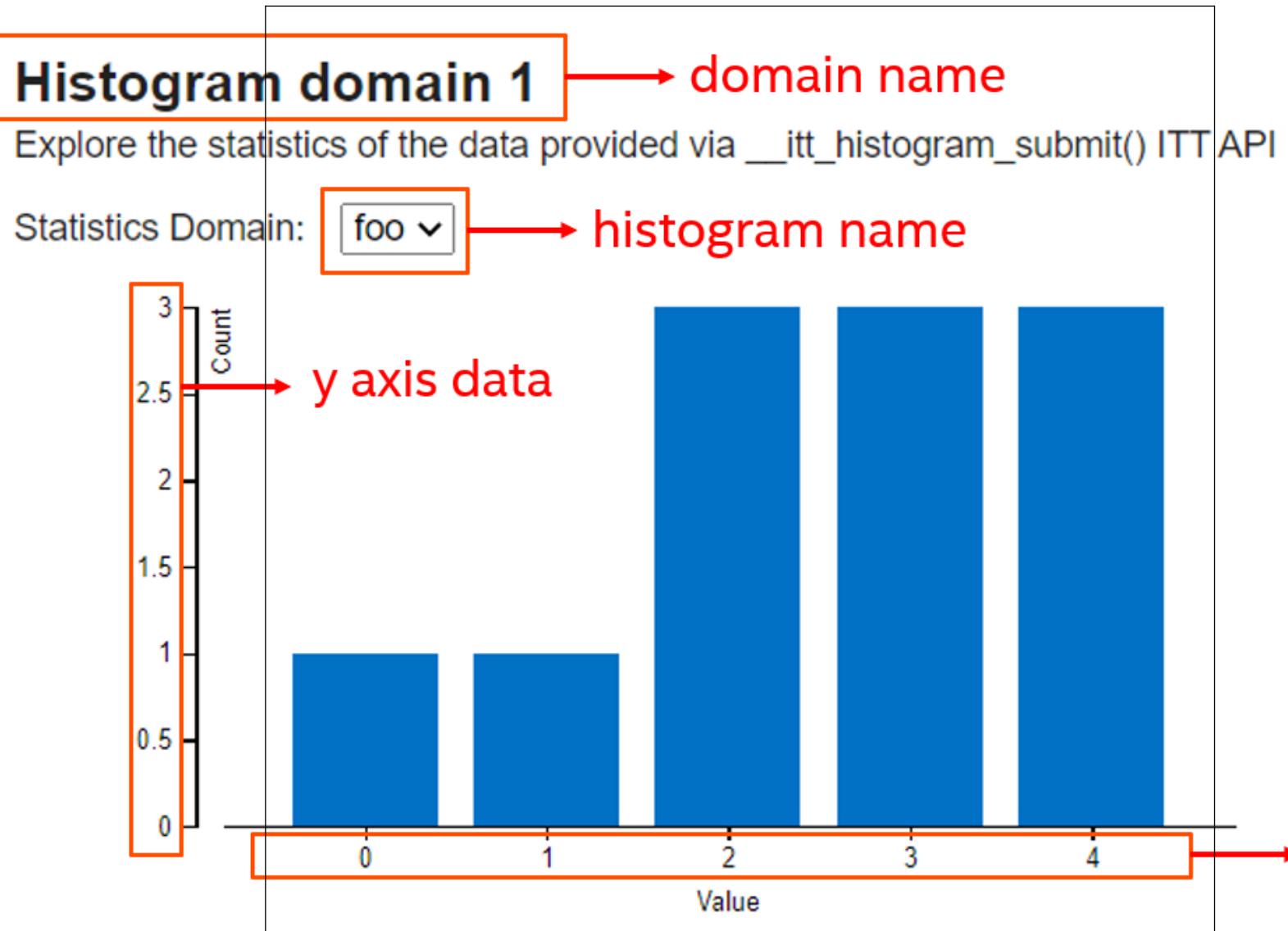
Before creating the histogram, an [ITT API Domain](#) must be created. The pointer to this domain is then passed to the primitive.

The domain name provides a heading for the histogram section on the **Summary** tab of VTune Profiler result.

One domain can combine any number of histograms. However, the name of the histogram must be unique within the same domain.

Parameters of the primitives:

[in]	domain	Domain controlling the call
[in]	name	Histogram name
[in]	x_axis_type	Type of X axis data
[in]	y_axis_type	Type of Y axis data



Use This Primitive	To Do This
<pre data-bbox="151 234 719 798"> __itt_histogram* __itt_histogram_create(__itt_domain* domain, const char* name, __itt_metadata_type x_axis_type, __itt_metadata_type y_axis_type); __itt_histogram* __itt_histogram_createA(__itt_domain* domain, const char* name, __itt_metadata_type x_axis_type, __itt_metadata_type y_axis_type); __itt_histogram* __itt_histogram_createW(__itt_domain* domain, const wchar_t* name, __itt_metadata_type x_axis_type, __itt_metadata_type y_axis_type); </pre>	<p>Create a histogram instance with the specified domain, name, and data type on Linux* and Android* OS.</p> <p>Create a histogram instance with the specified domain, name, and data type on Windows* OS for ASCII strings (char).</p> <p>Create a histogram instance with the specified domain, name, and data type on Windows* OS for UNICODE strings (wchar_t).</p>

Submit Data to Histogram

Parameters of the primitives:

[in]	histogram	Histogram instance to submit data to
[in]	length	Number of elements in submitted axis data array
[in]	x_axis_data	Array containing X axis data (may be NULL).
[in]	y_axis_data	If x_axis_data is NULL, VTune Profiler uses the indices of the y_axis_data array. Array containing Y axis data.

Primitives:

Use This Primitive	To Do This
<pre data-bbox="151 1419 719 1831"> void __itt_histogram_submit(__itt_histogram* histogram, size_t length, void* x_axis_data, void* y_axis_data); </pre>	<p>Submit user statistics for the selected histogram instance.</p> <p>Array data for the Y-axis is mapped to array data for the X-axis, similar to coordinates of a point on a 2D plane.</p> <p>Data submitted during workload run is summarized into one common histogram for all calls of this primitive.</p> <p>It is recommended to determine an efficient interval between data submissions to lower collection overhead.</p>

Usage Example

The following example creates a histogram to store worker thread statistics:

```
#include "ittnotify.h"
#include "ittnotify_types.h"

void submit_stats()
{
    // Create domain
    __itt_domain* domain = __itt_domain_create("Histogram statistics domain");

    // Create histogram
    __itt_histogram* histogram = __itt_histogram_create(domain, "Worker TID 13454",
__itt_metadata_u64, __itt_metadata_u64);

    // Fill the statistics arrays with profiling data:
    uint64_t* x_stats, y_stats;
    size_t array_size;
    get_worker_stats(x_stats, y_stats, array_size);

    // Submit histogram statistics:
    __itt_histogram_submit(histogram, array_size, x_stats, y_stats);
}
```

Basic Usage and Configuration

Domain API

User-Defined Synchronization API

While the Intel® VTune™ Profiler supports a significant portion of the Windows* OS and POSIX* APIs, it is often useful for you to define your own synchronization constructs. Any specially built constructs that you create are not normally tracked by the VTune Profiler. However, the VTune Profiler includes the synchronization API to help you gather statistical information related to user-defined synchronization constructs.

The User-Defined Synchronization API is a per-thread function that works in resumed state. This function does not work in paused state.

Synchronization constructs may generally be modeled as a series of signals. One thread or many threads may wait for a signal from another group of threads telling them they may proceed with some action. By tracking when a thread begins waiting for a signal, and then noting when the signal occurs, the synchronization API can take a user-defined synchronization object and give you an understanding of your code. The API uses memory handles along with a set of primitives to gather statistics on the user-defined synchronization object.

NOTE

The User-Defined Synchronization API works with the **Threading** analysis type.

- [Using User-Defined Synchronization API in Your Code](#)
- [Usage Example: User-Defined Spin-Waits](#)
- [Usage Example: User-Defined Synchronized Critical Section](#)
- [Usage Example: User-Level Synchronized Barrier](#)

Using User-Defined Synchronization API in Your Code

The following table describes the user-defined synchronization API primitives, available for use on Windows* and Linux* operating systems:

Use This Primitive	To Do This
<pre>void __itt_sync_create (void *addr, const __itt_char *objtype, const __itt_char *objname, int attribute)</pre>	Register the creation of a sync object using char or Unicode string.
<pre>void __itt_sync_rename (void *addr, const __itt_char *name)</pre>	Assign a name to a sync object using char or Unicode string, after it was created.
<pre>void __itt_sync_destroy (void *addr)</pre>	Track lifetime of the destroyed object.
<pre>void __itt_sync_prepare (void *addr)</pre>	Enter spin loop on user-defined sync object.
<pre>void __itt_sync_cancel (void *addr)</pre>	Quit spin loop without acquiring spin object.
<pre>void __itt_sync_acquired (void *addr)</pre>	Define successful spin loop completion (sync object acquired).
<pre>void __itt_sync_releasing (void *addr)</pre>	Start sync object releasing code. This primitive is called before the lock release call.

Each API call has a single parameter, `addr`. The address, not the value, is used to differentiate between two or more distinct custom synchronization objects. Each unique address enables the VTune Profiler to track a separate custom object. Therefore, to use the same custom object to protect access in different parts of your code, use the same `addr` parameter around each.

When properly embedded in your code, the primitives tell the VTune Profiler when the code is attempting to perform some type of synchronization. Each `prepare` primitive must be paired with a `cancel` or `acquired` primitive.

Each user-defined synchronization construct may involve any number of synchronization objects. Each synchronization object must be triggered off of a unique memory handle, which the user-defined synchronization API uses to track the object. Any number of synchronization objects may be tracked at one time using the user-defined synchronization API, as long as each object uses a unique memory pointer. You can think of this as modeling objects similar to the `WaitForMultipleObjects` function in the Windows* OS API. You can create more complex synchronization constructs out of a group of synchronization objects; however, it is not advisable to interlace different user-defined synchronization constructs as this results in incorrect behavior.



API Usage Tips

The user-defined synchronization API requires proper placement of the primitives within your code. Appropriate usage of the user-defined synchronization API can be accomplished by following these guidelines:

- Put a `prepare` primitive immediately before the code that attempts to obtain access to a synchronization object.
- Put either a `cancel` primitive or an `acquired` primitive immediately after your code is no longer waiting for a synchronization object.
- The `releasing` primitive should be used immediately before the code signals that it no longer holds a synchronization object.
- When using multiple `prepare` primitives to simulate any construct that waits for multiple objects, the last individual `cancel` or `acquired` primitive on an object related to the group of `prepare` primitives determines if the behavior of the construct is a `cancel` or `acquired` respectively.
- The time between a `prepare` primitive and an `acquired` primitive may be considered impact time

- The time between a `prepare` primitive and a `cancel` primitive is considered blocking time, even though the processor does not necessarily block.
- Improper use of the user-defined synchronization API results in incorrect statistical data.

Usage Example: User-Defined Spin-Waits

The `prepare` API indicates to the VTune Profiler that the current thread is about to begin waiting for a signal on a memory location. This call must occur before you invoke the user synchronization construct. The `prepare` API must always be paired with a call to either the `acquired` or `cancel` API.

The following code snippet shows the use of the `prepare` and `acquired` API used in conjunction with a user-defined spin-wait construct:

```
long spin = 1;
. . .
. . .
__itt_sync_prepare((void *) &spin );
while(ResourceBusy);
// spin wait;
__itt_sync_acquired((void *) &spin );
```

Using the `cancel` API may be applicable to other scenarios where the current thread tests the user synchronization construct and decides to do something useful instead of waiting for a signal from another thread. See the following code example:

```
long spin = 1;
. . .
. . .
__itt_sync_prepare((void *) &spin );
while(ResourceBusy)
{
    __itt_sync_cancel((void *) &spin );

    //
    // Do useful work
    //
    . . .
    . . .
    //
    // Once done with the useful work, this construct will test the
    // lock variable and try to acquire it again. Before this can
    // be done, a call to the prepare API is required.
    //
    __itt_sync_prepare((void *) &spin );
}
__itt_sync_acquired((void *) &spin);
```

After you acquire a lock, you must call the releasing API before the current thread releases the lock. The following example shows how to use the releasing API:

```
long spin = 1;
. . .
. . .
__itt_sync_releasing((void *) &spin );
// Code here should free the resource
```

Usage Example: User-Defined Synchronized Critical Section

The following code snippet shows how to create a critical section construct that can be tracked using the user-defined synchronization API:

```
CSEnter()
{
    __itt_sync_prepare((void*) &cs);
    while(LockIsUsed)
    {
        if(LockIsFree)
        {
            // Code to actually acquire the lock goes here
            __itt_sync_acquired((void*) &cs);
        }
        if(timeout)
        {
            __itt_sync_cancel((void*) &cs );
        }
    }
}
CSLeave()
{
if(LockIsMine)
{
    __itt_sync_releasing((void*) &cs);
    // Code to actually release the lock goes here
}
}
```

This simple critical section example demonstrates how to use the user-defined synchronization primitives. When looking at this example, note the following points:

- Each `prepare` primitive is paired with an `acquired` primitive or a `cancel` primitive.
- The `prepare` primitive is placed immediately before the user code begins waiting for the user lock.
- The `acquired` primitive is placed immediately after the user code actually obtains the user lock.
- The `releasing` primitive is placed before the user code actually releases the user lock. This ensures that another thread does not call the `acquired` primitive before the VTune Profiler realizes that this thread has released the lock.

Usage Example: User-Level Synchronized Barrier

Higher level constructs, such as barriers, are also easy to model using the synchronization API. The following code snippet shows how to create a barrier construct that can be tracked using the synchronization API:

```
Barrier()
{
    teamflag = false;
    __itt_sync_releasing((void *) &counter);
    InterlockedIncrement(&counter); // use the atomic increment primitive appropriate to your
OS and compiler

    if( counter == thread count )
    {
        __itt_sync_acquired((void *) &counter);
        __itt_sync_releasing((void *) &teamflag);
        teamflag = true;
        counter = 0;
    }
    else
}
```

```
{
    __ itt_sync_prepare((void *) &teamflag);
    Wait for team flag
    __ itt_sync_acquired((void *) &teamflag);
}
}
```

When looking at this example, note the following points:

- There are two synchronization objects in this barrier code. The `counter` object is used to do a gather-like signaling from all the threads to the final thread indicating that each thread has entered the barrier. Once the last thread hits the barrier it uses the `teamflag` object to signal all the other threads that they may proceed.
- As each thread enters the barrier it calls `__itt_sync_releasing` to tell the VTune Profiler that it is about to signal the last thread by incrementing `counter`
- The last thread to enter the barrier calls `__itt_sync_acquired` to tell the VTune Profiler that it was successfully signaled by all the other threads.
- The last thread to enter the barrier calls `__itt_sync_releasing` to tell the VTune Profiler that it is going to signal the barrier completion to all the other threads by setting `teamflag`
- Each thread, except the last, calls the `__itt_sync_prepare` primitive to tell the VTune Profiler that it is about to start waiting for the `teamflag` signal from the last thread.
- Finally, before leaving the barrier, each thread calls the `__itt_sync_acquired` primitive to tell the VTune Profiler that it successfully received the end-of-barrier signal.

See Also

[Basic Usage and Configuration](#)

Event API

The event API is used to observe when demarcated events occur in your application, or to identify how long it takes to execute demarcated regions of code. Set annotations in the application to demarcate areas where events of interest occur. After running analysis, you can see the events marked in the Timeline pane.

Event API is a per-thread function that works in resumed state. This function does not work in paused state.

NOTE

- On Windows* OS platforms you can define Unicode to use a wide character version of APIs that pass strings. However, these strings are internally converted to ASCII strings.
- On Linux* OS platforms only a single variant of the API exists.

Use This Primitive	To Do This
<pre>__itt_event __itt_event_create(const __itt_char *name, int namelen);</pre> <pre>int __itt_event_start(__itt_event event);</pre>	<p>Create an event type with the specified name and length. This API returns a handle to the event type that should be passed into the following event start and event end APIs as a parameter. The <code>namelen</code> parameter refers to the name length in number of characters, not the number of bytes.</p> <p>Call this API with your previously created event type handle to register an instance of the event. Event start appears in the Timeline pane display as a tick mark.</p>

Use This Primitive	To Do This
<code>int __itt_event_end(__itt_event event);</code>	Call this API following a call to <code>__itt_event_start()</code> to show the event as a tick mark with a duration line from start to end. If this API is not called, this event appears in the Timeline pane as a single tick mark.

Guidelines for Event API Usage

- An `__itt_event_end()` is always matched with the nearest preceding `__itt_event_start()`. Otherwise, the `__itt_event_end()` call is matched with the nearest unmatched `__itt_event_start()` preceding it. Any intervening events are nested.
- You can nest user events of the same or different type within each other. In the case of nested events, the time is considered to have been spent only in the most deeply nested user event region.
- You can overlap different ITT API events. In the case of overlapping events the time is considered to have been spent only in the event region with the later `__itt_event_start()`. Unmatched `__itt_event_end()` calls are ignored.

NOTE

To see events and user tasks in your results, [create a custom analysis](#) (based on the pre-defined analysis you are interested in) and select the **Analyze user tasks, events and counters** checkbox in the analysis settings.

Usage Example: Creating and Marking Single Events

The `__itt_event_create` API returns a new event handle that you can subsequently use to mark user events with the `__itt_event_start` API. In this example, two event type handles are created and used to set the start points for tracking two different types of events.

```
#include "ittnotify.h"

__itt_event mark_event = __itt_event_create( "User Mark", 9 );
__itt_event frame_event = __itt_event_create( "Frame Completed", 15 );
...
__itt_event_start( mark_event );
...
for( int f ; f<number_of_frames ; f++ ) {
    ...
    __itt_event_start( frame_event );
}
```

Usage Example: Creating and Marking Event Regions

The `__itt_event_start` API can be followed by an `__itt_event_end` API to define an event region, as in the following example:

```
#include "ittnotify.h"

__itt_event render_event = __itt_event_create( "Rendering Phase", 15 );
...
for( int f ; f<number_of_frames ; f++ ) {
    ...
    do_stuff_for_frame();
    ...
    __itt_event_start( render_event );
```

```

...
do_rendering_for_frame();
...
__itt_event_end( render_event );
...
}

```

See Also

[Basic Usage and Configuration](#)

[View Instrumentation and Tracing Technology \(ITT\) API Task Data in Intel® VTune™ Profiler](#)

Counter API

Use the Counter API to observe user-defined global characteristic counters that are unknown to VTune Profiler. For example, system on a chip (SoC) development benefits from several counters representing different parts of the SoC to count some hardware characteristics.

Define and create a counter object

Use these primitives:

```

__itt_counter
__itt_counter_create(const char *name, const char *domain);
__itt_counter_createA(const char *name, const char *domain);
__itt_counter_createW(const wchar_t *name, const wchar_t *domain);
__itt_counter_create_typed (const char *name, const char *domain, __itt_metadata_type
type);
__itt_counter __itt_counter_create_typedA __itt_counter_create_typedA(const char *name,
const char *domain, __itt_metadata_type type)
__itt_counter __itt_counter_create_typedW __itt_counter_create_typedW(const wchar_t
*name, const wchar_t *domain, __itt_metadata_type type)
__itt_counter_create_v3(__itt_domain* domain, const char* name, __itt_metadata_type
type);

```

A counter name and domain name should be specified. To load a specialized type of data, specify the counter type. By default the unsigned int64 type is used.

Parameters of the primitives:

[in]	domain	Counter domain
[in]	name	Counter name
[in]	type	Counter type

Increment/decrement a counter value

Use these primitives:

```

__itt_counter_inc (__itt_counter id);
__itt_counter_inc_delta(__itt_counter id, unsigned long long value);
__itt_counter_dec(__itt_counter id);
__itt_counter_dec_delta(__itt_counter id, unsigned long long value);

```

NOTE

These primitives are applicable to uint64 counters only.

Directly set the counter value

Use:

```
__itt_counter_set_value(__itt_counter id, void *value_ptr);
__itt_counter_set_value_v3(__itt_counter counter, void *value_ptr);
```

Parameters of the primitive:

[in]	id	Counter ID
[in]	value_ptr	Counter value

Remove an existing counter

Use:

```
__itt_counter_destroy(__itt_counter id);
```

Usage Example

This example creates a counter that measures temperature and memory usage metrics:

```
#include "ittnotify.h"

__itt_counter temperatureCounter = __itt_counter_create("Temperature", "Domain");
__itt_counter memoryUsageCounter = __itt_counter_create("Memory Usage", "Domain");
unsigned __int64 temperature;

while (...)

{
    ...
    temperature = getTemperature();
    __itt_counter_set_value(temperatureCounter, &temperature);

    __itt_counter_inc_delta(memoryUsageCounter, getAllocatedMemSize());
    __itt_counter_dec_delta(memoryUsageCounter, getDeallocatedMemSize());
    ...

}

__itt_counter_destroy(temperatureCounter);
__itt_counter_destroy(memoryUsageCounter);
```

See Also

[Basic Usage and Configuration](#)

Context Metadata API

Use context metadata to collect counter-based metrics and attribute them to hardware topology like:

- PCIe devices
- Block devices
- CPU cores
- Threads

With the Context Metadata API, you can define custom counters in your code with special attributes. You can also get a set of metrics for the collected data in any classical form of data representation in Intel® VTune™ Profiler.

Availability:

By default, the Context Metadata API for data collection and visualization is available in the [Input and Output analysis](#) only.

To see this data when running other analysis types,

1. [Create a custom analysis](#) based on the predefined analysis type of your interest.
2. In [custom analysis options](#), enable the **Analyze all ITT API user data** checkbox.

Define and create a counter object

Use this structure to store context metadata:

```
_itt_context_metadata
{
    _itt_context_type type;      /*!< Type of the context metadata value */
    void* value;                /*!< Pointer to context metadata value itself */
}
```

The structure accepts the following types of context metadata:

_itt_context_type	Value	Description
<code>_itt_context_name</code>	ASCII string char*/ Unicode string wchar_t* type	The name of the counter-based metric. This value is required.
<code>_itt_context_device</code>	ASCII string char*/ Unicode string wchar_t* type	Statistics subdomain to break down the counter samples (for example, network port ID, disk partition, etc.)
<code>_itt_context_units</code>	ASCII string char*/ Unicode string wchar_t* type	Units of measurement. For measurement of time, use the ns/us/ms/s units to correct data representation in VTune Profiler.
<code>_itt_context_pci_addr</code>	ASCII string char*/ Unicode string wchar_t* type	PCI address of device to associate with the counter.
<code>_itt_context_tid</code>	Unsigned 64-bit integer type	Thread ID to associate with the counter.
<code>_itt_context_bandwidth_flag</code>	Unsigned 64-bit integer type (0,1)	If this flag is set to 1, calculate latency histogram and counter/sec timeline distribution.
<code>_itt_context_latency_flag</code>	Unsigned 64-bit integer type (0,1)	If this flag is set to 1, calculate the throughput histogram and counter/sec timeline distribution.
<code>_itt_context_on_thread_flag</code>	Unsigned 64-bit integer type (0,1)	If this flag is set to 1, show the counter on top of the Thread graph as percentage of the CPU Time distribution.

Before you associate context metadata with a counter, you should create an ITT API Domain and ITT API Counter Instances.

The domain name provides a heading for the section of metrics for the counters in the results of VTune Profiler. A single domain can combine data from any number of counters. However, the name of the counters must be unique within the same domain.

You can combine different counters under a single metric of the Context Metadata.

Add context information

Once you have created all objects, you can add context information for the selected counters. Use these primitives:

```
_itt_bind_context_metadata_to_counter(_itt_counter counter, size_t length,
_itt_context_metadata* metadata);
```

Parameters of the primitive:

Type	Parameter	Description
[in]	<code>__itt_counter counter</code>	Pointer to the counter instance associated with the context metadata
[in]	<code>size_t length</code>	Number of elements in the array of context metadata
[in]	<code>__itt_context_metadata* metadata</code>	Pointer to the array of context metadata

To create counter instances and submit counter data, use:

```
__itt_counter_create_v3(__itt_domain* domain, const char* name, __itt_metadata_type type);
__itt_counter_set_value_v3(__itt_counter counter, void *value_ptr);
```

Usage Example

This example creates counters with context metadata that measures random read operation metrics for an SSD NVMe device:

```
#include "ittnotify.h"
#include "ittnotify_types.h"

// Create domain and counters:
__itt_domain* domain = __itt_domain_create("ITT API collected data");
__itt_counter counter_read_op = __itt_counter_create_v3(domain, "Read Operations",
__itt_metadata_u64);
__itt_counter counter_read_mb = __itt_counter_create_v3(domain, "Read Megabytes",
__itt_metadata_u64);
__itt_counter counter_spin_time = __itt_counter_create_v3(domain, "Spin Time",
__itt_metadata_u64);

// Create context metadata:
__itt_context_metadata metadata_read_op[] = {
    {__itt_context_name, "Reads"},
    {__itt_context_device, "NVMe SSD Intel DC 660p"},
    {__itt_context_units, "Operations"},
    {__itt_context_pci_addr, "0001:10:00.1"},
    {__itt_context_latency_flag, &true_flag}
};
__itt_context_metadata metadata_read_mb[] = {
    {__itt_context_name, "Read"},
    {__itt_context_device, "NVMe SSD Intel DC 660p"},
    {__itt_context_units, "MB"},
    {__itt_context_pci_addr, "0001:10:00.1"},
    {__itt_context_bandwidth_flag, &true_flag}
};
__itt_context_metadata metadata_spin_time[] = {
    {__itt_context_name, "Spin Time"},
    {__itt_context_device, "NVMe SSD Intel DC 660p"},
    {__itt_context_units, "ms"},
    {__itt_context_tid, &thread_id}
};

// Bind context metadata to counters:
__itt_bind_context_metadata_to_counter(counter_read_op, n, metadata_read_op);
__itt_bind_context_metadata_to_counter(counter_read_mb, n, metadata_read_mb);
```

```

__itt_bind_context_metadata_to_counter(counter_spin_time, n, metadata_spin_time);

while(1)
{
    // Get collected data:
    uint64_t read_op    = get_user_read_operation_num();
    uint64_t read_mb    = get_user_read_megabytes_num();
    uint64_t spin_time = get_user_spin_time();

    // Dump collected data:
    __itt_counter_set_value_v3(counter_read_op, &read_op);
    __itt_counter_set_value_v3(counter_read_mb, &read_mb);
    __itt_counter_set_value_v3(counter_spin_time, &spin_time);
}

```

See Also

[Basic Usage and Configuration](#)

[Domain API](#)

[Counters API](#)

Load Module API

You can use the Load Module API in your code to analyze a module that was loaded in an alternate location that cannot otherwise be tracked by Intel VTune Profiler. For example, this would allow you to analyze code that is typically executed in an isolated environment to which there is no visibility of the code. This API allows you to explicitly set the module location in an address space for analysis by VTune Profiler.

Use This Primitive	To Do This
<code>void __itt_module_loadW (void* start_addr, void* end_addr, const wchar_t* path)</code>	Call this function after the relocation of a module. Provide the new start and end addresses for the module and the full path to the module on the local drive.
<code>void __itt_module_loadA(void* start_addr, void* end_addr, const char* path)</code>	Call this function after the relocation of a module. Provide the new start and end addresses for the module and the full path to the module on the local drive.
<code>void __itt_module_load(void* start_addr, void* end_addr, const char* path)</code>	Call this function after the relocation of a module. Provide the new start and end addresses for the module and the full path to the module on the local drive.

Usage Example

```

#include "ittnotify.h"
__itt_module_load(relocatedBaseModuleAddress, relocatedEndModuleAddress, '/some/path/to/dynamic/
library.so');

```

See Also

[Basic Usage and Configuration](#)

[Instrumenting Your Application](#)

[Minimizing ITT API Overhead](#)

Memory Allocation APIs

Intel® VTune™ Profiler provides a set of APIs to help it identify the semantics of your `malloc`-like heap management functions.

Annotating your code with these APIs allows VTune Profiler to correctly determine memory objects as part of Memory Access Analysis.

Usage Tips

Follow these guidelines when using the memory allocation APIs:

- Create *wrapper* functions for your routines, and put the `__itt_heap_*_begin` and `__itt_heap_*_end` calls in these functions.
- Allocate a unique domain for each pair of `allocate`/`free` functions when calling `__itt_heap_function_create`. This allows the VTune Profiler to verify a matching `free` function is called for every `allocate` function call.
- Annotate the beginning and end of every `allocate` function and `free` function.
- Call all function pairs from the same stack frame, otherwise the VTune Profiler assumes an exception occurred and the allocation attempt failed.
- Do not call an `end` function without first calling the matching `begin` function.

Using Memory Allocation APIs in Your Code

Use This	To Do This
<pre> typedef void* __itt_heap_function; __itt_heap_function __itt_heap_function_create(const __itt_char* <name>, const __itt_char* <domain>); </pre> <pre> void __itt_heap_allocate_begin(__itt_heap_function <h>, size_t <size>, int <initialized>); void __itt_heap_allocate_end(__itt_heap_function <h>, void** <addr>, size_t <size>, int <initialized>); void __itt_heap_free_begin(__itt_heap_function <h>, void* <addr>); void __itt_heap_free_end(__itt_heap_function <h>, void* <addr>); void __itt_heap_reallocate_begin(__itt_heap_function <h>, void* <addr>, size_t <new_size>, int <initialized>); </pre>	<p>Declare a handle type to match <code>begin</code> and <code>end</code> calls and domains.</p> <ul style="list-style-type: none"> • <i>name</i> = Name of the function you want to annotate. • <i>domain</i> = String identifying a matching set of functions. For example, if there are three functions that all work with <code>my_struct</code>, such as <code>alloc_my_structs</code>, <code>free_my_structs</code>, and <code>realloc_my_structs</code>, pass the same domain to all three <code>__itt_heap_function_create()</code> calls. <p>Identify allocation functions.</p> <ul style="list-style-type: none"> • <i>h</i> = Handle returned when this function's name was passed to <code>__itt_heap_function_create()</code>. • <i>size</i> = Size in bytes of the requested memory region. • <i>initialized</i> = Flag indicating if the memory region will be initialized by this routine. • <i>addr</i> = Pointer to the address of the memory region this function has allocated, or 0 if the allocation failed. <p>Identify deallocation functions.</p> <ul style="list-style-type: none"> • <i>h</i> = Handle returned when this function's name was passed to <code>__itt_heap_function_create()</code>. • <i>addr</i> = Pointer to the address of the memory region this function is deallocating. <p>Identify reallocation functions.</p> <p>Note that <code>itt_heap_reallocate_end()</code> must be called after the attempt even if no memory is returned. VTune Profiler assumes C-runtime <code>realloc</code> semantics.</p>

Use This	To Do This
<pre>void __itt_heap_reallocate_end(__itt_heap_function <h>, void* <addr>, void** <new_addr>, size_t <new_size>, int <initialized>);</pre>	<ul style="list-style-type: none"> • <i>h</i> = Handle returned when this function's name was passed to <code>__itt_heap_function_create()</code>. • <i>addr</i> = Pointer to the address of the memory region this function is reallocating. If <i>addr</i> is NULL, the VTune Profiler treats this as if it is an allocation. • <i>new_addr</i> = Pointer to a pointer to hold the address of the reallocated memory region. • <i>size</i> = Size in bytes of the requested memory region. If <i>new_size</i> is 0, the VTune Profiler treats this as if it is a deallocation.

Usage Example: Heap Allocation

```
#include <ittnotify.h>

void* user_defined_malloc(size_t size);
void user_defined_free(void *p);
void* user_defined_realloc(void *p, size_t s);

__itt_heap_function my_allocator;
__itt_heap_function my_reallocator;
__itt_heap_function my_freer;

void* my_malloc(size_t s)
{
    void* p;

    __itt_heap_allocate_begin(my_allocator, s, 0);
    p = user_defined_malloc(s);
    __itt_heap_allocate_end(my_allocator, &p, s, 0);

    return p;
}

void my_free(void *p)
{
    __itt_heap_free_begin (my_freer, p);
    user_defined_free(p);
    __itt_heap_free_end (my_freer, p);
}

void* my_realloc(void *p, size_t s)
{
    void *np;

    __itt_heap_reallocate_begin (my_reallocator, p, s, 0);
    np = user_defined_realloc(p, s);
    __itt_heap_reallocate_end(my_reallocator, p, &np, s, 0);

    return(np);
}

// Make sure to call this init routine before any calls to
// user defined allocators.
```

```

void init_itt_calls()
{
    my_allocator = __itt_heap_function_create("my_malloc", "mydomain");
    my_reallocator = __itt_heap_function_create("my_realloc", "mydomain");
    my_free = __itt_heap_function_create("my_free", "mydomain");
}

```

See Also

[Basic Usage and Configuration](#)

[Basic Usage and Configuration](#)

[Minimize ITT API Overhead](#)

JIT Profiling API

NOTE

The Instrumentation and Tracing Technology API (ITT API) and the Just-in-Time Profiling API (JIT API) are open source components. Visit the [GitHub* repository](#) to access source code and contribute.

The JIT (Just-In-Time) Profiling API provides functionality to report information about just-in-time generated code that can be used by performance tools. You need to insert JIT Profiling API calls in the code generator to report information before JIT-compiled code goes to execution. This information is collected at runtime and used by tools like Intel® VTune™ Profiler to display performance metrics associated with JIT-compiled code.

You can use the JIT Profiling API to profile such environments as dynamic JIT compilation of JavaScript code traces, JIT execution in OpenCL™ applications, Java*/.NET* managed execution environments, and custom ISV JIT engines.

The standard VTune Profiler installation contains a static part (as a static library and source files) and a profiler object. The JIT engine generating code during runtime communicates with a profiler object through the static part. During runtime, the JIT engine reports the information about JIT-compiled code stored in a trace file by the profiler object. After collection, the VTune Profiler uses the generated trace file to resolve the JIT-compiled code. If the VTune Profiler is not installed, profiling is disabled.

Use the JIT Profiling API to:

- [Profile trace-based and method-based JIT-compiled code](#)
- [Analyze split functions](#)
- [Explore inline functions](#)

JIT profiling is supported with the **Launch Application** target option for event based sampling.

Profile Trace-based and Method-based JIT-compiled Code

This is the most common scenario for using JIT Profiling API to profile trace-based and method-based JIT-compiled code:

```

#include <jitprofiling.h>

if (iJIT_IsProfilingActive() != iJIT_SAMPLING_ON) {
    return;
}

iJIT_Method_Load jmethod = {0};
jmethod.method_id = iJIT_GetNewMethodID();
jmethod.method_name = "method_name";
jmethod.class_file_name = "class_name";

```

```
jmethod.source_file_name = "source_file_name";
jmethod.method_load_address = code_addr;
jmethod.method_size = code_size;

iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED,
    (void*)&jmethod);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_SHUTDOWN, NULL);
```

Usage Tips

- If any `iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED` event overwrites an already reported method, then such a method becomes invalid and its memory region is treated as unloaded. VTune Profiler displays the metrics collected by the method until it is overwritten.
- If supplied line number information contains multiple source lines for the same assembly instruction (code location), then VTune Profiler picks up the first line number.
- Dynamically generated code can be associated with a module name. Use the `iJIT_Method_Load_V2` structure.
 - If you register a function with the same method ID multiple times, specifying different module names, then the VTune Profiler picks up the module name registered first. If you want to distinguish the same function between different JIT engines, supply different method IDs for each function. Other symbolic information (for example, source file) can be identical.

Analyze Split Functions

You can use the JIT Profiling API to analyze split functions (multiple joint or disjoint code regions belonging to the same function) including re-JITting with potential overlapping of code regions in time, which is common in resource-limited environments.

```
#include <jitprofiling.h>

unsigned int method_id = iJIT_GetNewMethodID();

iJIT_Method_Load a = {0};
a.method_id = method_id;
a.method_load_address = 0x100;
a.method_size = 0x20;

iJIT_Method_Load b = {0};
b.method_id = method_id;
b.method_load_address = 0x200;
b.method_size = 0x30;

iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&a);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&b)
```

Usage Tips

- If a `iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED` event overwrites an already reported method, then such a method becomes invalid and its memory region is treated as unloaded.
- All code regions reported with the same method ID are considered as belonging to the same method. Symbolic information (method name, source file name) will be taken from the first notification, and all subsequent notifications with the same method ID will be processed only for line number table information. So, the VTune Profiler will map samples to a source line using the line number table from the current notification while taking the source file name from the very first one.

- If you register a second code region with a different source file name and the same method ID, this information will be saved and will not be considered as an extension of the first code region, but VTune Profiler will use the source file of the first code region and map performance metrics incorrectly.
- If you register a second code region with the same source file as for the first region and the same method ID, the source file will be discarded but VTune Profiler will map metrics to the source file correctly.
- If you register a second code region with a null source file and the same method ID, provided line number info will be associated with the source file of the first code region.

Explore Inline Functions

You can use the JIT Profiling API to explore inline functions including multi-level hierarchy of nested inline methods that shows how performance metrics are distributed through them.

```
#include <jitprofiling.h>

//                                     method_id    parent_id
//   [-- c --]                      3000        2000
//           [---- d -----]          2001        1000
//   [---- b ----]                  2000        1000
// [----- a -----]                1000        n/a

iJIT_Method_Load a = {0};
a.method_id = 1000;

iJIT_Method_Inline_Load b = {0};
b.method_id = 2000;
b.parent_method_id = 1000;

iJIT_Method_Inline_Load c = {0};
c.method_id = 3000;
c.parent_method_id = 2000;

iJIT_Method_Inline_Load d = {0};
d.method_id = 2001;
d.parent_method_id = 1000;

iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&a);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED, (void*)&b);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED, (void*)&c);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED, (void*)&d);
```

Usage Tips

- Each inline (*iJIT_Method_Inline_Load*) method should be associated with two method IDs: one for itself; one for its immediate parent.
- Address regions of inline methods of the same parent method cannot overlap each other.
- Execution of the parent method must not be started until it and all its inline methods are reported.
- In case of nested inline methods an order of *iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED* events is not important.
- If any event overwrites either inline method or top parent method, then the parent, including inline methods, becomes invalid and its memory region is treated as unloaded.

See Also

[JIT Profiling API Reference](#)

[Using JIT Profiling API](#)

Basic Usage and Configuration

See prerequisites here

Using JIT Profiling API

To include JIT Profiling support, do one of the following:

- Include the following files to your source tree:
 - jitprofiling.h, located under <install-dir>\include (Windows*) or <install-dir>/include (Linux*)
 - ittnotify_config.h, ittnotify_types.h and jitprofiling.c, located under <install-dir>\sdk\src\ittnotify (Windows*) or <install-dir>/sdk/src/ittnotify (Linux*)

NOTE To locate your <install-dir> see [Installation Directory](#).

- Use the static library provided with the product:

1. Include jitprofiling.h file, located under the <install-dir>\include (Windows*) or <install-dir>/include (Linux*) directory, in your code. This header file provides all API function prototypes and type definitions.
2. Link to jitprofiling.lib (Windows*) or jitprofiling.a (Linux*), located under <install-dir>\lib32 or <install-dir>\lib64 (Windows*) or <install-dir>/lib32 or <install-dir>/lib64 (Linux*).

Use This Primitive	To Do This
<pre>int iJIT_NotifyEvent(i JIT_JVM_EVENT event_type, void *EventSpecificData); unsigned int iJIT_GetNewMethodID (void);</pre>	<p>Use this API to send a notification of event_type with the data pointed by EventSpecificData to the agent. The reported information is used to attribute samples obtained from any Intel® VTune™ Profiler collector.</p> <p>Generate a new method ID. You must use this function to assign unique and valid method IDs to methods reported to the profiler.</p> <p>This API returns a new unique method ID. When out of unique method IDs, this API function returns 0.</p>
<pre>iJIT_IsProfilingAct iveFlags iJIT_IsProfilingAct ive(void);</pre>	<p>Returns the current mode of the profiler: off, or sampling, using the iJIT_IsProfilingActiveFlags enumeration.</p> <p>This API returns iJIT_SAMPLING_ON by default, indicating that Sampling is running. It returns iJIT_NOTHING_RUNNING if no profiler is running.</p>

Lifetime of Allocated Data

You send an event notification to the agent (VTune Profiler) with event-specific data, which is a structure. The pointers in the structure refer to memory you allocated and you are responsible for releasing it. The pointers are used by the iJIT_NotifyEvent method to copy your data in a trace file, and they are not used after the iJIT_NotifyEvent method returns.

JIT Profiling API Sample Application

VTune Profiler is installed with a sample application in the jitprofiling_vtune_amp_xe.zip (Windows*) or jitprofiling_vtune_amp_xe.tgz (Linux*) that emulates the creation and execution of dynamic code. In addition, it uses the JIT profiling API to notify the VTune Profiler when it transfers execution control to dynamic code.

To install and set up the sample code:

1. Copy the jitprofiling_vtune_amp_xe.zip (Windows*) or jitprofiling_vtune_amp_xe.tgz (Linux*) file from the <install-dir>\samples\<locale>\C++ (Windows*) or <install-dir>/samples/<locale>/C++ (Linux*) directory to a writable directory or share on your system.
2. Extract the sample from the archive file.

Build jitprofiling.c in Microsoft Visual Studio*

1. Copy the jitprofiling_vtune_amp_xe.zip file from the <install-dir>\samples\<locale>\C++ directory to a writable directory or share on your system.
2. Extract the sample from the .zip file.
3. Open the jitprofiling.sln file with Microsoft Visual Studio.
4. Right-click **jitprofiling** in the **Solution Explorer** and select **Properties**. The **jitprofiling Property Pages** window opens.
5. Set the **Platform** (top of the window) to **x64**.
6. Select **Configuration Properties** > **C/C++** > **General** and add the path to the headers (<install-dir>/include) to **Additional Include Directories**.
7. Select **Configuration Properties** > **C/C++** > **Linker** > **General** and add the path to the library (<install-dir>/lib32 or <install-dir>/lib64) to **Additional Library Directories**.
8. Click **OK** to apply the changes and close the window.
9. Rebuild the solution with the new project settings.

Installation Information

Whether you downloaded Intel® VTune™ Profiler as a standalone component or with the Intel® oneAPI Base Toolkit, the default path for your <install-dir> is:

Operating System	Path to <install-dir>
Windows* OS	<ul style="list-style-type: none"> • C:\Program Files (x86)\Intel\oneAPI\ • C:\Program Files\Intel\oneAPI\ (in certain systems)
Linux* OS	<ul style="list-style-type: none"> • /opt/intel/oneapi/ for root users • \$HOME/intel/oneapi/ for non-root users
macOS*	/opt/intel/oneapi/

For OS-specific installation instructions, refer to the [VTune Profiler Installation Guide](#).

See Also

- [About JIT Profiling API](#)
- [JIT Profiling API Reference](#)
- [Basic Usage and Configuration](#)
- See prerequisites here

JIT Profiling API Reference

iJIT_NotifyEvent

Reports information about JIT-compiled code to the agent.

Syntax

```
int iJIT_NotifyEvent( iJIT_JVM_EVENT event_type, void *EventSpecificData );
```

Description

The `iJIT_NotifyEvent` function sends a notification of `event_type` with the data pointed by `EventSpecificData` to the agent. The reported information is used to attribute samples obtained from any Intel® VTune™ Profiler collector. This API needs to be called after JIT compilation and before the first entry into the JIT-compiled code.

Input Parameters

Parameter	Description
<code>iJIT_JVM_EVENT event_type</code>	Notification code sent to the agent. See a complete list of event types below.
<code>void *EventSpecificData</code>	Pointer to event specific data .

The following values are allowed for `event_type`:

<code>iJVM_EVENT_TYPE_METHOD_LOAD_FINISH_ED</code>	Send this notification after a JITted method has been loaded into memory, and possibly JIT compiled, but before the code is executed. Use the iJIT_Method_Load structure for <code>EventSpecificData</code> . The return value of <code>iJIT_NotifyEvent</code> is undefined.
<code>iJVM_EVENT_TYPE_SHUTDOWN</code>	Send this notification to terminate profiling. Use <code>NULL</code> for <code>EventSpecificData</code> . <code>iJIT_NotifyEvent</code> returns <code>1</code> on success.
<code>JVM_EVENT_TYPE_METHOD_UPDATE</code>	Send this notification to provide new content for a previously reported dynamic code. The previous content will be invalidated starting from the time of the notification. Use the iJIT_Method_Load structure for <code>EventSpecificData</code> with the following required fields: <ul style="list-style-type: none"> • <code>method_id</code> to identify the code to update • <code>method_load_address</code> to specify the start address within an identified code range where the update should be started • <code>method_size</code> to specify the length of an updated code range
<code>JVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED</code>	Send this notification when an inline dynamic code is JIT compiled and loaded into memory by the JIT engine, but before the parent code region starts executing. Use the iJIT_Method_Inline_Load structure for <code>EventSpecificData</code> .
<code>iJVM_EVENT_TYPE_METHOD_LOAD_FINISH_ED_V2</code>	Send this notification when a dynamic code is JIT compiled and loaded into memory by the JIT engine, but before the code is executed. Use the iJIT_Method_Load_V2 structure for <code>EventSpecificData</code> .

The following structures can be used for `EventSpecificData`:

iJIT_Method_Inline_Load Structure

When you use the `iJIT_Method_Inline_Load` structure to describe the JIT compiled method, use `iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED` as an event type to report it. The `iJIT_Method_Inline_Load` structure has the following fields:

Field	Description
<code>unsigned int method_id</code>	Unique method ID. Method ID cannot be smaller than 999. You must either use the API function <code>iJIT_GetNewMethodID</code> to get a valid and unique method ID, or else manage ID uniqueness and correct range by yourself.
<code>unsigned int parent_method_id</code>	Unique immediate parent's method ID. Method ID may not be smaller than 999. You must either use the API function <code>iJIT_GetNewMethodID</code> to get a valid and unique method ID, or else manage ID uniqueness and correct range by yourself.
<code>char *method_name</code>	The name of the method, optionally prefixed with its class name and appended with its complete signature. This argument cannot be set to NULL.
<code>void *method_load_address</code>	The base address of the method code. Can be NULL if the method is not JITted.
<code>unsigned int method_size</code>	The virtual address on which the method is inlined. If NULL, then data provided with the event are not accepted.
<code>unsigned int line_number_size</code>	The number of entries in the line number table. 0 if none.
<code>pLineNumberInfo line_number_table</code>	Pointer to the line numbers info array. Can be NULL if <code>line_number_size</code> is 0. See <code>LineNumberInfo</code> structure for a description of a single entry in the line number info array.
<code>char *class_file_name</code>	Class name. Can be NULL.
<code>char *source_file_name</code>	Source file name. Can be NULL.

iJIT_Method_Load Structure

When you use the `iJIT_Method_Load` structure to describe the JIT compiled method, use `iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED` as an event type to report it. The `iJIT_Method_Load` structure has the following fields:

Field	Description
<code>unsigned int method_id</code>	Unique method ID. Method ID cannot be smaller than 999. You must either use the API function <code>iJIT_GetNewMethodID</code> to get a valid and unique method ID, or else manage ID uniqueness and correct range by yourself.
<code>char *method_name</code>	The name of the method, optionally prefixed with its class name and appended with its complete signature. This argument cannot be set to NULL.
<code>void *method_load_address</code>	The base address of the method code. Can be NULL if the method is not JITted.

Field	Description
<code>unsigned int method_size</code>	The virtual address on which the method is inlined. If NULL, then data provided with the event are not accepted.
<code>unsigned int line_number_size</code>	The number of entries in the line number table. 0 if none.
<code>pLineNumberInfo line_number_table</code>	Pointer to the line numbers info array. Can be NULL if <code>line_number_size</code> is 0. See LineNumberInfo structure for a description of a single entry in the line number info array.
<code>unsigned int class_id</code>	This field is obsolete.
<code>char *class_file_name</code>	Class name. Can be NULL.
<code>char *source_file_name</code>	Source file name. Can be NULL.
<code>void *user_data</code>	This field is obsolete.
<code>unsigned int user_data_size</code>	This field is obsolete.
<code>iJDEnvironmentType env</code>	This field is obsolete.

iJIT_Method_Load_V2 Structure

When you use the `iJIT_Method_Load_V2` structure to describe the JIT compiled method, use `iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED_V2` as an event type to report it. The `iJIT_Method_Load_V2` structure has the following fields:

Field	Description
<code>unsigned int method_id</code>	Unique method ID. Method ID cannot be smaller than 999. You must either use the API function <code>iJIT_GetNewMethodID</code> to get a valid and unique method ID, or else manage ID uniqueness and correct range by yourself.
<code>char *method_name</code>	The name of the method, optionally prefixed with its class name and appended with its complete signature. This argument cannot be set to NULL.
<code>void *method_load_address</code>	The base address of the method code. Can be NULL if the method is not JITted.
<code>unsigned int method_size</code>	The virtual address on which the method is inlined. If NULL, then data provided with the event are not accepted.
<code>unsigned int line_number_size</code>	The number of entries in the line number table. 0 if none.
<code>pLineNumberInfo line_number_table</code>	Pointer to the line numbers info array. Can be NULL if <code>line_number_size</code> is 0. See LineNumberInfo structure for a description of a single entry in the line number info array.
<code>char *class_file_name</code>	Class name. Can be NULL.
<code>char *source_file_name</code>	Source file name. Can be NULL.

Field	Description
char *module_name	Module name. Can be NULL. The module name can be useful for distinguishing among different JIT engines. VTune Profiler will display reported methods grouped by specific module.

LineNumberInfo Structure

Use the `LineNumberInfo` structure to describe a single entry in the line number information of a code region. A table of line number entries provides information about how the reported code region is mapped to source file. VTune Profiler uses line number information to attribute the samples (virtual address) to a line number. It is acceptable to report different code addresses for the same source line:

Off	Line Number
-----	-------------

set

1	2
12	4
15	2
18	1
21	30

VTune Profiler constructs the following table using the client data:

Cod	Line Number
-----	-------------

e
sub
-
ran
ge

0-1	2
1-1	4
2	
12-	2
15	
15-	1
18	
18-	30
21	

The `LineNumberInfo` structure has the following fields:

Field	Description
unsigned int Offset	Opcode byte offset from the beginning of the method.
unsigned int LineNumber	Matching source line number offset (from beginning of source file).

Return Values

The return values are dependent on the particular `iJIT_JVM_EVENT`.

See Also

[About JIT Profiling API](#)

[Using JIT Profiling API](#)

iJIT_IsProfilingActive

Returns the current mode of the agent.

Syntax

```
iJIT_IsProfilingActiveFlags JITAPI iJIT_IsProfilingActive ( void )
```

Description

The `iJIT_IsProfilingActive` function returns the current mode of the agent.

Input Parameters

None

Return Values

`iJIT_SAMPLING_ON`, indicating that agent is running, or `iJIT_NOTHING_RUNNING` if no agent is running.

See Also

[About JIT Profiling API](#)

[Using JIT Profiling API](#)

iJIT_GetNewMethodID

Generates a new unique method ID.

Syntax

```
unsigned int iJIT_GetNewMethodID(void);
```

Description

The `iJIT_GetNewMethodID` function generates new method ID upon each call. Use this API to obtain unique and valid method IDs for methods or traces reported to the agent if you do not have your own mechanism to generate unique method IDs.

Input Parameters

None

Return Values

A new unique method ID. When out of unique method IDs, this API function returns 0.

See Also

[About JIT Profiling API](#)

[Using JIT Profiling API](#)

System APIs Supported by Intel® VTune™ Profiler

VTune Profiler supports interpretation of Linux* and Microsoft* Windows* OS APIs.

The following table lists all of the 32-bit and 64-bit OS threading and synchronization functions that are currently supported by VTune Profiler. Check the Release Notes to see if support for new APIs has been added recently. If an API is not supported, the collected statistics will be incomplete.

API for Windows* OS

.NET* APIs	
RegisterClassA	ThreadPool_UnsafeRegisterWaitForSingleObject_4
RegisterClassW	ThreadPool_QueueUserWorkItem_1
RegisterClassExA	ThreadPool_QueueUserWorkItem_2
RegisterClassExW	ThreadPool_UnsafeQueueUserWorkItem
UnregisterClassA	ThreadPool_UnsafeQueueNativeOverlapped
UnregisterClassW	Timer_Ctor_1
GetClassInfoA	Timer_Ctor_2
GetClassInfoW	Timer_Ctor_3
GetClassInfoExA	Timer_Ctor_4
GetClassInfoExW	Timer_Ctor_5
GetWindowLongA	Monitor_Exit
GetWindowLongW	MonitorWait
GetWindowLongPtrA	Monitor_Wait_1
GetWindowLongPtrW	Monitor_Wait_2
GetClassLongA	Monitor_Wait_3
GetClassLongW	Monitor_Wait_4
GetClassLongPtrA	Monitor_Wait_5
GetClassLongPtrW	Monitor_Pulse
SetWindowLongA	Monitor_PulseAll
SetWindowLongW	Monitor_Enter
SetWindowLongPtrA	Monitor_Enter_1
SetWindowLongPtrW	MonitorTryEnter
SetClassLongA	Monitor_TryEnter_1
SetClassLongW	Monitor_TryEnter_2
SetClassLongPtrA	Monitor_TryEnter_3
SetClassLongPtrW	Monitor_TryEnter_4
AutoResetEvent_Ctor	Monitor_TryEnter_5
ManualResetEvent_Ctor	Mutex_Ctor_1
EventWaitHandle_Ctor_1	Mutex_Ctor_2

.NET* APIs

EventWaitHandle_Ctor_2	Mutex_Ctor_3
EventWaitHandle_Ctor_3	Mutex_Ctor_4
EventWaitHandle_Ctor_4	Mutex_Ctor_5
EventWaitHandle_OpenExisting_1	Mutex_Release
EventWaitHandle_OpenExisting_2	Mutex_OpenExisting_1
EventWaitHandle_Set	Mutex_OpenExisting_2
EventWaitHandle_Reset	Semaphore_Ctor_1
WaitHandle_WaitOne_1	Semaphore_Ctor_2
WaitHandle_WaitOne_2	Semaphore_Ctor_3
WaitHandle_WaitOne_3	Semaphore_Ctor_4
WaitHandle_WaitAny_1	Semaphore_OpenExisting_1
WaitHandle_WaitAny_2	Semaphore_OpenExisting_2
WaitHandle_WaitAny_3	Semaphore_Release_1
WaitHandle_WaitAll_1	Semaphore_Release_2
WaitHandle_WaitAll_2	ReaderWriterLock_Ctor
WaitHandle_WaitAll_3	ReaderWriterLock_AcquireReaderLock_1
WaitHandle_SignalAndWait_1	ReaderWriterLock_AcquireReaderLock_2
WaitHandle_SignalAndWait_2	ReaderWriterLock_AcquireWriterLock_1
WaitHandle_SignalAndWait_3	ReaderWriterLock_AcquireWriterLock_2
Thread_Join_1	ReaderWriterLock_ReleaseReaderLock
Thread_Join_2	ReaderWriterLock_ReleaseWriterLock
Thread_Join_3	ReaderWriterLock_UpgradeToWriterLock_1
Thread_Sleep_1	ReaderWriterLock_UpgradeToWriterLock_2
Thread_Sleep_2	ReaderWriterLock_DowngradeFromWriterLock
Thread_Interrupt	ReaderWriterLock_RestoreLock
ThreadPool_RegisterWaitForSingleObject_1	ReaderWriterLock_ReleaseLock
ThreadPool_RegisterWaitForSingleObject_2	WaitHandle_WaitOne_4
ThreadPool_RegisterWaitForSingleObject_3	WaitHandle_WaitOne_5
ThreadPool_RegisterWaitForSingleObject_4	WaitHandle_WaitAny_4
ThreadPool_UnsafeRegisterWaitForSingleObject_1	WaitHandle_WaitAny_5
ThreadPool_UnsafeRegisterWaitForSingleObject_2	WaitHandle_WaitAll_4
ThreadPool_UnsafeRegisterWaitForSingleObject_3	WaitHandle_WaitAll_5

Callback APIs

BindIoCompletionCallback	QueueUserAPC
GetOverlappedResult	RaiseException

Condition variable APIs	
RtlInitializeConditionVariable	SleepConditionVariableCS
RtlWakeAllConditionVariable	SleepConditionVariableSRW
RtlWakeConditionVariable	
Critical section APIs	
InitializeCriticalSection	RtlInitializeCriticalSection
InitializeCriticalSection	RtlTryEnterCriticalSection
InitializeCriticalSectionEx	RtlEnterCriticalSection
InitializeCriticalSectionAndSpinCount	RtlLeaveCriticalSection
RtlInitializeCriticalSectionAndSpinCount	RtlSetCriticalSectionSpinCount
	RtlDeleteCriticalSection
Event APIs	
CreateEventA	OpenEventW
CreateEventExA	PulseEvent
CreateEventExW	ResetEvent
CreateEventW	SetEvent
OpenEventA	PulseEvent
Fiber APIs	
SwitchToFiber	DeleteFiber
CreateFiberEx	FiberStartRoutineWrapper
File/Directory APIs	
CreateFileA	FindFirstFileW
CreateFileW	FindFirstFileExA
OpenFile	FindFirstFileExW
WriteFile	FindNextChangeNotification
WriteFileEx	FindNextFileA
WriteFileGather	FindNextFileW
ReadFile	GetCurrentDirectoryA
ReadFileEx	GetCurrentDirectoryW
ReadFileScatter	MoveFileA
FindFirstChangeNotificationA	MoveFileW
FindFirstChangeNotificationW	MoveFileExA
FindCloseChangeNotification	MoveFileExW
CreateDirectoryA	ReadDirectoryChangesW

File/Directory APIs	
CreateDirectoryW	RemoveDirectoryA
CreateDirectoryExA	RemoveDirectoryW
CreateDirectoryExW	SetCurrentDirectoryA
DeleteFileA	SetCurrentDirectoryW
DeleteFileW	lock
FindFirstFileA	unlock

Input/output APIs	
CreateMailslotA	ReadConsoleInputA
CreateMailslotW	ReadConsoleInputW
DeviceIoControl	ReadConsoleA
FindFirstPrinterChangeNotification	ReadConsoleW
FindClosePrinterChangeNotification	WaitCommEvent
GetStdHandle	WaitForInputIdle

Memory Allocation APIs	
malloc	LocalReAlloc
calloc	LocalSize
realloc	LocalUnlock
free	GetProcessHeap
RtlAllocateHeap	GetProcessHeaps
RtlReAllocateHeap	HeapAlloc
RtlFreeHeap	HeapCompact
RtlSizeHeap	HeapCreate
GlobalAlloc	HeapDestroy
GlobalFlags	HeapFree
GlobalFree	HeapLock
GlobalHandle	HeapQueryInformation
GlobalLock	HeapReAlloc
GlobalReAlloc	HeapSetInformation
GlobalSize	HeapSize
GlobalUnlock	HeapUnlock
LocalAlloc	HeapValidate
LocalFlags	HeapWalk
LocalFree	
LocalHandle	
LocalLock	

Mutex APIs	
CreateMutexA	OpenMutexA
CreateMutexExA	OpenMutexW
CreateMutexExW	ReleaseMutex
CreateMutexW	

Networking APIs	
RpcNsBindingLookupBeginA	closesocket
RpcNsBindingLookupBeginW	connect
RpcNsBindingLookupNext	recv
RpcNsBindingLookupDone	recvfrom
RpcNsBindingImportBeginA	send
RpcNsBindingImportBeginW	sendto
RpcNsBindingImportNext	select
RpcNsBindingImportDone	WSASocketA
RpcStringBindingComposeA	WSASocketW
RpcStringBindingComposeW	WSAAccept
RpcServerListen	WSACreateEvent
RpcMgmtWaitServerListen	WSACloseEvent
RpcMgmtInqIfIds	WSACloseHandle
RpcEpResolveBinding	WSACloseThread
RpcCancelThread	WSACloseWaitable
RpcMgmtEpEltInqBegin	WSACloseWaitableHandle
RpcMgmtEpEltInqDone	WSACloseWaitableThread
RpcMgmtEpEltInqNextA	WSACloseWaitableThreadHandle
RpcMgmtEpEltInqNextW	WSACloseWaitableThreadHandleW
socket	WSACloseWaitableThreadHandleW
accept	WSACloseWaitableThreadHandleW

Object APIs	
CloseHandle	DuplicateHandle

One-time initialization APIs	
InitOnceBeginInitialize	InitOnceExecuteOnce
InitOnceComplete	RtlRunOnceInitialize

Pipe APIs	
CallNamedPipeA	TransactNamedPipe

Pipe APIs	
CallNamedPipeW	WaitNamedPipeA
ConnectNamedPipe	WaitNamedPipeW
CreateNamedPipeA	
CreateNamedPipeW	
Process APIs	
CreateProcessA	TerminateProcess
CreateProcessW	ExitProcess
OpenProcess	RtlExitUserProcess
Semaphore APIs	
CreateSemaphoreA	OpenSemaphoreA
CreateSemaphoreExA	OpenSemaphoreW
CreateSemaphoreExW	ReleaseSemaphore
CreateSemaphoreW	
Sleep APIs	
Sleep	SleepEx
Slim Reader/Writer (SRW) Locks APIs	
RtlInitializeSRWLock	RtlAcquireSRWLockShared
RtlAcquireSRWLockExclusive	RtlReleaseSRWLockShared
RtlReleaseSRWLockExclus	
Thread APIs	
CreateThread	RtlExitUserThread
CreateRemoteThread	TerminateThread
OpenThread	SuspendThread
ExitThread	Wow64SuspendThread
FreeLibraryAndExitThread	ResumeThread
Threadpool APIs	
CreateIoCompletionPort	CreateTimerQueue
GetQueuedCompletionStatus	CreateTimerQueueTimer
PostQueuedCompletionStatus	DeleteTimerQueueTimer
CreateThreadpoolWait	DeleteTimerQueueEx
CreateThreadpoolWork	DeleteTimerQueue
TrySubmitThreadpoolCallback	UnregisterWait

Threadpool APIs	
CreateThreadpoolTimer	UnregisterWaitEx
CreateThreadpoolIo	QueueUserWorkItem
CreateThreadpoolCleanupGroup	RegisterWaitForSingleObject
Timer APIs	
CancelWaitableTimer	OpenWaitableTimerA
CreateWaitableTimerA	OpenWaitableTimerW
CreateWaitableTimerW	SetWaitableTimer
Wait APIs	
MsgWaitForMultipleObjects	WaitForMultipleObjectsEx
MsgWaitForMultipleObjectsEx	WaitForSingleObject
SignalObjectAndWait	WaitForSingleObjectEx
WaitForMultipleObjects	RegisteredWaitHandle_Unregister
Windows Messaging APIs	
GetMessageA	PostThreadMessageW
GetMessageW	ReplyMessage
PeekMessageA	WaitMessage
PeekMessageW	DialogBoxParamA
SendMessageA	DialogBoxParamW
SendMessageW	DialogBoxIndirectParamA
SendMessageTimeoutA	DialogBoxIndirectParamW
SendMessageTimeoutW	MessageBoxA
SendMessageCallbackA	MessageBoxW
SendMessageCallbackW	MessageBoxExA
SendNotifyMessageA	MessageBoxExW
SendNotifyMessageW	NdrSendReceive
BroadcastSystemMessageExA	NdrNsSendReceive
BroadcastSystemMessageExW	PrintDlgA
BroadcastSystemMessageA	PrintDlgW
BroadcastSystemMessageW	PrintDlgExA
PostMessageA	PrintDlgExW
PostMessageW	ConnectToPrinterDlg
PostThreadMessageA	

API for Linux* OS

Timer, signal and wait APIs	
setitimer	clock_nanosleep
getitimer	pause
wait	alarm
waitpid	signal
waitid	sigaction
wait3	sigprocmask
wait4	sigsuspend
sleep	sigpending
usleep	sigtimedwait
ualarm	sigwaitinfo
nanosleep	sigwait

I/O API	
getwc	read
getw	write
getchar	readv
getwchar	writev
getch	open
wgetch	fopen
mvgetch	fdopen
gets	close
fgetc	fclose
fgetwc	io_submit
fgets	io_cancel
fgetws	io_setup
fread	io_destroy
fwrite	io_getevents
pipe	

Synchronous I/O multiplexing APIs	
select	epoll_pwait
pselect	poll
epoll_wait	ppoll

Network API	
socket	recv

Network API	
accept	recvfrom
connect	send
shutdown	sendto
File Locking API	
ioctl	funlockfile
flock	lockf
flockfile	fcntl
DSO API	
dlopen	dlvsym
dlclose	dladdr
dlsym	dladdr1
RPC API	
callrpc	pmap_rmtcall
clnt_broadcast	pmap_set
clntudp_create	svc_run
clntudp_bufcreate	svc_sendreply
clntraw_create	svccraw_create
pmap_getmaps	svctcp_create
pmap_getport	svcupd_bufcreate
	svcupd_create
POSIX Thread Function API	
pthread_exit	pthread_rwlock_timedrdlock
pthread_cancel	pthread_rwlock_timedwrlock
pthread_barrier_init	pthread_spin_init
pthread_barrier_destroy	pthread_spin_destroy
pthread_barrier_wait	pthread_spin_lock
pthread_mutex_init	pthread_spin_unlock
pthread_mutex_destroy	pthread_cond_init
pthread_mutex_lock	pthread_cond_destroy
pthread_mutex_unlock	pthread_cond_broadcast
pthread_mutex_timedlock	pthread_cond_signal
pthread_rwlock_init	pthread_cond_timedwait
pthread_rwlock_destroy	pthread_cond_wait

POSIX Thread Function API

pthread_rwlock_rdlock	pthread_key_create
pthread_rwlock_wrlock	pthread_key_delete
pthread_rwlock_unlock	pthread_sigmask
pthread_create	pthread_setcancelstate
pthread_join	

POSIX Interprocess Communication API

sem_init	recvmsg
sem_destroy	sendmsg
sem_wait	msgrcv
sem_timedwait	msgsnd
sem_post	msgget
semop	semget
semtimedop	

POSIX Message Queue API

mq_close	mq_timedreceive
mq_open	mq_send
mq_receive	mq_timedsend

See Also[API Support](#)

Troubleshooting

Use these topics to learn about best practices, common problems, and their solutions when you run performance analyses with Intel® VTune™ Profiler:

Best Practices

- Best Practice: Resolve VTune Profiler BSODs, Crashes, and Hangs in Windows OS

Error Messages

- Error Message: Application Sets Its Own Handler for Signal
- Error Message: Cannot Enable Event-Based Sampling Collection
- Error Message: Cannot Collect GPU Hardware Metrics
- Error Message: Cannot Load Data File
- Error Message: Cannot Locate Debugging Symbols
- Error Message: Result Is Empty
- Error Message: Client Is Not Authorized To Connect to Server
- Error Message: Make sure you have root privileges to analyze Processor Graphics hardware events
- Error Message: No Pre-built Driver Exists for This System
- Error Message: Not All OpenCL Code Profiling Callbacks Are Received
- Error Message: Problem Accessing the Sampling Driver

- Error Message: Required Key Not Available
- Error Message: Scope of ptrace System Call Application Is Limited
- Error Message: Stack Size Is Too Small
- Error Message: Symbol File Is Not Found

Problems

- Problem: Analysis of the .NET* Application Fails
- Problem: Cannot Access Documentation
- Problem: CPU Time for Hotspots and Threading Analysis Is Too Low
- Problem: Events= Sample After Value (SAV) * Samples Is Wrong for Disabled Multiple Runs
- Problem: Guessed Stack Frames
- Problem: GUI Hangs or Crashes
- Problem: Inaccurate Sum in the Grid
- Problem: Information Collected via ITT API Is Not Available When Attaching to a Process
- Problem: No GPU Utilization Data Is Collected
- Problem: Same Functions Are Compared As Different Instances
- Problem: Skipped Stack Frames
- Problem: Stack in the Top-Down Tree Window Is Incorrect
- Problem: Stacks in Call Stack and Bottom-Up Panes Are Different
- Problem: System Functions Appear in the User Functions Only Mode
- Problem: VTune Profiler is Slow to Respond When Collecting or Displaying Data
- Problem: VTune Profiler is Slow on XServers with SSH Connection
- Problem: Unexpected Paused Time
- Problem: {Unknown Timer} in the Platform Power Analysis Viewpoint
- Problem: Unknown Critical Error Due to Disabled Loopback Interface
- Problem: Unknown Frames
- Problem: Unreadable text in Intel VTune Profiler on macOS*
- Problem: Unsupported Windows Operating System

Warnings

- Warnings about Accurate CPU Time Collection

Best Practices: Resolve Intel® VTune™ Profiler BSODs, Crashes, and Hangs in Windows* OS

Scenario

When you use Intel® VTune™ Profiler to profile target applications on Windows systems, if you experience problems with an unresponsive UI or tool crashes, the following suggestions can help you get better clarity on the root causes. Verify if one of these scenarios apply to your environment. If you need further assistance, [contact us to get help](#).

BSOD from Incompatible Intel® VTune™ Profiler Driver or Windows Update

What is happening?

Intel® VTune™ Profiler runs well on a Windows system. After an update to the OS, Intel® VTune™ Profiler crashes for certain analysis types, while others are unavailable.

Why is it happening?

Sometimes, the latest version of Intel® VTune™ Profiler may be one update behind the latest version of Windows OS. The changes contained in the Windows update can then cause an incompatibility with VTune Profiler drivers, particularly with the stack sampling driver for hardware-event based sampling (HEBS) collections. Ideally, you should upgrade to the latest version of Intel® VTune™ Profiler after every time you update Windows OS. This ensures that all relevant drivers are installed.

When you install Intel® VTune™ Profiler on an unsupported version of Windows, the installer does not install incompatible drivers. This disables HEBS and stack collection. However, you may still be able to run hotspots or threading analyses that use user-mode sampling. If you proceed to upgrade Windows to a newer, unsupported version, the user-mode sampling collections attempt to use unavailable drivers and cause Intel® VTune™ Profiler to crash.

Suggestion Every time you upgrade to the latest version of Windows OS, uninstall your existing version of Intel® VTune™ Profiler and install the latest available version.

BSOD from Driver Conflict

What is happening?

A BSOD occurs due to a driver conflict that affects Intel® VTune™ Profiler drivers.

Why is it happening?

Sometimes, there can be a conflict between Intel® VTune™ Profiler drivers and graphics or third-party drivers. This can likely happen if the Intel® VTune™ Profiler drivers are out of date.

Suggestion Update all Intel® VTune™ Profiler drivers by installing the latest available version.

VTune Profiler UI turns Unresponsive or 'Hangs'

What is happening?

During symbol resolution stage, Intel® VTune™ Profiler stalls or hangs without any response.

Why is it happening?

Several reasons can cause Intel® VTune™ Profiler to hang during the collection and finalization phase.

PDB File Retrieval

When symbol resolution happens in the finalization process, Intel® VTune™ Profiler may have to retrieve and process large .pdb files. If used within Microsoft Visual Studio, Intel® VTune™ Profiler uses the Visual Studio settings to find symbol files and any additional paths provided in Intel® VTune™ Profiler settings. However, if Intel® VTune™ Profiler uses a symbol server, the resolution waits on updates and therefore slows down. Depending on the size of the .pdb files, this may cause Intel® VTune™ Profiler to stall or hang.

Suggestion If your analysis requires symbols for system libraries, use a local cache (like the location defined in Visual Studio) instead of a symbol server. Also, remove large .pdb files from the symbol location you provide to Intel® VTune™ Profiler if these files are not required for your analysis.

Synchronization with other Processes

Certain processes like virus scanners or synchronization/back-up utilities can interfere with data collection and finalization in Intel® VTune™ Profiler. Virus scanners can cause problems in the process that Intel® VTune™ Profiler uses for software-based analysis types, such as threading. Some synchronization utilities can cause finalization to fail if the backup happens as Intel® VTune™ Profiler is processing it.

Suggestion Exclude the pin.exe process from your virus scanning software or disable the scan when running a Intel® VTune™ Profiler collection. Also, pause synchronization and/or back-up utilities until Intel® VTune™ Profiler finalization is complete.

Intel® VTune™ Profiler Crashes during a Collection

What is happening?

Intel® VTune™ Profiler crashes in the middle of a collection operation.

Why is it happening?

A crash can happen if Intel® VTune™ Profiler attempts to instrument or attach to a privileged process or service.

Suggestion Run Intel® VTune™ Profiler as an administrator. You can then profile processes with elevated privileges. You can also configure Intel® VTune™ Profiler to profile specific modules. See the **Advanced** section in the WHAT pane for this purpose.

Other Techniques to Enable Data Collection

What is happening?

Intel® VTune™ Profiler does not perform data collection in some specific situations.

Why is it happening?

Certain specific actions can cause data collection to fail. See if one of these suggestions helps to resolve your issue.

Problem	Suggestion
User-mode sampling for Threading analysis is too slow or creates too much overhead.	Run Threading analysis with Hardware-Event Based Sampling (HEBS) and context switches enabled. This provides the context switch data necessary to understand thread behavior.
Hotspots analysis is unavailable with HEBS and stack collection enabled.	Disable stack collection. To correlate hotspots with stacks, run a separate hotspots analysis with user-mode sampling enabled.
Intel® VTune™ Profiler hangs or crashes when attaching to a running process.	Run Intel® VTune™ Profiler with the application in paused state. Resume data collection when the application gets to an area of interest.
Data collection crashes when using the Instrumentation and Tracing Technology (ITT) API.	Create a custom analysis . Disable the checkbox to analyze user tasks, events, and counters. Identify if the API is causing the crash.

Get Help

The suggestions described in this topic can help resolve several crashes or stalls. If you are still facing issues, contact us so we may better assist you.

- Contact [Customer Support](#).
- Discuss in the [Analyzers developer forum](#).
- See if the issue has been addressed in the [Intel® VTune™ Profiler release notes](#).

See Also

[Hardware Event-based Sampling Collection](#)

Error Message: Application Sets Its Own Handler for Signal

Full error message: *Application sets its own handler for signal <conflicting_signal> that is used for internal needs of the tool. Collection cannot continue.* This is a Linux* only message.

Cause

User-mode sampling and tracing collector cannot profile applications that set up the signal handler for a signal used by the Intel® VTune™ Profiler.

Solution

When collecting data with `vtune`, add the `--run-pass-thru=--profiling-signal=<not_used_signal>` command line option, where `<not_used_signal>` is a signal that should not be used by your application to analyze; you need to select the signal from `SIGRTMIN..SIGRTMAX`.

Alternatively, you may set the environment variable `AMPLXE_RUNTOOL_OPTIONS=--profiling-signal=<not_used_signal>`. You may do this, either from your terminal window before running the VTune Profiler GUI or from the **Configure Analysis** window entering the variable into the **User-defined Environment Variables** field.

See Also

[Set Up Analysis Target](#)

Error Message: Cannot Enable Event-Based Sampling Collection

Cause

Intel® VTune™ Profiler cannot access the PMU resources in the virtualization environment since either the PMU resources are made unavailable through BIOS options or Hyper-V has been activated on an unsupported platform.

Known System Limitations

- The sampling-based performance profiling on Hyper-V has only been available since Windows 10 RS3 release (version 1709) or later. Check your Windows OS version to make sure the VTune Profiler can run on the system:

```
> winver
```

For example, Version 1709 indicates that the supported Windows 10 Fall Creators Update (RedStone3) is running on the system:



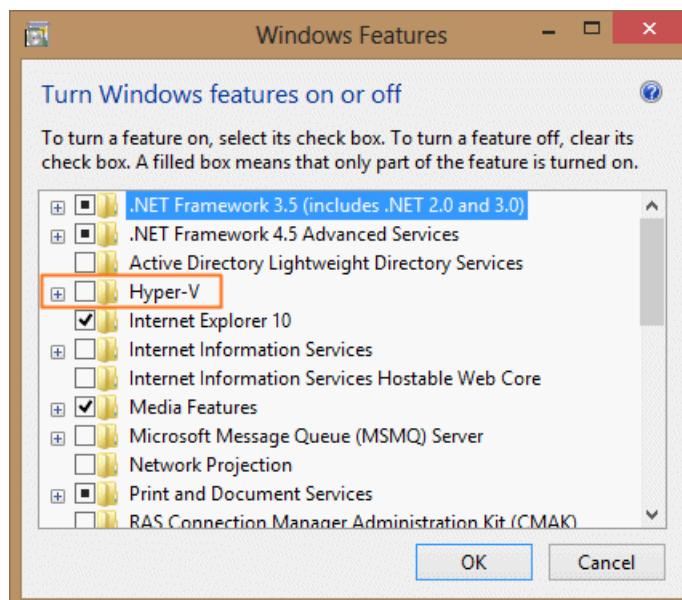
- The Hyper-V allows the sampling-based performance profiling on the latest generation of Intel microarchitectures code named Skylake and Goldmont onward. VTune Profiler will not be able to work in the Hyper-V environment running on Intel microarchitectures code named Haswell or Broadwell.

Solution

To enable hardware event-based sampling collection for systems *prior to* Windows 10 RS3, do the following:

- Enable access to the PMU resources through BIOS options (if it was disabled manually).
- Disable the **Hyper-V** feature as follows:

- From the **Start** menu select **Search > Settings > Turn Windows features on or off** to open the **Windows Features** window.
- Make sure to disable the **Hyper-V** feature and its sub-features and restart the system.



- 3.** If the **Hyper-V** feature is not disabled even after the system reboot, you must disable the BIOS VMX (virtualization feature) if it was not turned off already.

To troubleshoot hardware event-based sampling collection problems for Windows 10 RS3, make sure you have the [Credential Guard and Device Guard security features disabled](#) on your system.

See Also

[Profiling Targets in the Hyper-V* Environment](#)

Error Message: Cannot Collect GPU Hardware Metrics

Possible error messages:

- *Cannot collect GPU hardware metrics because neither libigdmd.so nor libmd.so was found.*
- *Cannot collect GPU hardware metrics because neither libigdmd.so nor libmd.so can be initialized.* Make sure you have installed Metrics Discovery API from <https://github.com/intel/metrics-discovery> correctly.
- *Cannot collect GPU hardware metrics because libmd.so cannot be loaded.* Make sure you have installed Metrics Discovery API from <https://github.com/intel/metrics-discovery> correctly.
- *Cannot collect GPU hardware metrics because libmd.so was not found.* Make sure you have installed Metrics Discovery Application Programming Interface from <https://github.com/intel/metrics-discovery>.
- *Cannot collect GPU hardware metrics because your version of the Metrics Discovery API is obsolete.*

Cause

To collect GPU hardware metrics and GPU utilization data on Linux, VTune Profiler uses the Intel® Metric Discovery API library distributed with the product. If VTune cannot access the library, one of the aforementioned error messages display.

Solution

Depending on the version of VTune Profiler you use, choose one of these solutions and follow the steps.

Version of VTune Profiler	Step 1	Step 2
Upgrade to the latest version of VTune Profiler	<p>Install as part of the Intel® oneAPI Base Toolkit</p> <p>Install as a standalone component</p>	<p>No further actions necessary. Product versions starting with 2021.1 automatically select the latest libstdc++ available in runtime to satisfy the GPU analysis requirements, so no additional configuration is required.</p> <p>Install the Intel Metric Discovery API library ver. 1.12.148 (or newer) from the official repository at https://github.com/intel/metrics-discovery.</p>
Use VTune Profiler versions 2020, 2021.1.0 beta04 or an older version	<p>Install the Intel Metric Discovery API library ver. 1.12.147 (or older) from the official repository at https://github.com/intel/metrics-discovery.</p>	<p>Ensure that the API library meets the following requirements:</p> <ul style="list-style-type: none"> • To enable VTune Profiler to successfully load the library, it should be linked to libstdc++ (version GLIBCXX_3.4.20 or

Version of VTune Profiler	Step 1	Step 2
		<p>older) or statically linked to libstd++. If libmd.so (renamed to libigdmd.so starting with MD API 1.12.148) is dynamically linked to a newer version of libstdc++, make sure to have it loaded to the process before loading libmd.so. You can do this, for example, by re-defining the environment variable LD_PRELOAD:</p> <pre>LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6 vtune -c gpu-hotspots.</pre> <ul style="list-style-type: none"> If you use su or sudo command to run the VTune Profiler, you need to redefine LD_PRELOAD directly in the command, for example: <pre>sudo LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6 vtune -c gpu-hotspots</pre> In case of remote target profiling, remove or rename the following file in the VTune Profiler package installed on the remote target: <pre><vtune-target-install-dir>/lib64/libstdc+.so.6</pre>

See Also

[Set Up System for GPU Analysis](#)

[GPU Compute/Media Hotspots Analysis \(Preview\)](#)

Error Message: Cannot Load Data File

Cause

The collected temporary data may have exceeded the current allocated or available global temporary storage space on a Linux* target system.

Solution

Consider providing an [alternative temporary directory](#) for collected data.

See Also

[Analysis Target Setup](#)

Error Message: Cannot Locate Debugging Information

Cause

Debugging information (PDB files on Windows* and DWARF format on Linux*) for applications and system modules is not generally available on the system by default. Missing debug information is not critical to performance analysis but prevents Intel® VTune™ Profiler from providing full-scale statistics on call stacks, source data, and so on.

If the VTune Profiler does not find debug information for the binaries, it statically identifies function boundaries and assigns hotspot addresses to generated pseudo names `func@address` for such functions, for example:

If a module is not found or the name of a function cannot be resolved, the VTune Profiler displays module identifiers within square brackets, for example: `[module]`.

If the debug information is absent, the VTune Profiler may not unwind the call stack and display it correctly in the **Call Stack** pane. Additionally in some cases, it can take significantly more time to [finalize](#) the results for modules that do not have debug information.

Solution

For accurate performance analysis, you are recommended to have the debug information available on the system where the VTune Profiler is installed. See detailed instructions to enable:

- [debug information for Windows application binaries](#)
- [debug information for Windows system libraries](#)
- [debug information for Linux application binaries](#)
- [debug information for Linux kernels](#)

See Also

[Compiler Switches for Performance Analysis on Windows* Targets](#)

[Compiler Switches for Performance Analysis on Linux* Targets](#)

Error Message: Cannot Open Data

Error message: *Cannot open data. Intel® VTune™ Profiler has faced a serious problem.*

Cause

The data collection period could be too short (for example, <10ms), so that the VTune Profiler could not capture performance data.

Solution

Consider the following options:

- Verify that you can run your application without the VTune Profiler.

You may have two console windows: the first one for building the application and the second one for launching the VTune Profiler. The second console should run the application smoothly before attempting to launch the VTune Profiler. If you see an error message reporting problems with loading shared libraries on the second console, set up the environment correctly either via the `LD_LIBRARY_PATH` variable or by running `source <install-dir>/env/vars.sh` for Linux* and `vars.bat` for Windows*. Once the application runs, start the VTune Profiler from that environment.

- If the analysis duration is too short, increase the workload for your application.

See Also

[Manage Result Files](#)

Error Message: Client Is Not Authorized to Connect to Server

Cause

If you are running a VNC* session as a standard user, but trying to launch a graphical application for GPU analysis as a root user, you may get a system error message: '*Client is not authorized to connect to Server*' because, by default, and for security reasons, root cannot connect to a non-root user's X Server.

Solution

You may [permanently allow root access](#) applying any of the two proposed methods.

See Also

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

Error Message: Root Privileges Required for Processor Graphics Events

Full error message: *Make sure you have root privileges to analyze Processor Graphics hardware events.*

Cause

You selected the **Analyze Processor Graphics events** option of the GPU analysis but do not have a supported version of the Intel® Metric Discovery API library installed.

Solution

To analyze Intel® HD Graphics and Intel® Iris® Graphics hardware events, make sure to [set up your system for GPU analysis](#)

See Also

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

Error Message: No Pre-built Driver Exists for This System

When executing the `build-driver` script on Linux*, you may see a warning message similar to the following if the kernel sources are not configured properly (they do not match the kernel that is running): `Warning: Current running kernel is version 2.4.18-e.31smp. After successfully building the driver and running the insmod-sep3 or insmod-sep command, the following message appears: No pre-built driver exists for this system.`

Solution

To resolve this issue, execute the following commands to configure the kernel sources:

```
$ cd /usr/src/linux  
$ make mrproper  
$ cp /boot/config-'uname-r' .config  
$ vi Makefile
```

Make sure that EXTRAVERSION matches the tail of the output of `uname -r`. The resulting `/user/src/linux/include/version.h` should have a `UTS_RELEASE` that matches the output of `uname -r`. Once that is true, run the following commands:

```
$ make oldconfig  
$ make dep
```

After completing these steps, run the `build-driver` script to build the sampling driver against the kernel sources in `/usr/src/linux`

See Also

[Cookbook: Profiling Hardware without Sampling Drivers](#)

[Build and Install the Sampling Drivers for Linux* Targets](#)

Error Message: Not All OpenCL™ API Profiling Callbacks Are Received

Cause

Intel® VTune™ Profiler uses OpenCL™ API to collect profiling information about OpenCL kernels. According to the [OpenCL Specification](#), completion callbacks must be thread-safe and can be called in different threads. It is possible that the completion callback is received while the collection is being stopped.

Solution

Use OpenCL API to set callbacks for events for `clEnqueue*` functions and wait for them to be received. For example:

```
#include <atomic>  
#include <thread>  
...  
#include <CL/cl2.hpp>  
  
std::atomic_uint32_t number_of_uncompleted_callbacks = 0;  
  
void CL_CALLBACK completion_callback(cl_event, cl_int , void*)  
{  
    --number_of_uncompleted_callbacks;  
}  
int main()  
{  
    ...  
    cl::Program prog(context,  
  
std::string((std::istreambuf_iterator<char>(programSourceFile)), std::istreambuf_iterator<char>());  
    ...
```

```
auto kernelFunc = cl::KernelFunctor<cl::Buffer, cl_int>(prog, "sin_cos");
cl::Event event = kernelFunc(cl::EnqueueArgs(cl::NDRange(dataBuf.size()), clDataBuf, 0);

++ number_of_uncompleted_callbacks;
event.setCallback(CL_COMPLETE, completion_callback);
...
while (number_of_uncompleted_callbacks.load())
{
    std::this_thread::yield();
}
return EXIT_SUCCESS;
}
```

See Also

[GPU OpenCL™ Application Analysis](#)

Error Message: Problem Accessing the Sampling Driver

Linux* only error message: *Problem accessing the sampling driver. The driver may need to be (re)started.*

Cause

Intel® VTune™ Profiler cannot access the [hardware event-based sampling](#) (EBS) driver required to run a hardware event-based sampling analysis type. This problem happens if the sampling driver was not loaded or you do not have correct permissions.

Solution

Make sure the sampling drivers are loaded:

```
> lsmod | grep sep3_1 or > lsmod | grep sep4_
> lsmod | grep pax
```

If the drivers are already loaded, make sure you are a member of the `vtune` user group. You can check the `/etc/group` file or contact your system administrator to find out if you are a member of this group.

See Also

[Sampling Drivers](#)

[Cookbook: Profiling Hardware without Sampling Drivers](#)

Error Message: Required Key Not Available

Cause

For hardware event-based sampling analysis and Intel Energy Profiler analysis with VTune Profiler for Systems, an Android* system requires signed drivers. Every time the Android kernel is built, a random private/public key is generated. [Drivers must be signed](#) with the random private key to be loaded. The drivers must be signed with the same key and be compiled against the same kernel headers/sources as what is installed on the Android target system.

Solution

Make sure you use the same signing key that was produced at the time and on the system where your kernel was built for your target.

See Also

[Android* System Setup](#)

Error Message: Scope of `ptrace` System Call Is Limited

Full error message: *Failed to start profiling because the scope of `ptrace()` system call application is limited. To enable profiling, please set `/proc/sys/kernel/yama/ptrace_scope` to 0. See the Release Notes for instructions on enabling it permanently.* This is a Linux* only message.

Cause

VTune Profiler may fail to collect data for Hotspots and Threading analysis types on the Ubuntu* operating system if the scope of `ptrace()` system call application is limited.

Solution

Set the value of the `kernel.yama.ptrace_scope` sysctl option to 0 with this command:

```
sysctl -w kernel.yama.ptrace_scope=0
```

To make this change permanent, set the `kernel.yama.ptrace_scope` value to 0 in the `/etc/sysctl.d/10-ptrace.conf` file using root permissions and reboot the machine.

Error Message: Stack Size Is Too Small

Full error message: *Stack size provided to `sigaltstack` is too small. Please increase the stack size to 64K minimum.* This message is Linux* only.

Cause

When setting up SIGPROF signal handler, the VTune Profiler attempts to configure the signals to use the alternative stack size using `sigaltstack()` API to make sure that its signal handler does not depend on the stack size of the profiled application. If the application uses alternative signal stack itself, the Intel® VTune™ Profiler requires that the alternative stack size is 64K at a minimum. This may be not the case if the application uses `SIGSTKSZ` constant for the alternative stack size (which is 8192 bytes). In this case, the data collection may terminate with the error message.

Solution

Configure the VTune Profiler not to set up the alternative stack and use the stack provided by the application. To do this, pass the following command line options to the tool:

```
vtune --run-pass-thru=--no-altstack
```

Or, set up the environment variable `AMPLXE_RUNTOOL_OPTIONS=--no-altstack`.

See Also

[Pane: Call Stack](#)

[View Stacks](#)

Error Message: Symbol File Is Not Found

This is a Linux* only message. Intel® VTune™ Profiler may display the error message about missing symbol file during user-mode sampling and tracing collection. For example:

```
/opt/intel/vtune_profiler/bin64/vtune -collect hotspots -r test1 - my_test_exe
vtune: Warning: Symbol file is not found.
vtune: The call stack passing through the module [vdso] may be incorrect.
vtune: Using result path '/home/user/test1'
vtune: Executing actions 75 % Generating a report
-----
Summary
-----
Elapsed Time: 6.354 CPU Time: 6.210
...
vtune: Executing actions 100 % done
```

Cause and Solution

VTune Profiler notifies you that there is a module [vdso] that cannot be resolved for symbols (the square brackets are used for that purpose) and therefore the call stack may be incorrect. In some cases it might be a [vsyscall] module.

You may check that the vdso module is in a dynamic dependency list:

```
ldd -d my_test_exe linux-vdso.so.1
=> (0x00002aaaaac6000) libtbb.so.2
=> /opt/intel/tbb/tbb40_23oss/lib/libtbb.so.2 (0x00002aaaaabc7000) libstdc++.so.6
=> /usr/intel/pkg/gcc/4.5.2/lib64/libstdc++.so.6 (0x00002aaaaadf5000) libm.so.6
=> /lib64/libm.so.6 (0x00002aaaab117000) libgcc_s.so.1
=> /usr/intel/pkg/gcc/4.5.2/lib64/libgcc_s.so.1 (0x00002aaaab26c000) libc.so.6
=> /lib64/libc.so.6 (0x00002aaaab481000) librt.so.1
=> /lib64/librt.so.1 (0x00002aaaab6c2000) libdl.so.2
=> /lib64/libdl.so.2 (0x00002aaaab7cb000) libpthread.so.0
=> /lib64/libpthread.so.0 (0x00002aaaab8cf000) /lib64/ld-linux-x86-64.so.2 (0x00002aaaaaaab000)
```

You can safely ignore this message if you see a reference to the [vdso]. It means that the kernel dynamically made some temporary memory allocations by loading some executable code into memory space. The fact that VTune Profiler throws this message indicates that some Hotspot samples were taken when that code was running. During the post-processing time the VTune Profiler's collector could not find the vdso anymore. The module linux-vdso.so.1 (linux-vsyscall.so.1 or linux-gate.so.1 on earlier Linux kernels) is a Virtual Dynamic Shared Object (VDSO) that resides in the address space of the program. This is a virtual library that contains a complex logic providing user applications with a fast access to system functions, depending on a CPU microarchitecture, either via an interrupt mechanism or via the fast system calls mechanism (for modern CPUs).

See Also

[Debug Information for Linux* Application Binaries](#)

[Window: Cannot Find <file type> File](#)

Problem: Analysis of the .NET* Application Fails

This problem is specific to Windows*.NET applications.

Cause

If your .NET application performs security checks based on a known public key (for example, checks whether its assemblies are strong-name signed), it may either crash when launched by the VTune Profiler or provide unpredicted analysis results.

Solution

This is a third-party technology limitation. To workaround this issue, you are recommended to disable the security check for any of the [user-mode sampling and tracing analysis types](#).

See Also

[.NET* Code Analysis](#)

Problem: Cannot Access VTune Profiler Documentation

Cause

Intel® VTune™ Profiler product help, including context-sensitive help (F1), is available *online* only. Make sure that you have a stable internet connection on your system.

If your browser or operating system has some specific limitations for displaying context help from VTune Profiler, you may see this message:

This browser or operating system do not support Intel VTune Profiler context-sensitive web documentation.

Solution

For the best experience with context help, use the Google Chrome* browser.

You can also access these VTune Profiler documents directly

- [Get Started Guide](#)
- [Installation Guide](#)
- [VTune Profiler User Guide](#)
- [Tutorials](#)
- [VTune Profiler Performance Analysis Cookbook](#)
- [Intel Processor Event Reference](#)

Download offline versions of the VTune documentation from this repository: <https://d1hdbi2t0py8f.cloudfront.net/index.html?prefix=vtune-docs/>.

Get Help

Problem: CPU time for Hotspots or Threading Analysis is Too Low

Cause

The CPU time is low for one of the following reasons:

- The analysis run was very short and the target consumed little CPU time.
- CPU time may be inaccurate for targets that work in short quanta less than the scheduler tick interval. For example, this can happen for frame-by-frame computation in video decoders. To capture CPU time more accurately on Windows* OS, you need to run the analysis with the [accurate CPU time detection mode](#) enabled.

Solution

Try one of the following:

- Extend the duration of the analysis run.
- Windows OS only: Enable accurate CPU time detection. To do this for the Hotspots or Threading analysis, it is enough to run the VTune Profiler with administrative permissions. You may also enable this option explicitly in the custom analysis configuration by checking the **Collect highly accurate CPU time** box. Make sure to extend maximum size of raw collector data.

NOTE

Accurate CPU time collection produces a significant amount of temporary data depending on the system configuration and the profiled target. VTune Profiler may generate up to 5 Mb of temporary data per minute per logical CPU.

See Also

[Warnings about Accurate CPU Time Collection](#)

[Custom Analysis Options](#)

[Add Administrative Privileges](#)

Problem: 'Events= Sample After Value (SAV) * Samples' Is Not True If Multiple Runs Are Disabled

Cause

The **Allow multiple runs** option is located in the **Advanced** section of the **WHAT** pane on the **Configure Analysis** window. By default, the option is disabled. As a result, the VTune Profiler uses event multiplexing and runs the data collection only once. This approach lowers the precision of the collected data. So, the VTune Profiler calculates the number of collected events using the formula: **Events= Sample After Value (SAV) * Samples**. But when the event multiplexing is enabled, this formula is modified as follows: **Events= Sample After Value * Samples * Event Group Count**, where the Event Group Count is the number of incompatible groups of events used during the collection. The Event Group Count multiplier is introduced based on the heuristics approach to fill the gaps when the VTune Profiler collected events for one event group only.

For example, you have three groups of events: A, B and C, where event A1 is in group A. During the application run, the VTune Profiler spends equal time on collecting each group of events. While group A is analyzed, event A1 has 10 samples with SAV of 10.000 and your application generates no A1 samples at all for the rest of the time. In this case, the accurate result is 100.000 events ($10.000 * 10$). But the VTune Profiler provides the resultant number of events as $100.000 * 3$ (A, B and C groups) = 300.000.

Solution

Select the **Allow multiple runs** option to disable event multiplexing and run a separate data collection for each event group. This mechanism provides more precise data on collected events.

See Also

[Sample After Value](#)

[Allow Multiple Runs or Multiplex Events](#)

Problem: Guessed Stack Frames

VTune Profiler displays [Guessed stack frame(s)] in the grid panes.

Cause

VTune Profiler did not unwind the stack to reduce data collection overhead, but resolved the stack heuristically.

[Guessed stack frame] is considered to be a system function. If the **Call Stack Mode** filter bar option is set to **User/system functions**, the VTune Profiler displays [Guessed stack frame(s)].

Solution

To avoid displaying [Guessed stack frame(s)], set the **Call Stack Mode** filter bar option to **Only user functions**.

See Also

[Manage Data Views](#)

[Call Stack Mode](#)

Problem: GUI Hangs or Crashes

Cause

You may face the Intel® VTune™ Profiler GUI hangs during symbol resolution in the finalization process. This typically results from retrieving or processing large .pdb files. If you run the VTune Profiler from Microsoft* Visual Studio* IDE, it automatically uses the Visual Studio settings to find symbol files and any additional paths provided in the VTune Profiler's [search settings](#). If the VTune Profiler uses a symbol server, the module resolution can be slow if it has to wait for updates. Some .pdb files can be large and take time to resolve.

There are also some processes that can interfere with the VTune Profiler collection and finalization, such as virus scanners and synchronization/back-up utilities. Virus scanners can cause problems in the process the VTune Profiler uses for software-based analysis types, such as **Threading**. Some synchronization utilities can also cause finalization to fail if they try to back up a file while the VTune Profiler is processing it.

Crashes during the collection are rare but may happen in some situations, for example, if the VTune Profiler tries to instrument or attach to a privileged process or service that is not accessible to it.

Solution

To workaround a problem with GUI hangs during finalization, consider the following:

- If symbols for system libraries are necessary for your analysis, use a local cache instead of a symbol server, such as the location defined for Visual Studio.
- Remove large pdb files from the search directories provided to the VTune Profiler if they are not the focus of your analysis.
- Exclude the pin.exe process from your virus scanner, or disable the virus scanner while running the VTune profiler collection.
- Pause synchronization and/or back-up utilities until the finalization is complete.

To prevent a possible crash for the VTune Profiler accessing processes with elevated privileges, run the VTune Profiler as administrator. You can also configure the VTune Profiler to profile specific modules in the **Advanced** section of the **WHAT** pane.

See Also[Finalization](#)[Add Administrative Privileges](#)**Problem: Inaccurate Sum in the Grid**

The sum of the several summands in the grid view may not equal the overall time shown for the parent of the items.

Cause

The values in the data columns are rounded. For items that are sums of several other items, such as a function with several stacks, the rounded sums may differ slightly from the sum of rounded summands.

For example:

Module / Function	Time (exact)	Time (rounded)
foo.dll	0.2468	0.247
foo()	0.1234	0.123
bar()	0.1234	0.123

The rounded values in the grid do not sum up exactly as $(0.123 + 0.123) != 0.247$.

See Also[Manage Data Views](#)**Problem: Information Collected via ITT API Is Not Available When Attaching to a Process****Solution**

If you use [ITT API](#) in your source code to collect statistics data, like Frame Analysis or JIT-profiling, by attaching to a process, make sure to set up the following environment variables before starting your target application:

- `INTEL_LIBITNOTIFY32=<>\bin32\runtime\ittnotify_collector.dll`
- `INTEL_LIBITNOTIFY64=<install-dir>\bin64\runtime\ittnotify_collector.dll`

NOTE

The variables should contain the full path to the library without quotes.

See Also[Analysis Target](#)**Problem: No GPU Utilization Data Is Collected**

Intel® VTune™ Profiler collects detailed GPU utilization data during GPU Offload (preview) or GPU Compute/Media Hotspots analysis.

Cause

Intel® VTune™ Profiler may not collect the detailed [GPU utilization](#) data in the following cases:

- GPU analysis is run without root privileges.
- Intel Graphics driver is not signed properly.
- Linux kernel is configured with the CONFIG_FTRACE option disabled.

Solution

Depending on the root cause, which is typically identified by the VTune Profiler and described in a warning message, consider one of the following workarounds:

- Make sure to properly [set up your system for GPU analysis](#).
- Since detailed GPU utilization analysis relies on the Ftrace* technology (i915 Ftrace events collection), your Linux kernel should be properly configured.
 - If you update the kernel rarely, configure and rebuild only module i915.
 - If you update the kernel often, build the special kernel for GPU analysis.

If your system does not support i915 Ftrace event collection, all the GPU Utilization statistics will be calculated based on the hardware events and attributed to the **Render and GPGPU** engine.

See Also

[Rebuild and Install the Kernel for GPU Analysis](#)

[Rebuild and Install Module i915 for GPU Analysis on CentOS*](#)

[Rebuild and Install Module i915 for GPU Analysis on Ubuntu*](#)

[Linux* and Android* Kernel Analysis](#)

Problem: Same Functions Are Compared As Different Instances

The same functions are compared as different instances in the grid panes.

Cause

You are using the **Function Stack** grouping for the recompiled binary. The **Function Stack** grouping uses function start addresses and is based on function instances.

Solution

Switch to the **Source Function Stack** grouping level to ignore start addresses and display the data by source file objects.

See Also

[Compare Results](#)

[Manage Data Views](#)

Problem: Skipped Stack Frames

Intel® VTune™ Profiler displays [Skipped stack frame(s)] in the grid panes.

Cause

VTune Profiler did not unwind the stack to reduce data collection overhead, and failed to resolve the stack heuristically.

Solution

You may collect deeper stacks by creating a custom event-based sampling analysis and increasing the **Stack size** option value in bytes (`-stack-size` option in CLI), though beware that this also increases the collection overhead.

See Also

[Custom Analysis Options](#)

[Manage Data Views](#)

Problem: Stack in the Top-Down Tree Window Is Incorrect

Cause

The target was built with an optimization level that removed stack information from the binary.

Solution

Decrease the optimization level of your project and rebuild the target. Then profile with the Intel® VTune™ Profiler.

See Also

[Compiler Switches for Performance Analysis on Linux* Targets](#)

[Compiler Switches for Performance Analysis on Windows* Targets](#)

[Debug Information for Windows* Application Binaries](#)

[Debug Information for Linux* Application Binaries](#)

Problem: Stacks in Call Stack and Bottom-Up Panes Are Different

The **Call Stack** pane shows more stacks than the call tree in the **Bottom-up** pane.

Cause

There are several stacks going to the same function, but to different code lines (*call sites*).

The call tree in the **Bottom-up** pane aggregates these stacks in one line but the **Call Stack** pane shows each as a separate stack. For more details, see the [Call Stacks in the Bottom-up Pane](#) and [Call Stack Pane](#) topic.

See Also

[Pane: Call Stack](#)

[Window: Bottom-up](#)

Problem: System Functions Appear in the User Functions Only Mode

The **Call Stack Mode** option on the filter bar is set to **Only user functions** but system functions still appear in the result windows.

Cause

If there is a system function that has no user function calling it, the system function appears and its time is shown in the analysis result windows.

See Also

[Call Stack Mode](#)

[Viewing Stacks](#)

Problem: VTune Profiler is Slow to Respond When Collecting or Displaying Data

This problem is specific to Linux* targets.

Cause

- If your project directory (and consequently, the result files) are located on an NFS-mounted directory and not on a local disk, this significantly impacts performance of the tool in several areas: writing of the results is slower, updating project information is slower, and when loading the results for display you may see delays of several minutes.
- If application binaries are on an NFS-mounted drive but not on a local drive, the VTune Profiler takes longer to parse symbol information and present the results.

Solution

Make sure your project directory (and consequently, the result files) and application binaries are located on a local disk and not an NFS-mounted directory. By default, the projects are stored in \$HOME/intel/vtune/projects. If your home directory is on an NFS-mounted drive to facilitate access from multiple systems, you should ensure that you set the project directory to a local directory at project creation.

See Also

[Set Up Project](#)

Problem: VTune Profiler is Slow on X-Servers with SSH Connection

Intel® VTune™ Profiler GUI may respond slowly when you run a remote Linux* collection using an X11-forwarding/X-server.

Cause

The GUI response may be slow if you use an X-server (for example, Xming*) with SSH on Windows to run the VTune Profiler GUI on a connected Linux machine and the X-server is slow.

Solution

Option 1: Enable Traffic Compression

Compression may help if you are forwarding X sessions on a dial-up or slow network. Turn on the compression with ssh -C or specify `Compression yes` in your configuration file.

SSH obtains configuration data in the following order:

- ssh -C command-line option
- user configuration file (~/.ssh/config)
- system configuration file (/etc/ssh/ssh_config)

NOTE

You can explore all available options with `man ssh_config`.

Option 2: Change Your Encryption Cipher

The default cipher on many systems is triple DES (3DES), which is slower than Blowfish and AES. New versions of OpenSSH default to Blowfish. You can change the cipher to Blowfish with `ssh -c blowfish`.

Change your configuration file with the `Cipher` option depending on whether you are connecting with SSH1 or SSH2:

- for SSH1, use `Cipher blowfish`
- for SSH2, use `Ciphers blowfish-cbc,aes128-cbc,3des-cbc,cast128-cbc,arcfour,aes192-cbc,aes256-cbc`

You may also follow recommendations provided in the documentation to an X-server you are using.

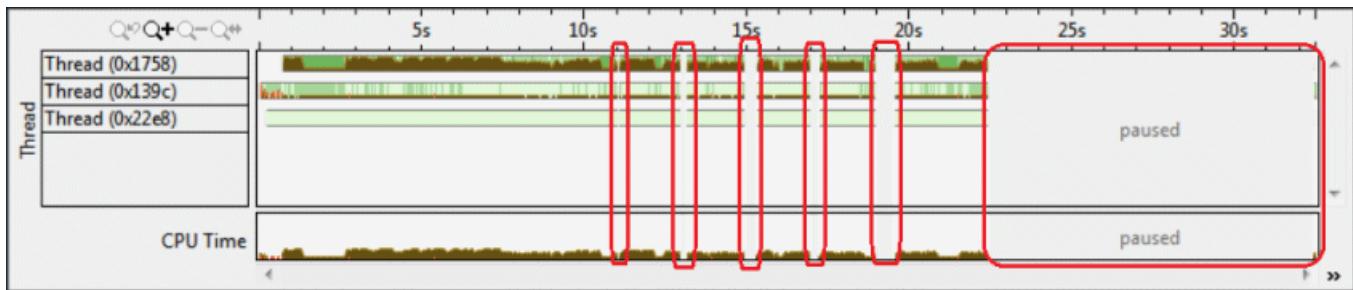
See Also

[Configure SSH Access for Remote Collection](#)

[Set Up Remote Linux* Target](#)

Problem: Unexpected Paused Time

You may see unexpected Paused time in the **Timeline** pane even though you did not add any calls to the `__itt_pause()` API or manually paused the analysis target. For example:



This may happen when collecting call stacks with hardware event-based sampling (EBS).

Cause

In the above example, the application called `__itt_pause()` at about the 22 sec mark. But the other, smaller pauses were inserted by the VTune Profiler, which temporarily pauses profiling when data generation rate exceeds data spill rate and it is about to lose data. The data is flushed and then the collection resumes. In the paused regions, your application is not executing: the VTune Profiler lets the application exhaust its current quanta and then prevents it from being scheduled on the CPU until all the data has been saved to a file.

Solution

You can ignore this injected paused time. For example, in the **Summary** information below, you can see that Paused Time is part of the Elapsed Time, but is not included in CPU Time.

Elapsed Time ⓘ:	32.488s
⌚ CPU Time ⓘ:	12.445s
⌚ Overhead and Spin Time ⓘ:	0.002s
Total Thread Count:	8
Paused Time ⓘ:	20.014s
Frame Count:	2

See Also

[Pausing Data Collection](#)

`start-paused vtune option`

Pane: Timeline

Problem: {Unknown Timer} in the Platform Power Analysis Viewpoint

Platform Power Analysis viewpoint displays an {Unknown Timer} with a blank process name and {Unknown} PID/TID.

Cause

The kernel configuration prevents the VTune Profiler from collecting the required data: it cannot identify the PID/TID/module or process name for the timer.

Solution

You may set the CONFIG_TIMER_STAT =Y in the boot configuration file and recompile the kernel.

See Also

[Interpreting Energy Analysis Data](#)

Problem: Unknown Critical Error Due to Disabled Loopback Interface

Cause

When running a command line Linux* target analysis, the Intel®VTune Profiler may display an error message: "*Fatal error: Unknown critical error*". One of possible reasons could be the disabled loopback interface.

Solution

Run the following command to enable the loopback interface: `ipconfig lo up`.

Problem: Unknown Frames

Cause

When the Intel® VTune™ Profiler finalizes collected data, it uses symbol information to display stack information for each function. If the VTune Profiler cannot find symbols for system modules used in your application, the stack data displayed in the **Bottom-up/Top-down Tree** windows and **Call Stack** pane may be either incomplete or incorrect. The following scenarios are possible:

If	Then
You run Hotspots or/and Threading analysis and your application uses a system API intensively	VTune Profiler cannot unwind the stack correctly since stacks do not reach user code and stay inside the system modules. Often such stacks may be limited to call sites from system modules. Since VTune Profiler tries to attach incomplete stacks to previous full stacks via [Unknown frame(s)], you may see [Unknown frame(s)] hotspots when attributing system layers to user code via the Call stack mode option on the Filter bar.
You run Threading analysis and your application uses synchronization API causing waits that slow down the application	

NOTE

Windows* only: Missing PDB files may lead to the incorrect stack information only for 32-bit applications. For 64-bit applications, stack unwinding information is encoded inside the application.

Solution

1. On Windows, make sure the search directories, specified in the **Binary/Symbol Search** dialog box, include paths to PDB files for your application modules. For more details, see the [Search Directories](#) topic.
2. On Windows, specify paths to the Microsoft* symbol server in **Tools > Options > Debugging > Symbols** page. On Linux, make sure to install the debug info packages available for your system version. For more details, see the [Using Debug Information](#) topic.
3. Re-finalize the result.

On Windows, the VTune Profiler will use the symbol files for system modules from the specified cache directory and provide a more complete call stack.

See Also

[Search Order](#)

[Control Data Collection](#)

Problem: Unreadable Text on macOS*

VTune Profiler displays unreadable text in the graphical user interface on a macOS* host system.

Cause

Running the X11* version of XQuartz* on a macOS system caused the text in the VTune Profiler graphical interface to appear garbled and unreadable. The problem is related to the XQuartz X11 server performing font anti-aliasing, even in 256 color mode.

Solution

Reset the XQuartz preference to "millions" of colors and restart XQuartz.

See Also

#unique_432

Problem: Unsupported Microsoft® Windows® OS

Intel® VTune™ Profiler does not support your current Windows* operating system.

Cause

In general, VTune Profiler is compatible with Windows OS versions supported by Microsoft, but there may be one update behind the latest major version. Depending on the changes in the OS update, this may cause incompatibility with the VTune Profiler drivers, particularly the sampling driver for hardware event-based collections. VTune Profiler installer detects an unsupported OS and fails to install incompatible drivers. While this can prevent hardware event-based sampling and stack collection, other analysis types using user-mode sampling, such as Hotspots and Threading, can still be run. If the VTune Profiler is already installed when your Windows system is updated to an unsupported version, the data collector may cause a crash or BSOD while accessing the required drivers (sampling, graphics, or third-party drivers).

Solution

After installing the latest major Windows update, uninstall and reinstall the latest version of the VTune Profiler.

Make sure [all drivers](#) are up to date.

[Intel VTune Profiler Installation Guide for Windows](#)

Warnings about Accurate CPU Time Collection

The following table lists warning messages you may encounter on Windows* OS when collecting data in the [highly accurate CPU time detection](#) mode (enabled by default) and suggests solutions.

Warning	Cause and Possible Solution
<i>Accurate CPU time detection was disabled. Another collection in this mode is already running on the system.</i>	<p>Cause</p> <p>Another collection with accurate CPU time detection is running on the system or was not terminated properly.</p> <p>Solution</p> <p>Make sure the other collection has finished and try again. Only one VTune Profiler collection can run with accurate CPU time detection on the system at a time.</p>
<i>Accurate CPU time detection was disabled. The NT Kernel Logger is already in use.</i>	<p>Cause</p> <p>VTune Profiler requires the Microsoft® NT Kernel Logger to capture precise CPU time data. The NT Kernel Logger is a system-wide resource which cannot be shared by different processes. Other tools, such as the Xperf utility of the Windows® Performance Tools Kit may use the NT Kernel Logger at the same time.</p> <p>Solution</p>

Warning	Cause and Possible Solution
<p><i>Accurate CPU time detection was disabled. Drive <drive name> has not enough disk space.</i></p>	<p>Make sure that other tools do not use the NT Kernel Logger and try again.</p> <p>Cause</p> <p>The temporary folder for storing accurate CPU time data is located on a drive with insufficient free space.</p> <p>Solution</p> <p>Make sure you have enough space on the drive. The free space must be not less than specified in the Limit collected data by: Result size from collection start option on the WHAT pane of the Configure Analysis window.</p>
<p><i>Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.</i></p>	<p>Cause</p> <p>VTune Profiler requires administrative privileges to enable accurate CPU time detection.</p> <p>Solution</p> <p>Make sure you are running the collection as a local administrator.</p>
<p><i>Accurate CPU time detection was disabled. The temporary data path <full path to file> is longer than 1024 symbols.</i></p>	<p>Cause</p> <p>VTune Profiler uses temporary files to collect precise CPU time data. The length of the path to the temporary files cannot be longer than 1024 symbols.</p> <p>Solution</p> <p>Make sure the TEMP environment variable for the path to the temporary files is no longer than 1024 symbols.</p>
<p><i>Accurate CPU time detection was disabled. This mode is not supported on this OS.</i></p>	<p>Cause</p> <p>Some operating systems may not support accurate CPU time detection. For the list of supported operating systems, please see the <i>System Requirements</i> section of the product <i>Release Notes</i>.</p>

See Also

[Control Data Collection](#)

Reference

Explore the following reference information for Intel® VTune™ Profiler:

- [Graphical User Interface Reference](#)
- [CPU Metrics Reference](#)
- [GPU Metrics Reference](#)
- [OpenCL™ Kernel Analysis Metrics Reference](#)
- [Energy Analysis Metrics Reference](#)
- [Intel Processor Events Reference](#)

User Interface Reference

This section provides reference context-sensitive topics for Intel® VTune™ Profiler user interface elements, typically accessed from the product via **Learn More** link,



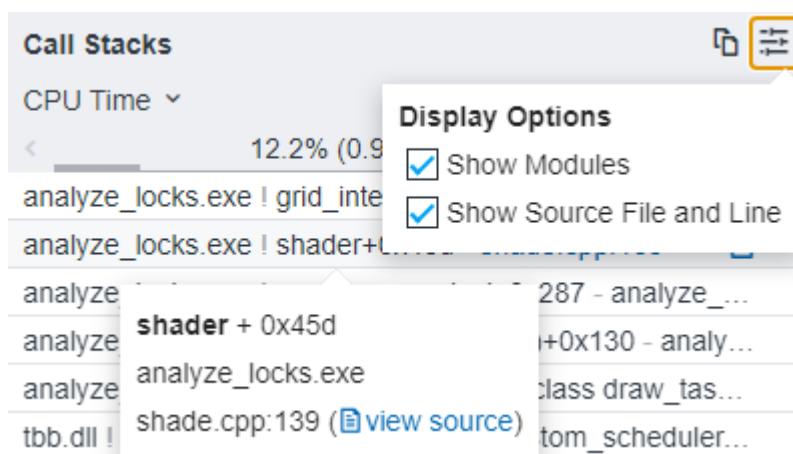
Context Help button, or F1 button.

- Context Menu: Grid
- Context Menu: Call Stack Pane
- Context Menu: Project Navigator
- Context Menu: Source/Assembly Window
- Dialog Box: Binary/Symbol Search
- Dialog Box: Source Search
- Hot Keys
- Menu: Customize Grouping
- Menu: Intel VTune Profiler
- Pane: Call Stack
- Pane: Options - General
- Pane: Options - Result Location
- Pane: Options - Source/Assembly
- Pane: Project Navigator
- Pane: Timeline
- Toolbar: Command
- Toolbar: Filter
- Toolbar: Source/Assembly
- Toolbar: Intel VTune Profiler
- Window: Bandwidth - Platform Power Analysis
- Window: Bottom-up
- Window: Caller/Callee
- Window: Cannot Find file type File
- Window: Collection Log
- Window: Compare Results
- Window: Configure Analysis
- Window: Core Wake-ups - Platform Power Analysis
- Window: Correlate Metrics - Platform Power Analysis
- Window: CPU C\P States - Platform Power Analysis
- Window: Debug
- Window: Event Count
- Window: Flame Graph
- Window: Graphics - GPU Hotspots
- Window: Graphics C/P States - Platform Power Analysis
- Window: NC Device States - Platform Power Analysis
- Window: Platform
- Window: Platform Power Analysis
- Window: Sample Count
- Window: SC Device States - Platform Power Analysis
- Window: Summary
- Summary - Input and Output
- Summary - Microarchitecture Exploration
- Summary - GPU Hotspots
- Summary - Hardware Events
- Summary - Hotspots by CPU Usage
- Summary - HPC Performance Characterization
- Summary - Memory Consumption
- Summary - Memory Usage
- Summary - Platform Power Analysis
- Window: System Sleep States - Platform Power Analysis
- Window: Temperature - Platform Power Analysis
- Window: Timer Resolution - Platform Power Analysis
- Window: Top-down Tree
- Window: Uncore Event Count
- Window: Wakelocks - Platform Power Analysis

Context Menu: Grid

Right-click a column in a grid pane (for example, **Bottom-up**) to access the options available from the context menu:

Use This	To Do This
View Source	Open the Source/Assembly window of the selected program unit.
Change Focus Function	Use a function selected in the Callers or Callees pane as a focus function and display its parent and child functions.
What's This Column?	Open a help topic describing the selected metric column.
Show Data As	Specify the data format for the collected data (for example, time, percent, bar, counts, and others). This option is available for columns displaying numeric data.
Hide Column	Hide the selected column.
Show All Columns	Show all the columns.
Select All	Select all items in the grid. The Selected data row at the bottom of the grid is updated to sum up all selected data per metric. Selecting data in one of the panes, Bottom-up or Top-down Tree , automatically updates the other pane and Call Stack pane.
Expand Selected Rows	Expand all child entries for the selected row(s).
Collapse All	Collapse all rows in the grid.
Find	Open a search bar and search for a string in the grid.
Export to CSV...	Export the content of the active pane to CSV format.
Copy Rows to Clipboard	Copy the content of the selected rows or a cell into the clipboard buffer.
Copy Cells to Clipboard	
Filter In by Selection	Filter in the grid and Timeline pane based on the currently selected rows. Selecting this menu item updates the filter bar based on the current selection. All rows except for the selected ones will be hidden. To show rows again, use the 
Filter Out by Selection	Clear all filters button on the Filter toolbar. If you applied filters available on the Filter bar to the data already filtered with the Filter In/Out by Selection context menu options, all filters are combined and applied simultaneously.
Filter Out by Selection	Filter out the grid and Timeline pane based on the currently selected rows. Selecting this menu item updates the filter bar based on the current selection. All selected rows will be hidden. To show rows again, use the 
Show Grouping Area	Clear Filter button in the Filter toolbar. If you applied filters available on the Filter bar to the data already filtered with the Filter In/Out by Selection context menu options, all filters are combined and applied simultaneously.
Show Grouping Area	Show/hide the Grouping drop-down menu at the top of the Bottom-up pane.

See Also[Window: Bottom-up](#)[Window: Top-down Tree](#)[Window: Caller/Callee](#)[Pane: Timeline](#)[Toolbar: Filter](#)[Source Code Analysis](#)**Context Menus: Call Stack Pane**

Use the available controls to address a number of options:

Use This	To Do This
View Source hyperlink	Open the Source/Assembly window for the program unit in the selected stack.
Show Modules toggle	Display the module names of the program units selected in the Call Stack pane.
Show Source File and Line toggle	Display the source file names of the program units selected in the Call Stack pane and a line number where the call was made.
Stack Selector	Switch between available stacks using the left/right arrows.
Copy to Clipboard button	Copy the data into the clipboard buffer to paste it to a different location.
Stack Type drop-down menu	Select a metric to arrange the stack by.

See Also[Viewing Source](#)[Pane: Call Stack](#)**Context Menus: Project Navigator**

Manage Intel® VTune™ Profiler projects/results using the **Project Navigator** context menus.

Directory Context Menu

Right-click the directory of the current project to choose one of the following options:

Use This	To Do This
New Project...	Open the Create a Project dialog box to browse to or create a directory in which the Intel® VTune™ Profiler will create a project (config.amplxeproj).
Open Project from New Location	Open the Select Project dialog box to browse to a directory containing VTune Profiler projects.
Copy Path to Clipboard	Copy the path to the currently opened project to the system clipboard.

Project Context Menu

Right-click a project to access the following options:

Use This	To Do This
Open Project	Open the VTune Profiler project.
Close Project	Close the current project and any opened results.
Configure Analysis...	Open the Configure Analysis window to modify project properties including a target system, a target type, and an analysis type.
<analysis type> Analysis	Rerun a recent analysis.
Close All Results	Close all opened results for this project.
Delete Project	Immediately delete the selected project and associated results from the Project Navigator and file system.
Rename Project	Rename the selected project in the Project Navigator immediately and in the file system after you close the project or exit the VTune Profiler.
Copy Project Path to Clipboard	Copy the path to the selected project to the system clipboard.

Result Context Menu

Right-click the result to choose one of the following options:

Use This	To Do This
Open Result	Open the VTune Profiler result.
Re-resolve and Open	Finalize the selected result again. You may use this option after changing the search directories settings to enable updating the symbol information. This option is available if the result is NOT open in the grid.
Compare	Open the Compare Results window and select a result to compare the current result with.
Delete Result	Delete the selected result from the Project Navigator and file system.
Rename Result	Rename the selected result in the Project Navigator immediately and in the file system after you close the result or project, or exit the VTune Profiler.

Use This	To Do This
	NOTE
	The corresponding result directory in the file system is not renamed.
Copy Result Path to Clipboard	Copy the path to the selected result to the system clipboard.

See Also

[Pane: Project Navigator](#)

[Set Up Project](#)

[Set Up Analysis Target](#)

[Analyze Performance](#)

[VTune Profiler Filenames and Locations](#)

[Manage Data Views](#)

[Finalization](#)

Context Menus: Source/Assembly Window

Manage the data in the **Source/Assembly** panes using one of the following mechanisms:

- Right-click the source/assembly code column to access the code column context menu.
- Right-click a data column with numeric data (for example, CPU Time) to access the data column context menu.

The following context menu options are available:

Use This	To Do This
Edit Source	Launch the source file editor. This option is only available for the Source pane.
Instruction Reference	Open the Reference help system for particular assembly instruction. This option is only available for the Assembly pane.
What's This Column?	Open a help topic for the selected performance metric column.
Show Data As	<p>Specify the format to display the collected data. You can view the data as:</p> <ul style="list-style-type: none"> • Time • Percent • Bar • Time and Bar • Percent and Bar <p>This option is only available for columns displaying numeric data.</p>
Hide Column	Hide the selected column.
Show All Columns	This option is only available for columns displaying numeric data.
	Show all the columns.

Use This	To Do This
	Define the current metric column in the Source and Assembly views. This option is only available for columns displaying numeric data.
Export to CSV	Export the content of the active pane to CSV format.
Select All	Select the content of the whole table.
Find	Open the search bar and search for a string .
Copy Rows to Clipboard	Copy the content of the selected rows into the clipboard buffer.
Copy Cell to Clipboard	Copy the content of the selected cell into the clipboard buffer.

See Also

[Source Code Analysis](#)

Dialog Box: Binary/Symbol Search

Use the **Binary/Symbol Search** dialog box to configure the search directories for binary and symbol files *on the host*, which is required to [finalization](#) and accurate source analysis. For example, specify non-standard directories for the supporting files needed to execute the target executable.

For remote data collection, if the symbol files are not available on the host, make sure to either copy them to the host or mount the directory with the source files and add it to the search paths. Binary files are copied from the target system to the host by default after data collection.

To access this dialog box:

1. On the Intel® VTune™ Profiler toolbar, click the



Configure Analysis button.

The result tab opens the **Configure Analysis** window.

2. Specify your analysis system on the **WHERE** pane and analysis target on the **WHAT** pane.
3. Click the



Search Sources/Binaries button on the command toolbar at the bottom.

4. In the dialog box, select **Binaries/Symbols** from the left pane.

To manage the search directories list, hover over a respective line to see the action buttons.

Use This	To Do This
Search Directories list button	<ul style="list-style-type: none"> • Add non-standard directories to the list. • View the directories currently in the search list, including their search order. <p>Browse for directories to include to the search list.</p>
<Add a new search location> field	<p>Add a new local search directory or a symbol server paths to the list by clicking the field and typing the path and name of the directory in the activated text box.</p> <p>If running an analysis from the standalone VTune Profiler GUI on Windows* OS, make sure to configure the Microsoft* symbol server by adding the following line to the list of search directories:</p>

Use This	To Do This
	<p><code>srv*C:\local_symbols_cache_location*http://msdl.microsoft.com/download/symbols</code></p> <p>where <code>local_symbols_cache_location</code> is the location of local symbols. VTune Profiler will download debug symbols for system libraries to this location and use them to resolve collected data and provide accurate performance data for system modules.</p>
	<p>NOTE The search is non-recursive. Make sure to specify correct paths to the binary/symbol files.</p>
 button	Move the selected directory up the search priority list.
 button	Move the selected directory down the search priority list.
 button	Remove the selected directory from the list.

See Also

[Search Directories](#)

[Dialog Box: Source Search](#)

[Debug Information for Windows* Application Binaries](#)

[Enable Linux* Kernel Analysis](#)

[Debug Information for Windows* System Libraries](#)

[Specifying Search Directories](#)

from command line

Dialog Box: Source Search

Use the **Source Search** dialog box to specify the directories used to search for source files *on the host*, which is required for data [finalization](#) and accurate source analysis. For remote data collection, if the source files are not available on the host, make sure to either copy them to the host or mount the directory with the source files and add it to the search paths.

To access this dialog box:

1. On the Intel® VTune™ Profiler toolbar, click the



Configure Analysis button.

The result tab opens the **Configure Analysis** window.

2. Specify your analysis system on the **WHERE** pane and analysis target on the **WHAT** pane.
3. Click the



Search Sources/Binaries button on the command toolbar at the bottom.

4. In the dialog box, select **Sources** from the left pane.

To manage the search directories list, hover over a respective line to see the action buttons.

Use This	To Do This
Search Directories list	<ul style="list-style-type: none"> • Add non-standard directories to the list. • View the directories currently in the search list, including their search order.
button	Browse for directories to include to the search list.
<Add a new search location> field	Add a new local search directory to the list by clicking the field and typing the path and name of the directory in the activated text box.
	<p>NOTE The search is non-recursive. Make sure to specify correct paths to the source files.</p>
button ↑	Move the selected directory up the search priority list.
button ↓	Move the selected directory down the search priority list.
button ✎	Remove the selected directory from the list.

See Also

[Search Directories](#)

[Dialog Box: Binary/Symbol Search](#)

[Debug Information for Windows* Application Binaries](#)

[Specifying Search Directories
from command line](#)

Hot Keys

Use hot keys supported by the Intel® VTune™ Profiler to quickly perform various tasks:

Use This	To Do This
Alt + 1	Launch the VTune Profiler and start the analysis of the selected type, or resume the data collection after it has been paused.
Alt + Break	Pause the current data collection.
Alt + Shift + 1	Stop the current data collection.
Alt + 9	Open the Configure Analysis window to choose and run a new analysis.
Ctrl + O	Open the Select Result dialog box to select and open an existing analysis result.

NOTE

You may program hot keys to start/stop a particular analysis. For more details, see <http://software.intel.com/en-us/articles/using-hot-keys-in-vtune-amplifier-xe/>.

Menu: Customize Grouping

Use the **Customize Grouping** menu to create a custom grouping of program units for the current viewpoint.

Typically default groupings provided in the VTune Profiler are enough for basic analysis workflows. But you may organize the collected data to explore it from a different perspective. For this, click the



Customize Grouping button in the grid view and combine a grouping you need.

Use This	To Do This
List of available grouping levels	Select grouping levels required for your custom grouping. This list provides all levels supported by the Intel® VTune™ Profiler. Make sure to select grouping levels applicable to your analysis type.
Custom grouping field	View the custom grouping you created. The grouping shows up in the Grouping menu in the order presented in this field. If the grouping uses levels not applicable to the current analysis, no data is shown in the grid.
Left and Right arrows	Use the left and right arrows to add/remove the grouping levels in the custom grouping. Use double right arrows to remove all levels from the custom grouping.
Up and Down arrows	Modify the order of grouping levels selected for the custom grouping.

The grouping you create is added to the **Grouping** menu for the current session and automatically removed when you close the result.

See Also

[Grouping and Filtering Data](#)

Menu: Intel VTune Profiler

If you work with the Microsoft Visual Studio* environment, a new Intel® VTune™ Profiler menu item appears under the Microsoft Visual Studio* **Tools** menu after the product installation. This menu contains commands for accessing all commonly used VTune Profiler features. This includes menu items to run and control performance analysis for the current solution.

These are the commands available from the Intel VTune Profiler toolbar in Visual Studio IDE:

Icon	Command	Description
	Open VTune Profiler	Open VTune Profiler within Microsoft Visual Studio IDE.
	Configure Analysis with VTune Profiler	Configure your VTune Profiler project and profile your target with VTune Profiler.

In the Visual Studio IDE sub-menu (**File > Intel VTune Profiler**), these options are available:

Icon	Command	Description
	Open VTune Profiler	Open VTune Profiler within Microsoft Visual Studio IDE.
	Configure Analysis with VTune Profiler	Configure your VTune Profiler project and profile your target with VTune Profiler.

Icon	Command	Description
	Import Result...	Open the Import window to import a data file, such as *.tb6.
	Compare...	Open the Compare Results dialog box and specify analysis results to compare. You can compare only the results of the same analysis type.

To access the VTune Profiler menu in the standalone GUI, click the button in the



Menu button in the upper left corner. The following commands are available:

Command	Hot Keys	Description
Welcome		Open the Welcome page that provides direct access to most recent projects and results. You can also use this page to open or create a VTune Profiler project or access the latest technical articles on the product functionality
Help Tour		Launch an interactive tour around the product that uses a sample pre-collected result to demo basic product functionality.
New > Project...	CTRL +SHIFT +N	Create a new VTune Profiler project that introduces your analysis target.
New > Compare Results...	CTRL+ALT +O	Open the Compare Results dialog box and specify analysis results to compare. You can compare only the results of the same analysis type.
New > Analysis...	CTRL+N	Open the Configure Analysis window to choose, configure, and run an analysis.
New > <analysis type> Analysis		Run the specified analysis types without opening the Configure Analysis window. For your convenience, this list of analysis types includes the most recent configurations you ran.
Open > Project...	CTRL +SHIFT +O	Open an existing VTune Profiler project to introduce your analysis target and start analysis.
Open > Result...	CTRL+O	Open an existing analysis result.
Close Project		Close the currently opened project.
Import Result...	CTRL+ALT +N	Open the Import window to import a data file, such as *.tb6.
Recent Projects		Quickly open a recently used VTune Profiler project.
Recent Results		Quickly open a recently collected analysis result.
View > Project Navigator		Open the project navigator window to explore the currently selected project.
Options...		Open the Options dialog box to configure general , result name , or source/assembly options .
Help > <doc_format>		Open one of the following online documentation format for the VTune Profiler:

Command	Hot Keys	Description
		<ul style="list-style-type: none"> • Intel VTune Profiler version User Guide • Get Started with Intel VTune Profiler version • VTune Profiler Developer Forum • Cookbooks and Tutorials • Intel Processor Event Reference
Help > Additional Resources		Access VTune Profiler documentation on the Intel Developer Zone or download it for offline usage.
Help > About...		View product license and product details.
Exit		Exit the VTune Profiler standalone interface.

See Also

[Toolbar: Intel VTune Profiler](#)

[Window: Compare Results](#)

[Compare Results](#)

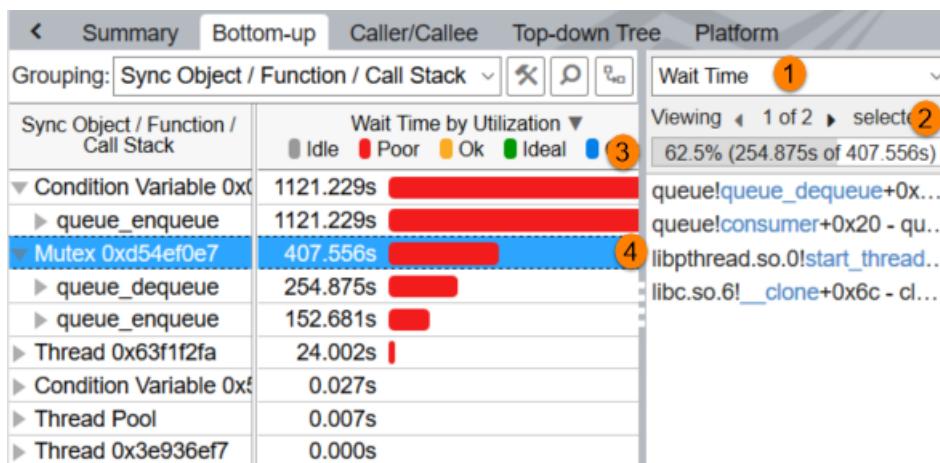
[Finalization](#)

Pane: Call Stack

The **Call Stack** pane is available for these collections:

- [User-mode sampling and tracing](#) collection, such as Hotspots and Threading analyses
- [GPU Offload analysis](#)
- [Hardware event-based sampling with the stack collection enabled](#)

Use the **Call Stack** pane to identify the call sequences (stacks) that called the program unit selected in the grid. Call stacks from different threads are aggregated together, showing all the call stacks for a function, without providing information on what threads were calling. See the table below to understand how to use the data provided in the **Call Stack** pane for the Threading analysis results.



1 **Stack metric drop-down menu.** Select a performance metric to explore the distribution of this metric over stacks of the selected object. For example, for the Threading Efficiency viewpoint the Wait Time metric is preselected. For the GPU Offload viewpoint, the Execution metric is preselected.

2 **Navigation bar.** Click the next/previous



arrows to view stacks for the selected program unit(s). The [stack types](#) are classified by metrics and depend on the selected viewpoint. For example, for the **Threading Efficiency** viewpoint the **Wait Time** stack type displays call stacks where the object selected in the grid contributed to the application Wait time.

When multiple stacks lead to the selected program unit, the **Call Stack** pane shows the stack that contributed most to the metric value, the *hottest path*, as the first stack. To see other stacks, click the navigation arrows.

NOTE

- If several stacks go to the same functions in different code lines, the bottom-up tree shown in the **Bottom-up** grid aggregates these stacks in one line. But the **Call Stack** pane shows each as a separate stack.
- If a selected stack type is not applicable to a selected program unit, the VTune Profiler automatically uses the first applicable stack type from the stack type list instead.

3 **Contribution bar.** Analyze the indicator of the contribution of the currently visible stack to the overall metric data for the selected program unit(s). If you select a single stack in the result window, the Contribution bar shows 100%. If more than one program unit is selected, all the related stacks are added to the calculation.

In the example above, the function selected in the **Bottom-up** grid had 3 Wait Time stacks leading to it with the total Wait time 23.718 seconds. The first stack is responsible for 97.9% (or 23.230s) of the overall 23.718 seconds. Note that the Bottom-up grid aggregates all 3 stacks into one since all of them go to the same function in different code lines.

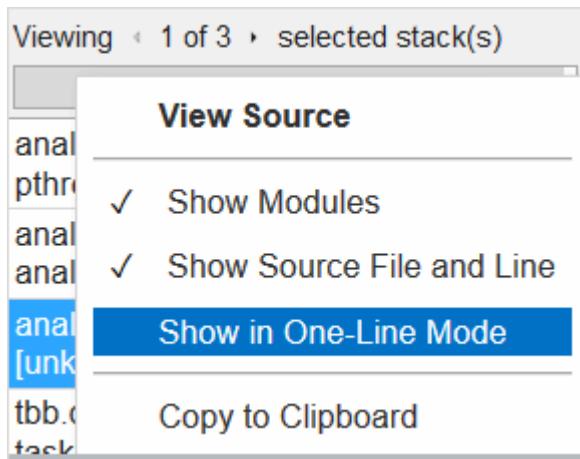
4 **Call stack for a program unit selected in the grid or in the Timeline pane.** Analyze the call sequence for the selected function according to the stack metric selected in the navigation bar. Each row in the stack represents a function (with an RVA and a line number of the call site, if available) that called the function in the row above it. When the **Call Stack Mode** on the filter toolbar is set to **Only user functions**, the system functions are shown at the bottom of the stack. When set to **User/system functions**, the system functions are shown in the correct location, according to the call sequence.

Click a hyperlink or double-click a function in the stack to open the source exactly where this function was called.

NOTE

If you see [Unknown frame(s)] identifiers in the stack, it means that the VTune Profiler could not locate symbol files for system or your application modules. See the [Resolving Unknown Frames](#) topic for more details.

Context menu. Manage the call stack representation in the **Call Stack** pane (applicable to all stacks). Right-click and select an option. For example, you may de-select the **Show in One-Line Mode** option to view functions in two lines:

**NOTE**

When you compare two analysis results, the **Call Stack** pane does not show any call stacks.

See Also

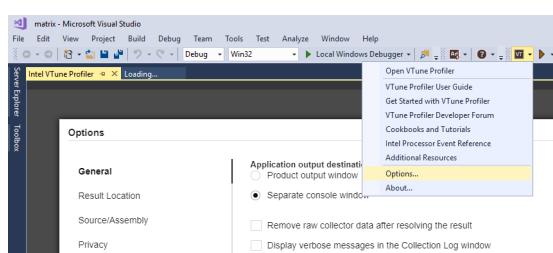
[Metrics Distribution Over Call Stacks](#)

[View Stacks](#)

[Context Menus: Call Stack Pane](#)

Pane: Options - General**To access this pane:**

In the Microsoft Visual Studio IDE, click the pull-down menu next to the **Open VTune Profiler** icon () and select **Options**:



From the standalone VTune Profiler interface: Click the 

menu button and select **Options... > Intel VTune Profiler version > General**.

The following options are available:

Use This	To Do This
Application output destination options	Choose the location for the output of the analyzed application:

Use This	To Do This
	<ul style="list-style-type: none"> Product output window: Direct the application output to the Application Output pane in the Collection Log window. Separate console window: Direct the application output to a separate console window (default). Microsoft Visual Studio* output window: View the application output in the Microsoft Visual Studio* output window. Use this option to see the output during the analysis.
Remove raw collector data after resolving the result check box	Enable/disable removing raw collector data after finalizing the result. Removing raw data makes the result file smaller but prevents future re-finalization.
Display verbose messages in the Collection Log window check box	Enable/disable detailed collection status messages in the Collection Log window. Make sure to re-open the result to apply this change.
Show all applicable viewpoints check box	Display all applicable viewpoints in the viewpoint selector for every analysis type.
Specify path to the adb executable field	Specify the path to the adb executable used to access an Android* device for analysis with the VTune Profiler.

See Also

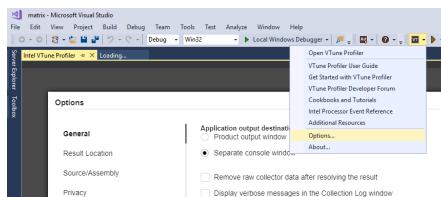
[Set Up Android* System](#)

[Microsoft Visual Studio* Integration](#)

Pane: Options - Result Location

To access this pane:

From Microsoft Visual Studio* IDE: Click the pull-down menu next to the **Open VTune Profiler** icon () and select **Options**:



From standalone VTune Profiler interface: Click the 

menu button and select **Options... > Intel VTune Profiler version > Result Location**.

Use the **Result Location** pane to configure the following options:

Do This	To Do This
Result name template text box	Change the default template defining the name of the result file and its directory.

Do This	To Do This
	<p>NOTE Do not remove the @@@ part from the template. This is a placeholder enabling multiple runs of the same analysis configuration.</p>

See Also

[VTune Profiler Filenames and Locations](#)

[Specify Result Directory from Command Line](#)

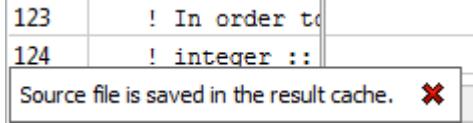
[Microsoft Visual Studio* Integration](#)

Pane: Options - Source/Assembly

To access this pane:

Go to **Tools > Options > Intel VTune Profiler version> Source/Assembly**.

Use this pane to configure the following options:

Use This	To Do This
Tab size: text box	Set the tab character display width in white spaces. The tab size should be an integer starting from 1.
CPU assembly syntax	Specify a formatting option to display the disassembled code: <ul style="list-style-type: none"> Default syntax: Show disassembled code using default syntax (MASM style for Windows* and GAS style for Unix*). GAS style syntax: Show disassembled code using GNU assembler syntax. MASM style syntax: Show disassembled code using MASM syntax.
Cache source files check box	Save your source files in the cache. You can go back to the cached sources at any time in the future and explore the performance data collected per code line at that moment of time. If you enable this option, the VTune Profiler caches your sources in the result database when you open the Source window for the first time and provides the following message:  <p>The message box shows two lines of text: "123 ! In order to" and "124 ! integer ::". Below these lines is a message: "Source file is saved in the result cache." followed by a red X button.</p>

When you open the **Source** window for this result for the second time, one of the following behaviors is possible:

- If the source file has not been changed, the VTune Profiler opens the source from the located source path. The message about caching the source file shows up at the bottom. The



- Open Source File Editor** toolbar button is enabled.
- If the source files have been changed, the VTune Profiler opens the source from the cached file and provides a proper notification on this at the bottom. The

Use This	To Do This
	<p> Open Source File Editor toolbar button is disabled.</p> <p>NOTE</p> <ul style="list-style-type: none"> • VTune Profiler opens previously cached source files even if the Cache source files option is disabled now. • If you have the Cache source files option enabled and open a changed source file that does not match the selected result, the VTune Profiler will cache it but will not use it for this result.

See Also

[Source Code Analysis](#)

[Microsoft Visual Studio* Integration](#)

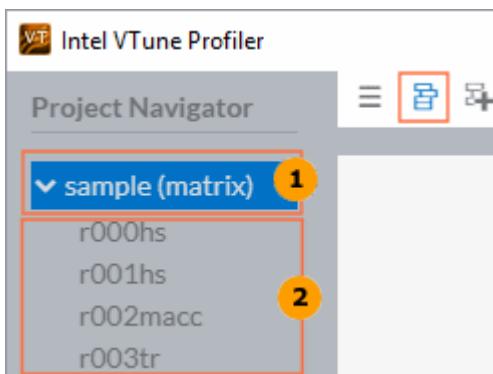
Project Navigator

The **Project Navigator** pane provides a hierarchical view of your projects and results based on the directory where the opened project resides.

To access this pane: Click the



Project Navigator icon on the Intel® VTune™ Profiler toolbar in the standalone graphical interface. To manage VTune Profiler projects/results from the Microsoft Visual Studio* IDE, use the Solution Explorer functionality.



Use this pane to perform the following actions:

- Delete a selected project or result.
- Rename a selected project or result.
- Close all opened results.
- Copy various directory paths to the system clipboard.

Use This	To Do This
1 Project node	<p>Double-click to open the project. Right-click the project node to access the project context menu.</p> <p>NOTE Opening a project closes the currently opened project.</p>

Use This	To Do This
2	Result node Double-click to open the result. Right-click the result node to access the result context menu .

NOTE

Opening a result opens the associated project if it is not already open.

See Also

[VTune Profiler Filenames and Locations](#)

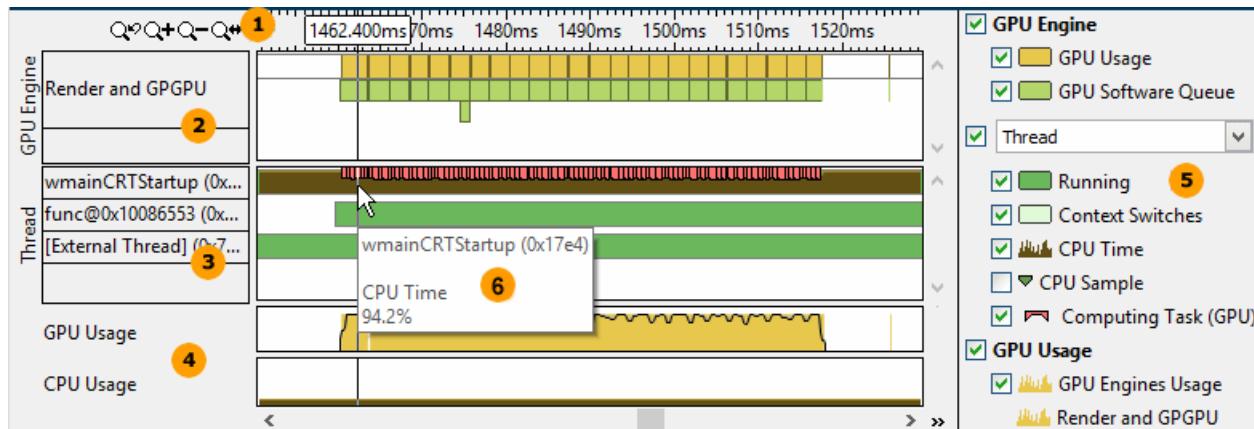
[Set Up Project](#)

Pane: Timeline

Use the **Timeline** pane to visualize metrics over time at either the thread level or platform level and identify patterns, anomalies, and trends in the data.

You can hover, zoom-in, and filter the data at interesting points in time to get more detail. Typically the **Timeline** pane is located at the bottom of the window but for the views focused on the metrics distribution over time, it may occupy the upper or central part of the window. Data presented in the Timeline pane varies depending on the analysis type and [viewpoint](#).

The **Timeline** pane typically provides the following data:



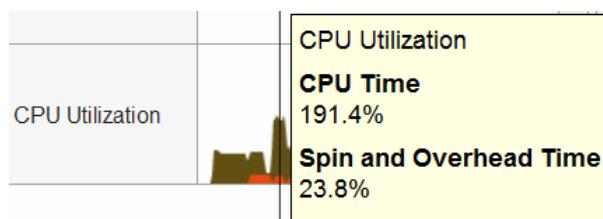
- 1 **Toolbar.** Navigation control to zoom in/out the view on areas of interest. For more details on the Timeline controls, see [Managing Timeline View](#) topic.
- 2 **Platform metrics.** Depending on the analysis type, the Timeline pane may present several areas with platform specific metrics such as [GPU engine usage](#), [computing queue for OpenCL™ applications](#), [bandwidth data](#), [power consumption](#), and so on. The most detailed analysis of the platform metrics is available with the Timeline pane in the [Platform window](#).
- 3 **Application metrics per grouping level.** Depending on the viewpoint, the data may be represented by threads, modules, processes, cores, packages, and other units monitored by the data collector during the analysis run. For most of the viewpoints, the **Thread** grouping is default. For some viewpoints, you may change the grouping level using the drop-down menu in the Legend area.

Note that the **CPU Time** metric value provided in the **Thread** area is applicable to a particular thread where 100% is the maximum possible utilization for a thread. For example, for the selection above 94.2% of CPU Time utilization means that the thread was active 94.2% of time and 5.8% it was waiting.

4

Selected metrics. Data on the most representative metrics may be presented as separate rows demonstrating an overall application performance over time (for example, CPU Usage or GPU HW metrics) or system-wide execution (for example, GPU Usage). See [Reference for Performance Metrics](#) for detailed metrics description.

Note that the **CPU Utilization** metric in the Timeline pane is calculated as a sum of CPU time per each thread where 100% is the maximum possible utilization per CPU. For example, at the moment selected in the picture below the application utilized 1.91 of logical CPU cores (if every CPU is 100%, then 191% is 1.91) out of 4, and 0.23 of CPU was used by the application threads for overhead or spinning. This means that the application utilized only 1.68 of CPUs effectively.



5

Legend. Types of data presented on the timeline. Filter in/out any type of data presented in the timeline by selecting/deselecting corresponding check boxes. The list of performance metrics presented in the view depend on the selected analysis type and viewpoint.

VTune Profiler also uses special indicators to classify the presented data on the timeline:

-



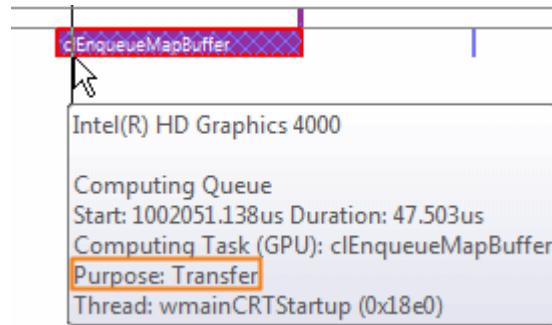
Markers. Color markers indicate an area on the timeline when a particular task/frame/event/etc. was executed. Hover over a marker to see the execution details for the selected element. The following markers are available:

- **Frame markers** show frame duration. Available for applications using [frames](#).
- **User task markers** provide information on a task executed at this particular moment of time. Available for applications using [Task API](#).
- **CPU sample markers** indicate exact points where profiling samples happened during [hardware event-based stack sampling collection](#). Use the markers density to estimate the data resolution. For example, the VTune Profiler interpolates the sampling data where accuracy depends on number of samples. In this case, the CPU Samples markers show more accurate information discovering the sporadic CPU utilization for the thread.

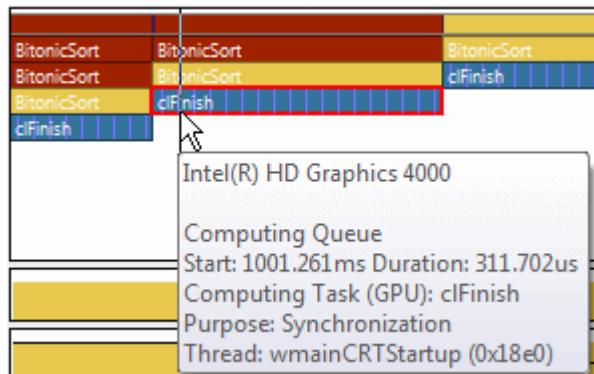
Sample markers also help understand how exactly filtering and Spin/Overhead time calculation works. VTune Profiler filters or classifies samples as a whole, so when you do time filtering it is important to know whether the sample point got into the selected time interval or not. No data interpolation is done for sampling data when filtering or classifying sample metrics.

- **VSync markers** for vertical synchronization. If your application uses vertical synchronization, you can select the **VSync** timeline option, estimate the correlation between VSync events and application frames, identify frames missing VSync events and explore possible reasons.

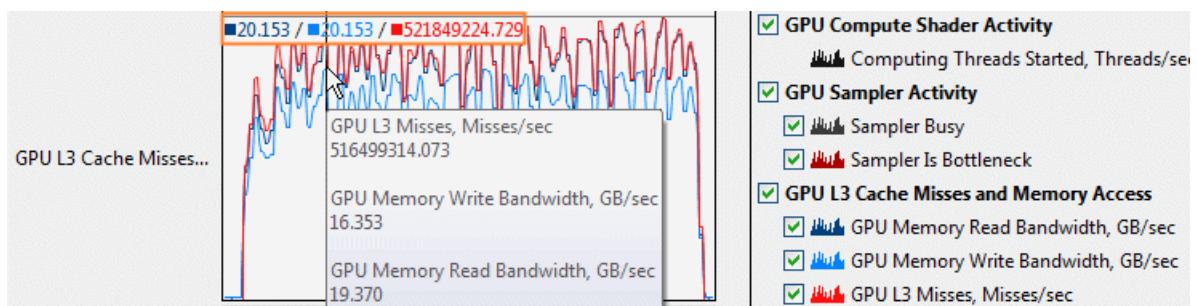
- **Sampling point markers** point at which a data sample was read during [energy analysis](#). Hovering over it gives the value(s) read at that time.
- **Wake-up object markers** for energy analysis that show processor wake-ups on the timeline. Hover over a yellow marker to see the time when the selected wake-up happened and the name of the wake-up object.
- **Slow tasks** markers show the duration of tasks (I/O Wait, Ftrace*, Atrace*, and so on) that is categorized as slow (according to the thresholds set up in the **Summary** window)
- **I/O APIs** markers
- **Context switches**. The time threads are spending on context switches. Hover over a context switch area to see the details on its duration, reason, and affected CPU. If you choose the **Context Switch Time** option in the **Call Stack** pane and select a context switch in the **Timeline** pane, the **Call Stack** pane shows a call sequence at which a preceding thread execution quantum was interrupted.
- **Transitions**. The execution flow between threads where one thread signals to another thread waiting to receive that signal. For example, one thread attempts to acquire a lock held by another thread, which then releases it. The release acts like a signal to the waiting thread. Hover over a transition for more details. Double-click a transition to open the source code.
- **Memory transfers**. OpenCL routines responsible for transferring data from the host system to a GPU are marked with cross-diagonal hatching on a computing queue:



- **Synchronizations**. OpenCL routines responsible for synchronization are marked with vertical hatching on a computing queue:



- **Scaling indicators**. For [GPU metrics](#) and bandwidth graphs, the VTune Profiler provides maximum Y-axis values used to scale the graphs. Color of such a value corresponds to the color of the relevant metric in the legend. For example, for the GPU L3 Cache Misses and Memory Access metrics, maximum Y value for the selected scale is 20.153 GB/sec for GPU Memory Read Bandwidth and for the GPU Memory Write Bandwidth, and 521849224.729 Misses/sec for GPU L3 Misses.



6

Tooltips. Hover over a chart element to get statistics on this metric/program unit for the selected moment of time.

For the GPU analysis of applications using OpenCL software technology, the Timeline pane in the **Graphics** window provides the following tabs:

- **Platform** tab that focuses on a per-thread and per-process distribution of the CPU and GPU hardware metrics collected during the analysis run.
- **Architecture Diagram** tab that is provided for OpenCL application analysis collected with the **Analyze Processor Graphics hardware events** option on systems with Intel® HD Graphics and Intel® Iris® Graphics. This tabs helps better understand the distribution of the GPU hardware metrics per architecture blocks for the period the selected OpenCL kernel was running.

NOTE

Collecting [energy analysis](#) data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Window: Bottom-up](#)

[Window: Top-down Tree](#)

[Window: Event Count - Hardware Events](#)

[Window: Uncore Event Count - Hardware Events](#)

[Pane: Call Stack](#)

Toolbar: Configure Analysis

Use the Intel® VTune™ Profiler command toolbar in the **Configure Analysis** window to access the project configuration options, manage your data collection (start, pause, resume, and so on) and analysis result (re-resolve, import).

The command toolbar shows up when you use one of the following options:

- Click the



(standalone GUI)/

(Visual Studio IDE) **Configure Analysis** button on the product toolbar.

- Windows* only: From the Microsoft Visual Studio* **Tools > Intel VTune Profiler <version>** menu, select the **Configure Analysis** option.
- From the standalone interface menu, select **New > Analysis....**

The VTune Profiler result tab opens providing the command bar on the right. The command bar is dynamically changing depending on the analysis phase. The following commands are available:

Use This Command Button	To Do This
 Start/Resume	Run the analysis, or resume the analysis after a pause. To enable this button: <ul style="list-style-type: none"> • Select a system for analysis on the WHERE pane • Specify an analysis target on the WHAT pane. If you work in Visual Studio, the project target is automatically associated with the current project. • Select an analysis type on the HOW pane.
 Start Paused	Launch the application but run the analysis after some delay. To resume the analysis, click the Resume button.
 Pause	Pause the data collection any time you need while the application is running. To resume the data collection, click the Resume button
 Stop	Stop the data collection. This button is only enabled during collection.
 Cancel	Cancel the data collection. This button is only enabled during collection.
 Mark Timeline	Mark an important moment in the application execution. These marks appear in the Timeline pane . This button is only enabled during collection.
 Search Sources/Binaries	Open the search dialog box with the Binary/Symbol Search tab to specify search directories for binary and symbol files in your project and the Source Search tab to specify search directories for source files in your project.
 Re-resolve	Finalize the result again. This button shows up on the command bar when you try to run the target after changes in the search directories settings.
 Import from CSV	Import external performance data into a VTune Profiler result as a <code>csv</code> file. You may collect the external performance data with a custom collector out of the VTune Profiler or with your target application used for the VTune Profiler analysis.
 Command Line	Generate a command line version of the selected configuration and save it to the buffer for running from a terminal window. You can use this approach to configure and run your remote application analysis.

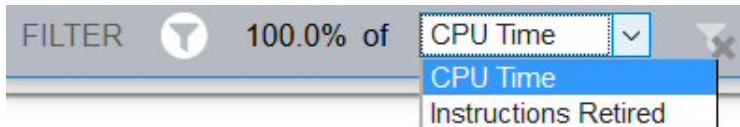
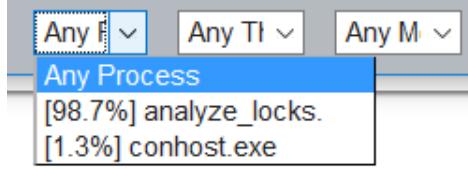
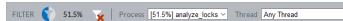
See Also

[Pause Data Collection](#)

[Finalization](#)

Toolbar: Filter

Use the Filter toolbar to filter the data displayed in the grid or **Timeline** pane. Filtering settings applied to the currently opened result are saved for the whole project and automatically applied to the subsequent results in this project.

Use This	To Do This
Metric filter	Mouse over the  Filter icon to enable the metric drop-down menu and select a filtering metric:
	
	By default, you see 100% of all metric data collected in the result. Metric values vary with a viewpoint and analysis type.
	For example, for the Hotspots viewpoint available for the Hotspots analysis result (hardware event-based sampling mode) there are CPU Time and Instructions Retired event metrics available, where the CPU Time is selected by default. Open any filtering drop-down menu to see the percentage of the CPU Time each module/process/thread introduces into the overall CPU Time for the result:
	
	If you select a program unit in the filtering drop-down menu, your grid and Timeline view will be filtered out to display data for this particular program unit. For example, if you select the analyze_locks process introducing 53.4% of the CPU Time, the result data will display statistics for this process only and the Filter bar provides an indicator that only 53.4% of the CPU Time data is currently displayed:
	
Module filter	Select a module to filter the collected data by its contribution. All data related to other modules is hidden. By default Any Module is selected. This option does not filter any data.
Thread filter	Select a thread to filter the collected data by its contribution. All data related to other threads is hidden. By default Any Thread is selected. This option does not filter any data.
Process filter	Select a process to filter the collected data by its contribution. All data related to other processes is hidden. By default Any Process is selected. This option does not filter any data.
Thread Efficiency filter	Select a thread efficiency level to filter the collected data by its contribution. All data related to other efficiency levels is hidden. By default Any Thread Efficiency is selected. This option does not filter any data. This filter is applied to the Hotspots by Thread Concurrency and Threading Efficiency viewpoints for user-mode sampling and tracing analysis results.

Use This	To Do This
Utilization filter	Filter data in the grid by available CPU utilization modes . This filter is applied to the Hotspots by CPU Utilization viewpoint for the user-mode sampling and tracing analysis results.
Sleep States filter	Select a sleep state (C0 - Cn) to filter the collected data by its contribution. The deeper the sleep state of the CPU is, the greater power savings are. This filter is available for Energy analysis results only.
Wake-up Reason filter	Filter data by types of the objects that force the processor to wake up. Possible wake-up reasons are timer, interrupt, IPI, and so on. This filter is available for Energy analysis results only.
Timer Type filter	Filter data by type of the timers that force the processor to wake up. Choose between User and Kernel Timers. This filter is available for Energy analysis results only.
 Clear Filter icon	Remove all filters and view all the available data.
Inline Mode option	Enable/Disable displaying performance data per inline functions . This option is available if information about inline functions is available in debug information generated by compilers. See View Data on Inline Functions for supported compilers and options.
Call Stack Mode option	Select whether to show system functions: <ul style="list-style-type: none"> Only user functions: Filter out all system functions. User/system functions: Do not filter any data. User functions + 1 (default): Filter out all system functions except those directly called from user functions.
Loop Mode option	Select a type of hierarchy to display loop data in the grid. The following types are available: <ul style="list-style-type: none"> Loops only: Display loops as regular nodes in the tree. Loop name consists of: <ul style="list-style-type: none"> start address of the loop number of the code line where this loop is created name of the function where this loop is created Loops and functions: Display both loops and functions as separate nodes. Functions only (default): Display data by function with no loop information.

NOTE

If you applied filters available on the Filter bar to the data already filtered with the **Filter In/Out by Selection** context menu options, all filters are combined and applied simultaneously.

See Also

[Group and Filter Data](#)

[Manage Grid Views](#)

[filter](#)

vtune option

call-stack-mode

vtune option

inline-mode

vtune option

loop-mode

vtune option

Toolbar: Source/Assembly

Use the Source/Assembly toolbar to navigate between the most performance-critical code sections (*hotspots*). In the **Source** pane, you can navigate between source code lines, in the **Assembly** pane you can navigate between assembly instructions.

Use This	To Do This
Source button	Toggle the Source pane on/off. This button is enabled only when both source and assembly code is available.
Assembly button	Toggle the Assembly pane on/off. This button is enabled only when both source and assembly code is available.
	Tile the Source and Assembly panes vertically.
Vertical Mode button	
	Tile the Source and Assembly panes horizontally.
Horizontal Mode button	
	Go to the code line that has the biggest hotspot navigation metric value in the selected function.
Go to Biggest Function Hotspot button	
	Go to the previous (by the hotspot navigation metric value) hot line in the selected function.
Go to Bigger Function Hotspot button	
	Go to the next (by the hotspot navigation metric value) hot line in the selected function.
Go to Smaller Function Hotspot button	
	Go to the code line that has the smallest hotspot navigation metric value in the selected function.
Go to Smallest Function Hotspot button	

Use This	To Do This
 Source File Editor button	Edit the source code in the default code editor. This option is available for the Source pane only.
 Find button (CTRL+F)	Search for a data string in the grid.
Assembly grouping menu	Group assembly instructions by one of the available granularity levels: <ul style="list-style-type: none"> • Address • Basic block/Address • Function range/Basic block/Address

NOTE

To select a hotspot navigation metric, right-click the required metric column in the Source view and select **Use for Hotspot Navigation**.

See Also

[Source Code Analysis](#)

Toolbar: Intel VTune Profiler

Here are the Intel® VTune™ Profiler toolbar buttons that enable you to control the analysis run:

Use This Button	To Do This
 Project Navigator (standalone client only)	Open the Project Navigator to manage the VTune Profiler projects and analysis results.
 (standalone)/  (Visual Studio*) Configure Analysis	Open the Configure Analysis window to select, configure, and run analysis.
 New Project (standalone client only)	Open the Create a Project dialog box to create and configure a project.
 Import Result (standalone client only)	Open the Import window and specify result or raw data collection file(s) to import into the current project. VTune Profiler creates a result with the imported data and locates it in the current project.

Use This Button	To Do This
	Open the Compare Results window and choose the results to compare.
	Navigate to a VTune Profiler data collection result (*.vtune file) and open it in the graphical interface.
	Set options to collect, display, and save profiling data. View privacy information about collected data.
	<p>Open the Help menu providing access to the following documentation formats:</p> <ul style="list-style-type: none"> • Help • Get Started that opens a start page with a list of documentation resources and product overview. • Developer Forum • Video and Articles that leads you to the product web page with How-to videos and technical articles. • Intel Processor Event Reference

NOTE

VTune Profiler toolbar icons look slightly different in different versions of the Microsoft Visual Studio* IDE. The **Compare Results** button is not available from the toolbar in the Microsoft Visual Studio* IDE.

VTune Profiler also provides a lightweight [integration to the Eclipse*](#) development environment, adding the following buttons in the Eclipse GUI:

Use This Button	To Do This
	Open the VTune Profiler standalone graphical interface and configure and run a performance analysis for your application.
	Open the VTune Profiler Get Started page providing access to the product documentation resources.

When you view results, VTune Profiler provides an additional toolbar for the **Bottom-up** and **Top-down Tree** windows:

Use This Button	To Do This
	Change the stack layout for the Call Stack grouping level.
View Stacks as a Chain/	
View Stacks as a Tree	

Use This Button	To Do This
 Find (CTRL+F)	Search for data in the Bottom-up , Top-down Tree , Source , or Assembly panes.
 Customize Grouping	Create a custom grouping for the current viewpoint using the Custom Grouping dialog box.

See Also

Menu: Intel VTune Profiler

Toolbar: Configure Analysis

Window: Bandwidth - Platform Power Analysis

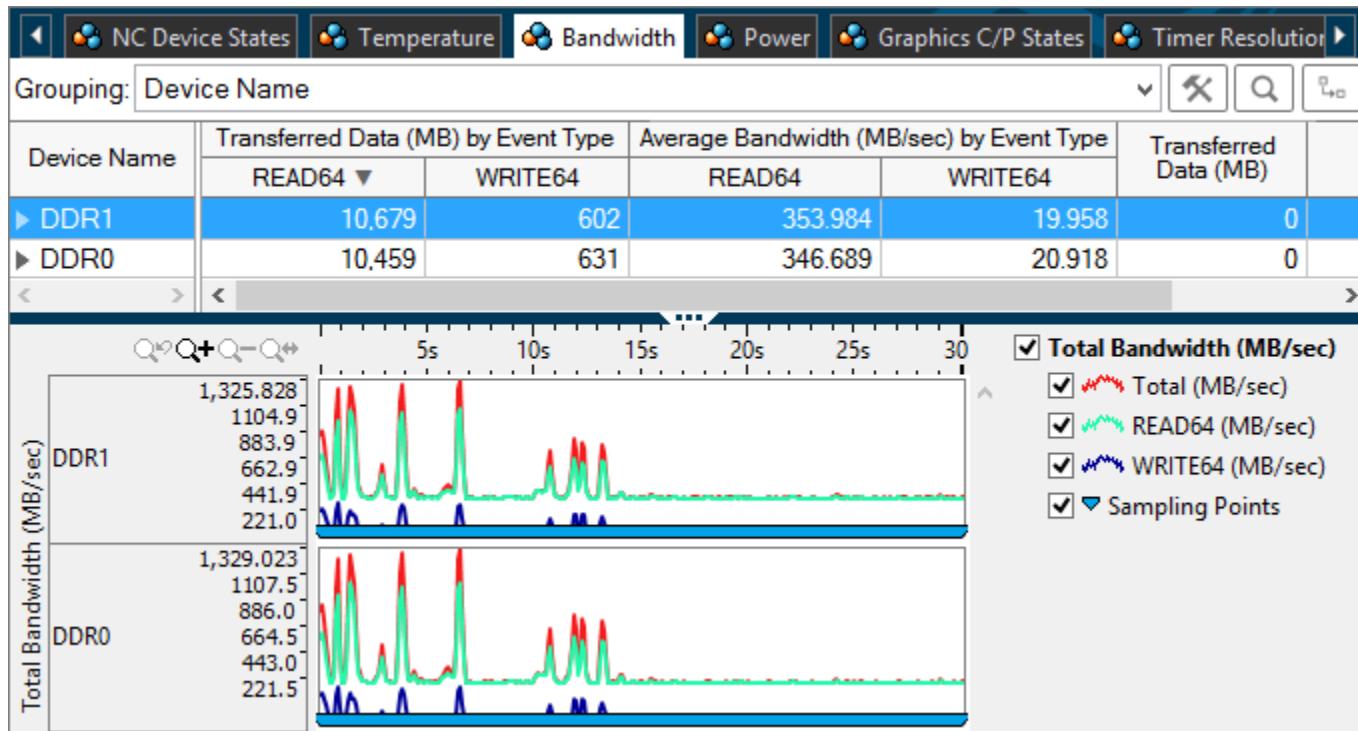
To access this window: Select the Platform Power Analysis viewpoint and click the **Bandwidth** sub-tab in the result tab.

Use the **Bandwidth** window to:

- Analyze the transaction rate for byte reads and writes.
- View an approximation of the different bandwidth types used by each component during collection (IA, GFX, IO).
- Review the DDR SDRAM memory events and bandwidth usage over time.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android®, Windows®, or Linux® devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.



Bandwidth Pane

The Bandwidth pane displays the bandwidth values for the data collected. Bandwidth data is collected as byte counts and is displayed as MB/sec. The bandwidth is given in both total values and the average bandwidth by event type and component. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit.

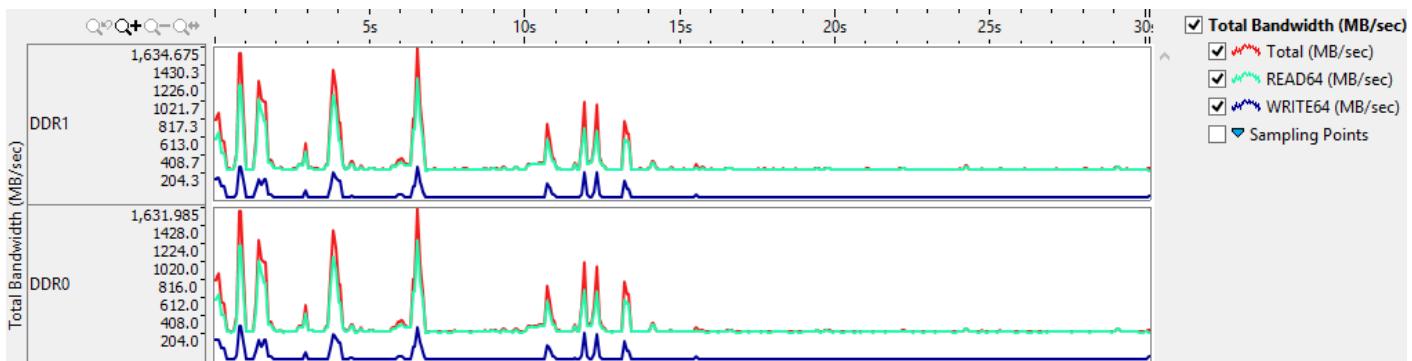
Grouping: Device Name						
Device Name	Transferred Data (MB) by Event Type		Average Bandwidth (MB/sec) by Event Type		Transferred Data (MB)	Average Bandwidth (MB/sec)
	READ64 ▼	WRITE64	READ64	WRITE64		
▶ DDR1	10,679	602	353.984	19.958	0	0.000
▶ DDR0	10,459	631	346.689	20.918	0	0.000

The average bandwidth displayed in this pane is typically the most important metric used to determine bandwidth usage during collection. The other columns display the number of bytes transferred by event and by the device or component.

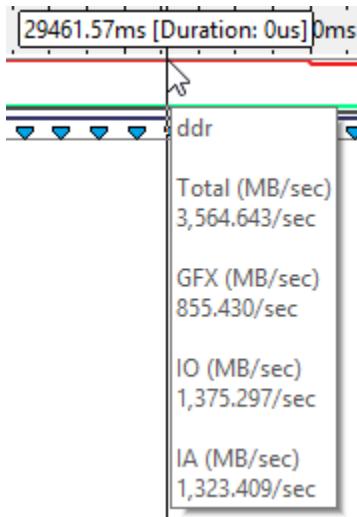
There are two types of bandwidth data that can be collected: approximate bandwidth and detailed bandwidth. Approximate bandwidth is measured across all devices with a lower level of detail. Detailed bandwidth allows in-depth collection for the specified device and events related to that device. The type of bandwidth collected is specified when running the Intel SoC Watch collector. For more information about the options to use for detailed bandwidth collection, see the *Intel SoC Watch User's Guide* for the operating system of your target device.

Timeline Pane

Use the Timeline pane to view bandwidth changes over time. Expand the timeline vertically to improve the data visualization and see more bandwidth values. Consider removing the Sampling Points from the timeline while viewing the full timeline to improve visibility to the lowest bandwidth values. You can add the sampling values again after zooming in on a section of the timeline.



Hover over the timeline to view a tooltip listing the exact bandwidth values at that time during the collection (MB/sec). The blue sampling points indicate the time at which the sample is obtained from the hardware. The duration between sampling points is the sampling interval that was specified at collection time.



Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

[Managing Timeline View](#)

Window: Bottom-up

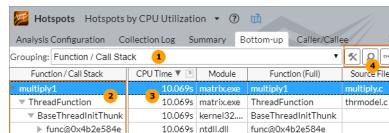
Use the **Bottom-up** window to analyze performance of each program from the bottom level when a child function is placed directly above its parent (bottom-up analysis).

To access this window: Click the **Bottom-up** tab. Depending on the analysis type, the **Bottom-up** window may include the following panes:

- [Bottom-up pane](#)
- [Call Stack pane](#)
- [Timeline pane](#)

Bottom-up Pane

Data provided in the Bottom-up pane depends on the analysis, data collection type, and viewpoint you apply.



1 **Grouping** menu. Each row in the grid corresponds to a **grouping level** (granularity) of program units (module, function, synchronization object, and others). For example, the data in the Hotspots viewpoint is grouped by **Function/Call Stack**.

2

Call stack. Analyze a tree hierarchy of the call stacks that lead to the selected program unit. Click the triangle sign to expand a row and view call trees for each program unit. Each tree is a call stack that called the selected unit. Each tree lists all the program units that had only one caller in the same row, with an arrow



indicating the call relationship. Program units that had more than one caller are split so that each caller has a separate row with the callers to that callee. If a function was called from different code lines (*call sites*) in the same parent function, the Bottom-up pane aggregates such stacks into one and sums up their CPU time. The full information on the stack is shown in the **Call Stack** pane.

The time value for a row is equal to the sum of all the nested items from that row.

NOTE

- Call stack information is always available for the results of the User-Mode Sampling collection. It is also available for the results of the hardware event-based sampling collection, if you enabled the **Collect stacks** option during the analysis configuration. Otherwise, the **Call Stack** column for the event-based results shows "Unknown" entries in the call tree.
- If you see [Unknown frame(s)] identifiers for the functions, it means that the VTune Profiler could not locate symbol files for system or your application modules. See the [Resolving Unknown Frame\(s\)](#) topic for more details.
- If the VTune Profiler does not find debug information in binaries, it statically identifies function boundaries and assigns hotspot addresses to generated pseudo names `func@address` for such functions, for example:

Function / Call Stack	CPU Time ▼
▼ func@0x6b29db95	2.405s
▶ ↵ pthread_mutex_lock ← draw_task	2.370s
▶ ↵ video::next_frame ← draw_task::o	0.036s
▶ GdipDrawImagePointRectI	0.990s

3

Performance metrics. Each data column in the grid corresponds to a [performance metric](#). By default, all program units are sorted in the descending order by metric values in the first column providing the most performance-critical program units first. You may click a column header to sort the table by the required metric.

The list of performance metrics varies depending on the analysis type. Mouse over a column header (metric) to read the metric description, or right-click and select the **What's This Column?** option from the context menu.

If a metric has a threshold value set up by the VTune Profiler architect and this value is exceeded, the VTune Profiler highlights such a value in pink. You may mouse over a pink cell to read the description of the detected issue and tuning advice for this issue.

For some analysis types, you may see grayed out metric values in the grid, which indicates that the data collected for such a metric is unreliable. This may happen, for example, if the number of samples collected for PMU events is too low. In this case, when you hover over such an unreliable metric value, the VTune Profiler displays a message: *The amount of collected PMU samples is too low to reliably calculate the metric.*

Depending on the analysis type and viewpoint, the Bottom-up view may represent the [CPU Time by utilization levels](#). Focus your tuning efforts on the program units with the largest **Poor** value. This means that during the execution of these program units your application underutilized the CPU time. The overall goal of optimization is to achieve **Ideal** (green



) or **OK** (orange)



) CPU utilization state and shorten the Poor and Over CPU utilization values.

Toolbar. Select the following options to manage the Bottom-up view:

4

- Click the



Customize Grouping button to open the [Custom Grouping](#) dialog box.

- Click the



Find button to open a search bar and search for a string in the grid.

- Click the

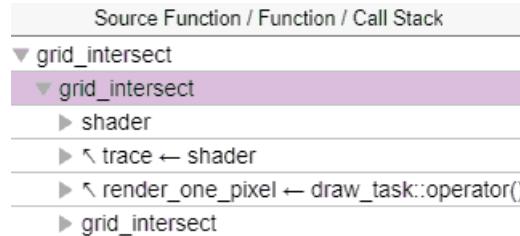


Change Stack Layout button to switch between call stack layouts.

Chain layouts



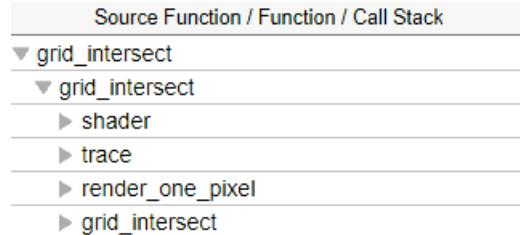
are typically more useful for the bottom-up view:



While tree layouts



are more natural for the top-down view:



See Also

[Manage Data Views](#)

[Reference](#)

[View Stacks](#)

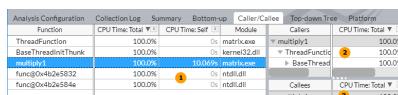
[Control Window Synchronization](#)

Window: Caller/Callee

To access this window: Click the **Caller/Callee** sub-tab in the result tab.

The Caller/Callee window is available in all viewpoints that provide call stack data.

Use this window to analyze parent and child functions of the selected *focus function* and identify the most time-critical call paths.



1 Functions pane. The Functions pane displays a flat list of functions providing data per the following metrics:

- **Self time:** Active processor time spent in a function.
- **Total time:** Active processor time spent in the function and its callees.

By default, the grid is sorted by the Total time metric. Select a function of interest in the grid (focus function) and explore its callers and callees on the right panes.

You may select a function of interest and filter the grid to display the functions included into all subtrees that contain the selected function at any level. To do this, select the function, right-click and choose the **Filter In by Selection** context menu option. For the call tree view, switch to the **Top-down Tree window**.

You can also change a focus function from the **Callers** or **Callees** panes by double-clicking a function of interest. Alternatively, you may select a function, right-click and choose the **Change Focus Function** context menu option.

VTune Profiler highlights this function in the **Functions** pane and updates the **Callers** and **Callees** panes to display its parent and child functions respectively.

You can double-click a function of interest in the **Functions** pane to go to the [source view](#) and explore the function performance by a source line.

2 Callers pane. The **Callers** pane shows parent functions (*callers*) for the function currently selected in the **Functions** pane.

3 Callees pane. The **Callees** pane shows child functions (*callees*) for the function currently selected in the **Functions** pane.

See Also

[CPU Metrics Reference](#)

Window: Cannot Find <file type> File

When you double-click a program unit in the analysis result, the Intel® VTune™ Profiler tries to open supporting module/source/symbol files. If it cannot locate the required file, the **Cannot find <file type> file** window appears, enabling you to enter the file manually. This window displays the original location of the file and provides the following options:

Use This	To Do This
Specify location of file to open text box	Specify the correct path to the file that is not found. You may choose the required file from the list. If the file you specify is invalid or partially valid, the VTune Profiler displays an error message.

Use This	To Do This
Add the directory to the search list as check box	Enable adding a new directory to the search list. This option is active when you enter a directory in the Specify location of file to open text box. To add a folder to the list of search directories for the current project, select it from the drop-down list. This helps locate the module/source/symbol files for the next analysis runs.
Assembly button on the toolbar	View the disassembly code for the current selection.
OK button	Close the window. If you provided a valid location in the Specify location of file to open text box, the VTune Profiler opens the source code for the selected item. If you cannot provide a valid location for the file, click the Assembly button on the toolbar to view the disassembly code or close the Source/Assembly window.
Skip button	Stop searching for symbol files and open the Source/Assembly window. This button is only available when a symbol file is not found.

See Also

[Dialog Box: Binary/Symbol Search](#)

[Dialog Box: Source Search](#)

[Search Directories](#)

Window: Collection Log

The **Collection Log** window opens when you click the **Start** button and run the analysis.

Intel® VTune™ Profiler uses two types of data collectors: [user-mode sampling and tracing collector](#) and [hardware event-based sampling collector](#). During data collection and finalization the VTune Profiler provides status messages in the **Collection Log** window. If required, you can click the



Clear Log button to delete the log.

NOTE

You may enable detailed collection messages by using the **Display verbose messages in the Collection Log window** option, available from the **Options... > Intel VTune Profiler > General** pane.

If analysis completes successfully, the VTune Profiler does the following:

- Creates an analysis result and saves it in the project directory. The project directory is specified in the **Configure Analysis** window > **WHAT** pane available via the



Configure Analysis toolbar button.

- (for VTune Profiler integrated into Visual Studio) Displays the analysis result in the Solution Explorer. The naming scheme of the analysis result is specified in the **Tools > Options... > Intel VTune Profiler > Result Location** pane.
- Opens the result tab with the default [viewpoint](#).

Application Output

If you configured the [General pane](#) options to display the application output in the product output window, the VTune Profiler redirects the output to the **Application Output** pane.

See Also

[Control Data Collection](#)

[Finalization](#)

[Troubleshooting](#)

Window: Compare Results

To access this window:

Click the **Compare Results**



button on the Intel® VTune™ Profiler toolbar.

You can compare two results that have common performance metrics. VTune Profiler provides comparison data for these common metrics only.

Dialog Item	Description
Result 1 / Result 2 drop-down menu	Specify the results you want to compare. Choose the result of the current project from the drop-down menu, or click the Browse button to choose a result from a different project.
Swap Results button	Click this button to change the order of the result files you want to compare. Result 1 always serves as the basis for comparison.
Compare button	Click this button to view the difference between the specified result files. This button is only active if the selected results can be compared. Otherwise, an error message is displayed.

When you click the **Compare** button, the VTune Profiler opens a new result tab with the performance data for Result 1 and Result 2 side-by-side and their calculated delta.

See Also

[Comparing Results](#)

[Bottom-up Comparison](#)

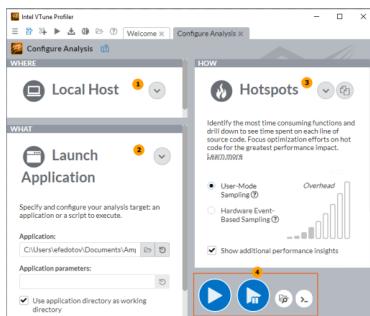
[Comparison Summary](#)

[Comparing Source Code](#)

Window: Configure Analysis

*Configure your performance analysis with the Intel VTune Profiler by specifying **WHAT** you need to profile, a target system **WHERE** you need to run the collection, and select an analysis type to define **HOW** you need to analyze your workload.*

As soon as you created a [project](#) for analysis, the VTune Profiler opens this window that navigates you through the analysis configuration with the following panes:

**1****WHERE:** Choose and set up a [system](#) for analysis.**2****WHAT:** Choose and configure your analysis [target](#).**3****HOW:** Choose and configure performance [analysis type](#).**4**

Run and control your analysis using these toolbar buttons:



starts the analysis;



pauses the data collection at any time of the app execution;

enables you to specify binary and source files for successful post-processing [finalization](#) (for example, for remote analysis);creates a [command line version](#) of the selected configuration that can be copied and used on other systems.

Window: Core Wake-ups - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **Core Wake-ups** sub-tab in the result tab.

Use the **Core Wake-ups** window to:

- Identify wake-up reasons on each core.
- Investigate wake-up reasons at a specific time during the collection.
- Sort data based on wake-up reason to identify common causes.
- View the objects that caused wake-ups.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

Core Wake-ups Pane

The Core Wake-ups pane displays information about events that caused the core to switch from a sleep state to an active state. This data is only collected when C-States data is collected. Display the information grouped by core or package, wake-up reason, wake-up object, and function stack using the **Grouping** drop-down. By default, the table is sorted by the **Total Wake-up Count** metric in the descending order providing objects with the highest wake-up count first.

A core is active when the core sleep state is C0 and is inactive, or sleeping, when the sleep state is Cn, where the higher the n value, the deeper the sleep state. The sleep states are displayed with a different prefix for package (PCn), module (MCn), or core (CCn). In the example below, the first Kernel Timer has a **Core Sleep State** value of CC6, which means the core was in the deepest sleep state.

NOTE

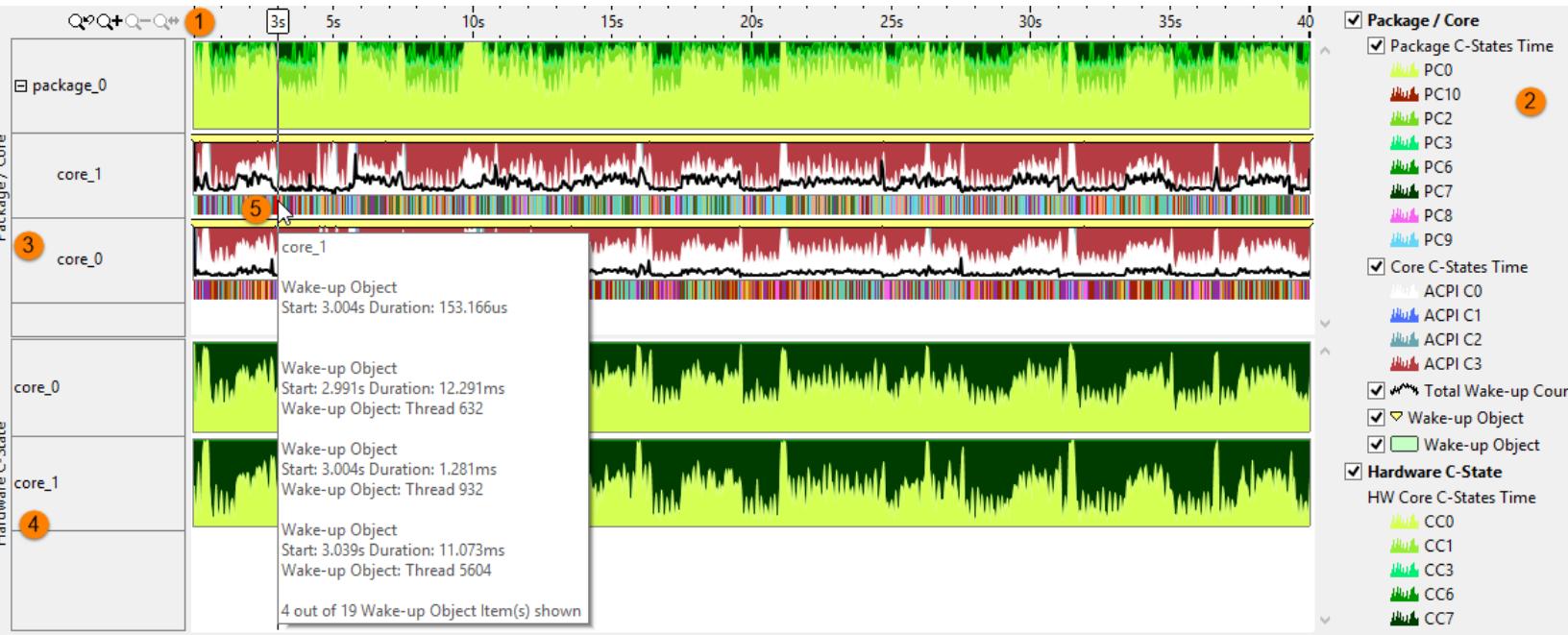
Additional details about the wake-up objects, such as Process Name or ThreadID, are available for results collected on a Linux* or Android* system only.

Grouping: Wake-up Reason / Wake-up Object / Function Stack							
Wake-up Reason / Wake-up Object / Function Stack	Total Wake-up Count	Core Sleep State	Process Name	ProcessID	ThreadID	Module Name	
IPI	1,146						
Timer	449						
Kernel Timer	258	CC6	swapper	0	0		
Kernel Timer	151	CC1	swapper	0	0		
User Timer	17	CC1	irq/144-ATML100	2464	2464		
User Timer	9	CC1	mmcqd/0	1577	1577		
User Timer	3	CC6	watchdogd	2494	2494		
User Timer	1	CC6	crashlogd	2780	2780		
User Timer	1	CC6	netd	2760	2760		
User Timer	1	CC6	AudioTrack	3069	3069		

Wake-up Reason	Description
CLK	Clock interrupt
DPC	Deferred procedure call
INT	Hardware interrupt
IPI	Inter-processor interrupt
IRQ	Interrupt request (Android*)
RDY	Ready event
Scheduler	Scheduler event
Timer	Timer event
Unknown	The operating system did not log a wake-up reason between exiting idle and re-entering idle or the wake-up reason was not passed to Intel VTune Profiler.

Timeline Pane

The Timeline pane shows the time spent in the active state (C0) or the various sleep states (Cn) as well as the total wake-up count for the package, package cores, and hardware cores. Use the Core Wake-ups pane to filter the wake-up types shown in the timeline by right-clicking a wake-up reason and selecting **Filter In by Selection**. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

**1****Toolbar**

Navigation control to zoom in/out on the view on areas of interest. For more details on the Timeline control, see [Managing Timeline View](#).

2**Legend**

Types of data presented on the timeline. Filter in/out any type of data presented on the timeline by selecting/deselecting corresponding check boxes. For example, each state is a different color and you may only be interested in the time spent in the active state. You can also filter in and out the hardware or package/core data.

The **Wake-up Object** marker shows processor wake-ups on the timeline. Hover over a yellow marker to see the time when the sleep state changed to an active state and the name of the wake-up object. Zoom in on the timeline to view individual markers if they are not visible when viewing the timeline for the full collection time.

3**Package/Module/Core C-states**

Graphical representation of the sleep states in each core and in the overall package. Each state is a different color, which can be filtered using the legend. Hover over the band to view the total wake-up count. Click the



to expand the package and view the individual modules and cores.

4**Hardware C-states**

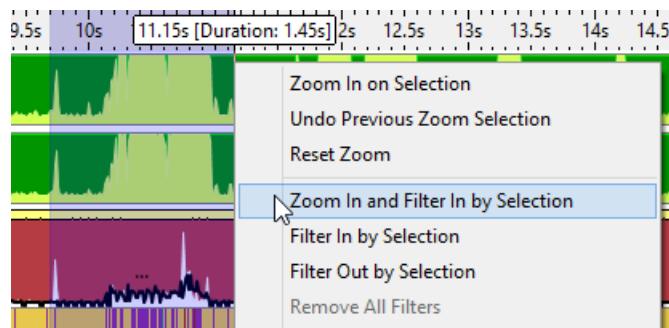
Graphical representation of the sleep states on the hardware. Each state is a different color, which can be filtered using the legend.

5

Wake up Band

Represents the wake-up objects that caused the core to switch from a sleep state to an active state. Each wake-up object type uses a unique color. By hovering over the band, you can view all of the wake-up objects at that point in time, including details such as wake-up object type, start time, and duration.

Find an area of interest in the timeline, such as a time when the core was active for a period of time, and then select the **Zoom In and Filter In by Selection** action to view the reasons the core became active. You can view the wake-up reasons and additional details for the time selected in the Core Wake-ups pane.

**See Also**

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

Window: Correlate Metrics - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **Correlate Metrics** sub-tab in the result tab.

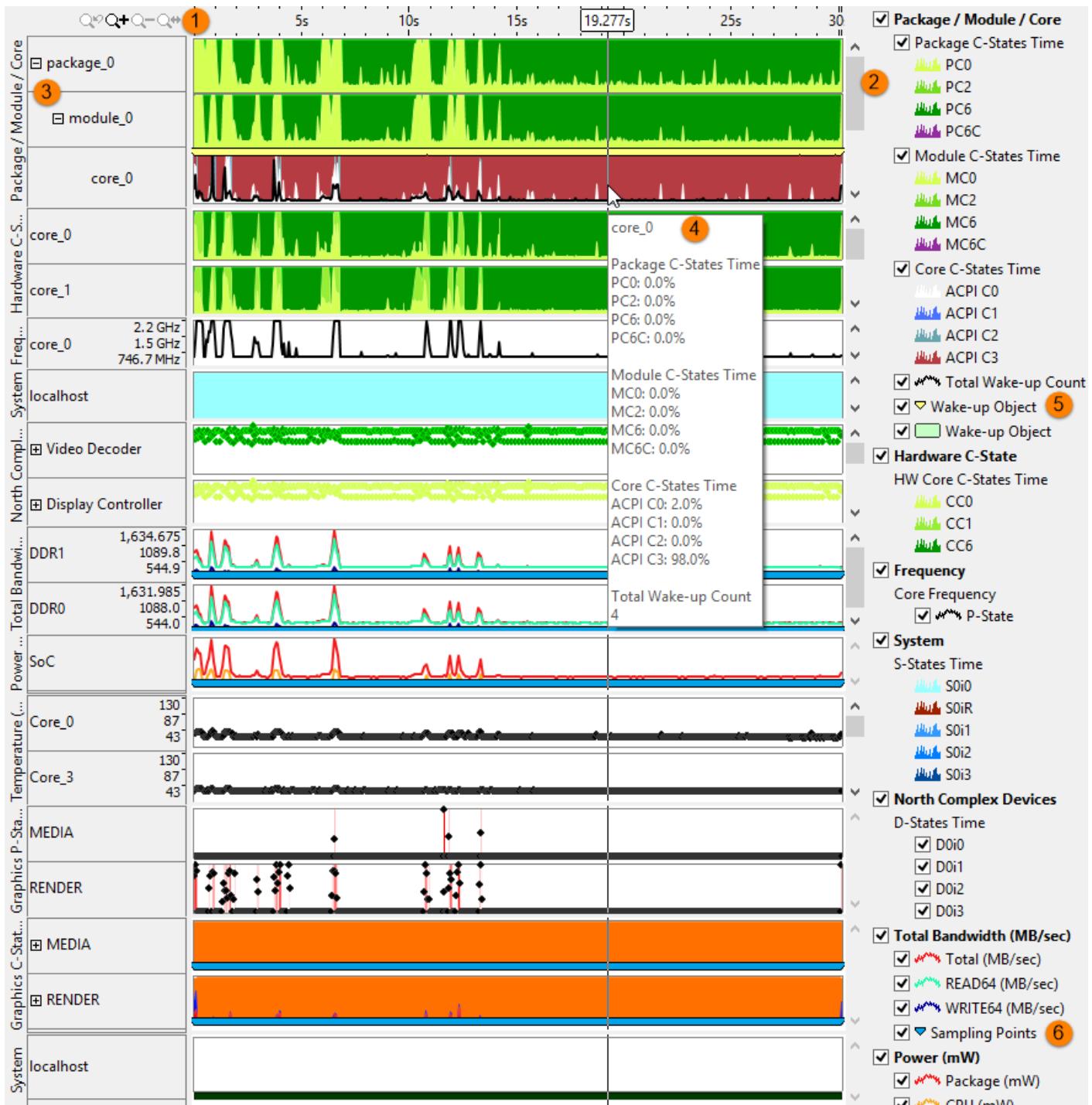
Use the **Correlate Metrics** window to:

- Assess energy-related metrics across the platform.
- View timeline data aggregated from all tabs in the Platform Power Analysis viewpoint.
- Identify trends that impacted energy usage during the collection period.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

The timelines in the **Correlate Metrics** window can also be found in other sub-tabs with the Platform Power Analysis result tab. The **Correlate Metrics** window is a good starting point if you are interested in identifying areas of energy inefficiency.

**Toolbar**

Navigation control to zoom in/out on the view on areas of interest. For more details on the Timeline control, see [Managing Timeline View](#).

Legend

Types of data presented on the timeline. Filter in/out any type of data presented on the timeline by selecting/deselecting corresponding check boxes. For example, to remove the timeline for the North Complex Devices from the view, uncheck the **North Complex Devices** checkbox.

3**Expandable Rows**

Click the

/



to expand the data and view metrics for individual cores or devices.

4**Tooltips**

Hover over the individual timelines to see data specific to that metric at that point during the collection. In the example, the C-States and Wake-up Counts for the Packages, Modules, and Cores are shown.

5**Wake-up Objects**

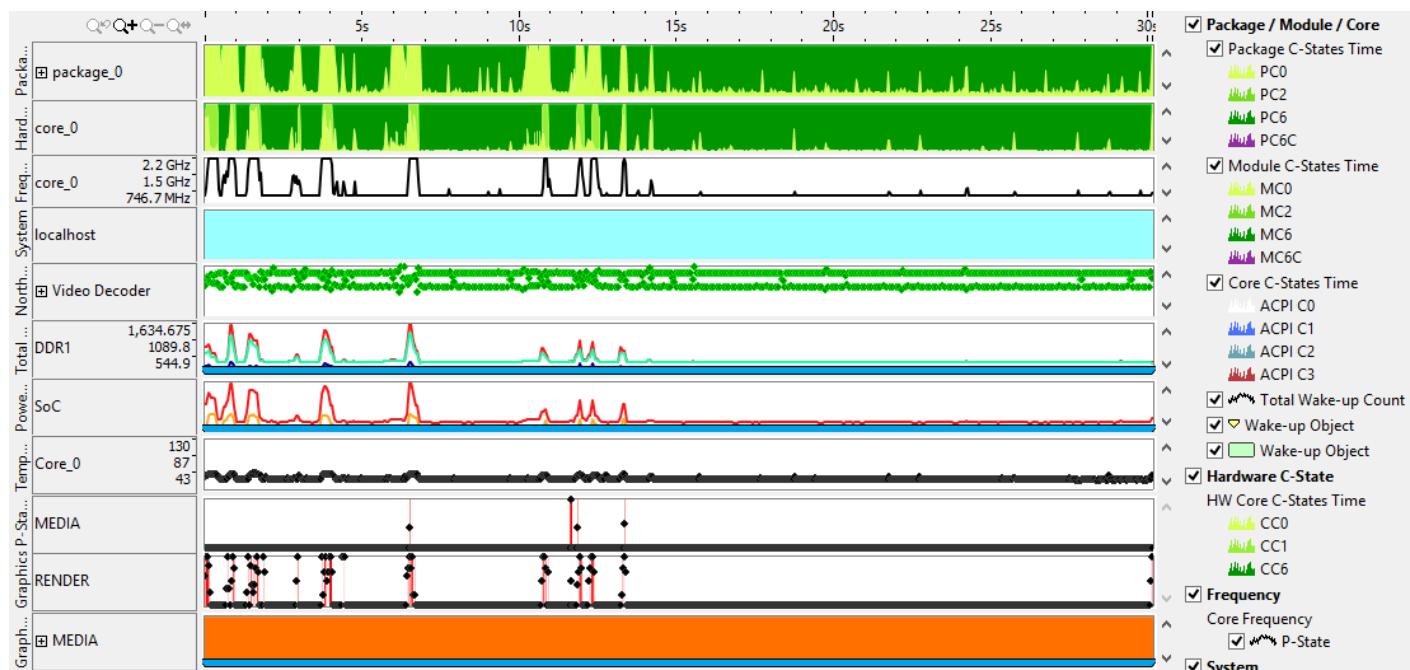
Processor wake-ups on the timeline. Hover over a yellow marker to see the time when the sleep state change happened and the name of the wake-up object. Zoom in on the timeline to view individual wake-up markers.

6**Sampling Points**

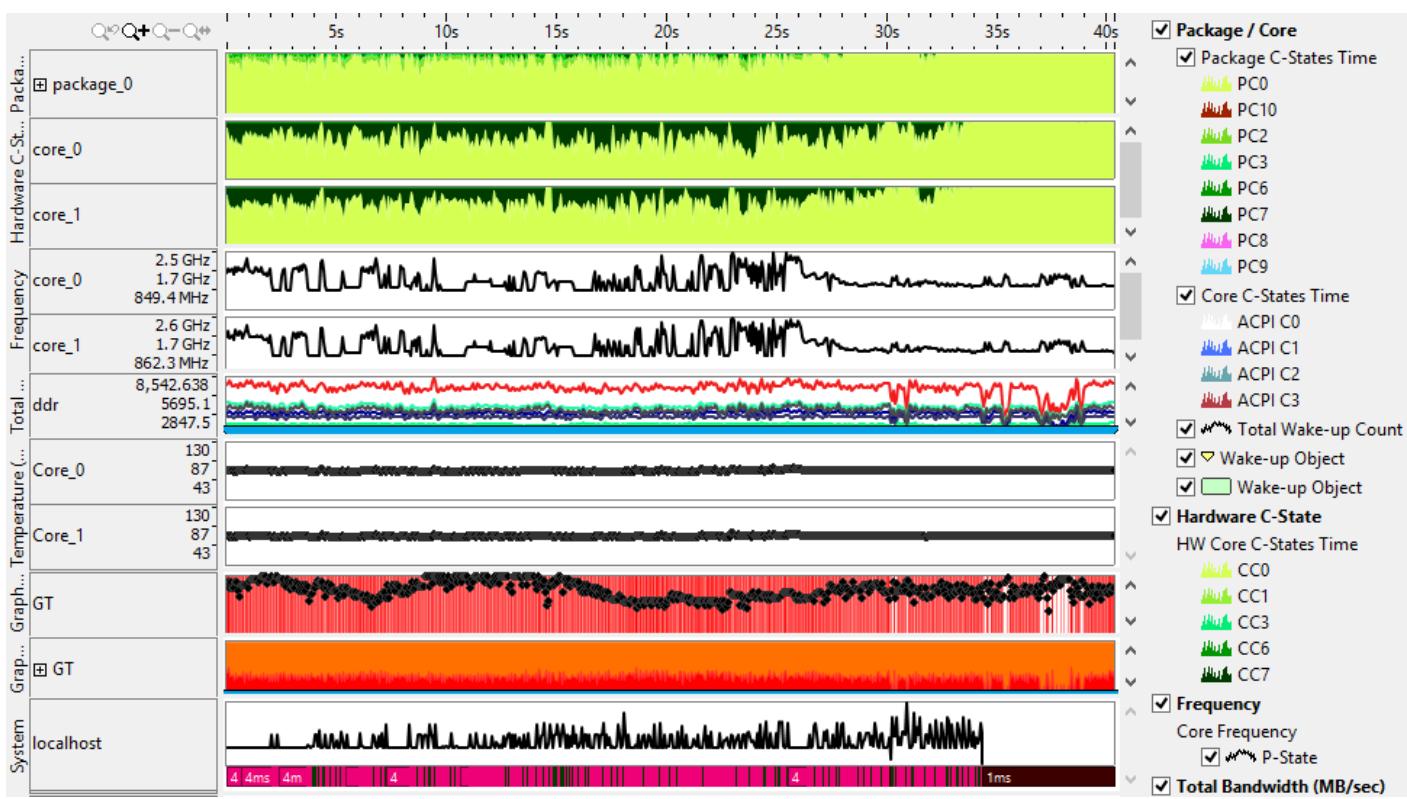
The point at which the sample was obtained from the hardware. The duration between sampling points is the sampling interval, which was specified during collection. Hover over a blue marker to see the time when the sample was obtained. Zoom in on the timeline to view individual sampling point markers and the time they occurred.

Examples

In the first example, the CPU starts in the active state and then drops into one of the deeper sleep states. The spikes in the CPU activity correspond to spikes in other timelines, such as the temperature and SoC power consumption. By viewing all data on one tab, you can identify trends and associations between metrics. To view each metric in more detail, visit the metric-specific tab.



In the second example, the CPU spends most time in the active state, and the similar activity levels for the Core C-States and Frequency indicates balance in the distribution of that activity.



See Also

[Pane: Timeline](#)

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Managing Timeline View](#)

Window: CPU C/P States - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **CPU C/P States** sub-tab in the result tab.

Use the **CPU C/P States** window to:

- Analyze the amount of time spent in each sleep state (C-State) and processor frequency (P-State).
- Identify which core spent time at what frequency.
- Understand which cores were active during which timeframes during data collection.
- Review the state residency by core, module, or package.
- Explore how the state and frequency changed over time.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

CPU C/P States Pane

The CPU C/P States pane shows the time spent in sleep states (C-States) and at each processor frequency (P-State). Intel SoC Watch can collect sleep states as requested by the OS (ACPI C-States) as well as the actual states used at the hardware level on a Windows* system. The data can be displayed per core or per package using the **Grouping** drop-down. Click the expand



/collapse



buttons in the data columns to expand or hide the columns of data for ACPI C-States, hardware C-State, and P-States. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit.

For example, if you are analyzing an idle scenario, you would use this report to see if most of the collection time was spent in the deepest possible sleep state. The time spent in CPU states is shown in the **Core C-States Time by Core Sleep State** columns (CC0-CCn for cores, MC0-MCn for modules, and PC0-PCn for packages). C0 represents the active state and Cn represents a sleep state, where the larger the number, the deeper the sleep state. Spending more time in deeper sleep states (C1-Cn) provides greater power savings. In the example below, both cores spent the most time in the deepest CPU sleep state C7, which corresponds to the OS request for the deepest sleep state ACPI C3. This is the desired result when the system being tested is idle. Expand the columns under **P-State by Core Frequency** to read the full values for the processor frequencies. Time in 0GHz indicates the time the processor was not running (total time in sleep states).

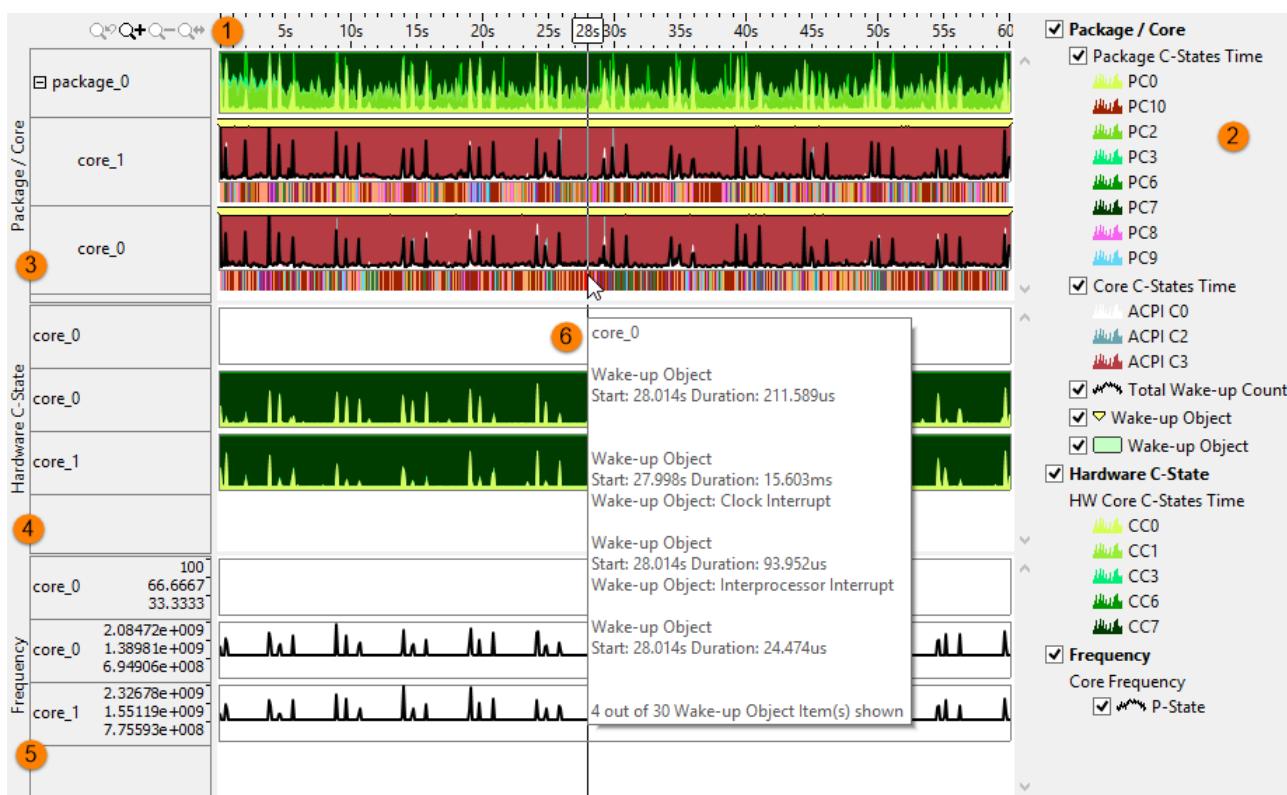
Core C-States Time by Core Sleep State						HW Core C-States Time by Core Sleep State						Core P-States Time by Frequency Type											
ACPI C0	ACPI C2	ACPI C3	CC0	CC1	CC3	CC6	CC7	0GHz	0.8G ...	0.9G ...	1.3G ...	1.5G ...	1.6G ...	1.7G ...	1.8G ...	1.9G ...	2GHz	2.1G ...	2.2G ...	2.3G ...	2.4G ...	2.5G ...	
2.409s	0.220s	57.421s	2.578s	0.399s	0.037s	0s	57.036s	57.474s	1.703s	0.002s	0.054s	0s	0s	0.002s	0.003s	0.004s	0.003s	0.027s	0.010s	0.006s	0.022s	0.059s	
2.908s	0.196s	56.946s	3.077s	0.410s	0.022s	0s	56.541s	56.973s	2.163s	0.002s	0.036s	0.004s	0.010s	0s	0.010s	0s	0.001s	0.015s	0.032s	0s	0.018s	0.071s	

Right-click in a column and select **Show Data As > Percent** to view the data in that column as a percent of the total time rather than in seconds. If the core spent a higher than expected percentage of time in an unexpected state, you can look at the timeline to identify when the core was in that state and then switch to the **Core Wake-ups** window to identify reasons for the change in state.

Core	Core C-States Time by Core Sleep State				HW Core C-States Time by Core Sleep State				Core P-States Time													
	ACPI C0	ACPI C2	ACPI C3	CC0	CC1	CC3	CC6	CC7	0GHz	0.8G ...	0.9G ...	1.3G ...	1.5G ...	1.6G ...	1.7G ...	1.8G ...	1.9G ...	2GHz	2.1G ...	2.2G ...	2.3G ...	2.4G ...
core_1	4.0%	0.4%	95.6%	4.3%	0.7%	0.1%	0.0%	95.0%										60.043s				
core_0	4.8%	0.3%	94.8%	5.1%	0.7%	0.0%	0.0%	94.2%										60.043s				

Timeline Pane

The Timeline pane graphically displays the C-States of each core, at each point in time. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

**1****Toolbar**

Navigation control to zoom in/out on the view on areas of interest. For more details on the Timeline control, see [Managing Timeline View](#).

2**Legend**

Types of data presented on the timeline. Filter in/out any type of data presented on the timeline by selecting/deselecting corresponding check boxes. For example, each state is a different color and you may only be interested in the time spent in the active state. You can also filter in and out the hardware or package/core data if you are only interested in frequency metrics.

The **Wake-up Object** marker shows processor wake-ups on the timeline. Hover over a yellow marker to see the time when the sleep state change happened and the name of the wake-up object.

3**Package/Core C-states**

Graphical representation of the sleep states in each core and in the overall package. Each state is a different color, which can be filtered using the legend. Hover over the band to view the total wake-up count. Click the



to expand the package and view the individual cores.

4**Hardware C-states**

Graphical representation of the sleep states on the hardware. Each state is a different color, which can be filtered using the legend.

**Frequency (by core)**

Core frequency values at each point during the collection. Hover over the frequency P-State line to view a tooltip listing the frequency at each time point.

**Wake up Band**

Represents the wake-up objects that caused the core to switch from a sleep state to an active state. Each wake-up object type uses a unique color. By hovering over the band, you can view all of the wake-up objects at that point in time, including details such as wake-up object type, start time, and duration.

See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

[Energy Analysis Metrics](#)

[C-State](#)

[P-State](#)

Window: Debug

By default, during data collection, all application output and collector event log displays in a separate console window. To change the output window for the standalone GUI menu



, go to **Options... > Intel VTune Profiler version > General** pane.

By default, the **Debug** window appears at the bottom of the view.

To choose what output to view, select an output source from the **Show output from** drop-down list.

See Also

[Pane: General](#)

Window: Event Count - Hardware Events

*Use the **Event Count** window to analyze the event count for PMU (Performance Monitoring Unit) events.*

Depending on the analysis type or viewpoint, the **Event Count** window may include the following panes:

- [Event Count pane](#)
- [Timeline pane](#)
- [Call Stack pane](#)

Event Count Pane

The Event Count pane attributes the Hardware Event Count by Hardware Event Type to program units. The **Hardware Event Count** metric estimates the number of times an event occurred during the collection.

By default, the data in the grid is sorted by the Clockticks event.

Microarchitecture Exploration			
Hardware Events			
Analysis Configuration Collection Log Summary Event Count Sample Count Caller/Callee Top-down			
Grouping: Function / Call Stack			
Function / Call Stack	INST_RETIRED ANY	CPU_CLK_UNHALTED	CPU_CLK_UNHALTED_REF
> multiply()	68,491,560,000	642,196,500,000	568,393,500,000
> [outside any known module]	23,000,000	1,303,500,000	1,188,000,000
> read_page()	14,500,000	396,000,000	429,000,000
> account_update_page_segments	12,000,000	0	0
> update_cfs_group	9,500,000	16,500,000	33,000,000
> get_page_from_free_list	33,000,000	33,000,000	0
> account_entry_dequeue	16,500,000	16,500,000	0
> wp_page_copy	16,500,000	0	0

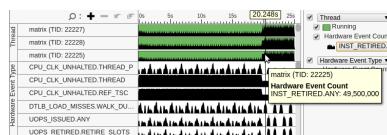
The list of hardware events depends on the analysis type. You may right-click an event column and select the **What's This Column** context menu option to open the description of the selected event.

When you explore the hardware events statistics for a result, you may drag and drop the columns in the grid for your convenience. VTune Profiler automatically saves your preferences and keeps the columns order for subsequent result views.

Timeline Pane

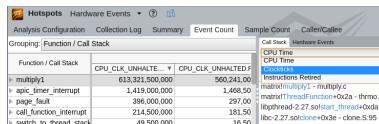
The **Timeline** pane is synchronized with the **Event Count** pane. The **Thread** area of the **Timeline** pane shows the number of times the selected event (CPU_CLK_UNHALTED.REF_TSC in the example below) occurred while a thread was running. You may use the **Hardware Event Count** drop-down menu in the legend area to choose a different event.

The **Hardware Event Type** area shows the application-level performance per each event.



Call Stack Pane

If you selected the [Collect stacks option](#) for the hardware event-based sampling analysis (for example, Hotspots), the VTune Profiler provides the [Call Stack pane](#). Use this pane to navigate between stacks and analyze the distribution of the event count for the object selected in the **Event Count** pane. For the example below, you select the Clockticks event to see stacks leading to the `multiply1` function and contributing to the Clockticks event count. You can use this data to identify the most performance-critical stacks with the highest contribution to the object's Clockticks event count.



See Also

[Intel Processor Events Reference](#)

[Window: Summary - Hardware Events](#)

[Switch Viewpoints](#)

[Hardware Events Report](#)

from command line

Window: Flame Graph

*Use the **Flame Graph** window to find the hottest code paths in your application.*

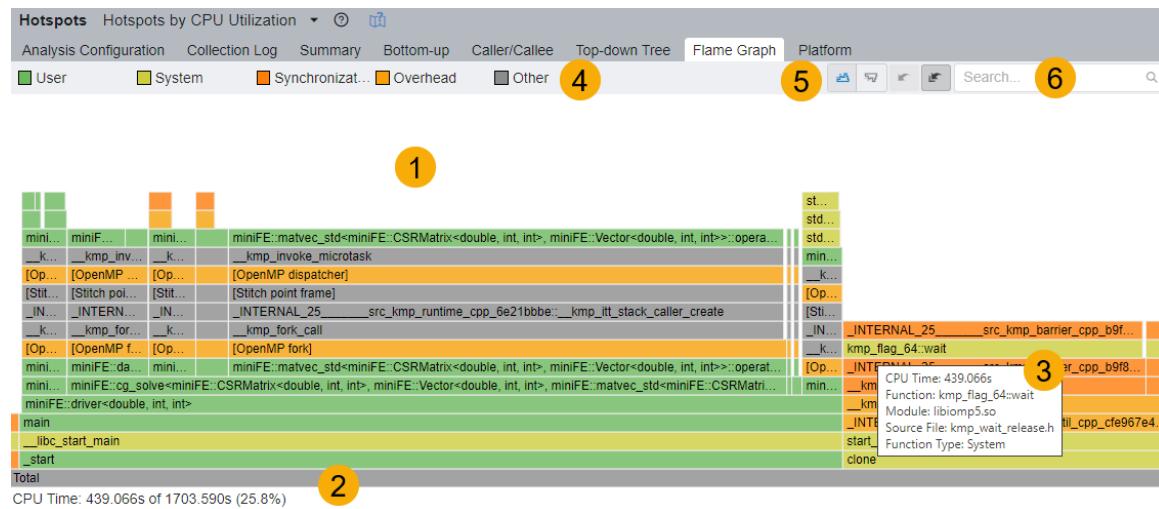
A flame graph is a visual representation of the stacks and stack frames in your application. The graph plots all of the functions in your application on the X-axis and displays the stack depth on the Y-axis. Functions are stacked in order of ancestry, with parent functions directly below child functions. The width of a function displayed in the graph is an indication of the amount of time it engaged the CPU. Therefore, the hottest functions in your application occupy the widest portions of the flame graph.

Access the Flame Graph Window

- Run the [Hotspots analysis](#) on your application. Ensure that you are collecting data with call stacks.

- a. If you are running the analysis in **User-Mode Sampling** mode, the option to collect CPU sampling data with stacks is enabled by default (see **Details**).
 - b. If you are running the analysis in **Hardware Event-Based Sampling** mode, check the **Collect Stacks** option.
2. When the analysis is complete and results display, switch to the **Flame Graph** tab. You can also click on the Flame Graph link in the **Insight** section of the **Summary** window.

Elements of the Flame Graph Window



Flame Graph Area:

1

This section displays stacks and stack frames for your application. Every box in the graph represents a stack frame with the complete function name. The horizontal axis shows the stack profile population, sorted alphabetically. The vertical axis shows the stack depth, starting from zero at the bottom.

The flame graph does not display data over time. The width of each box in the graph indicates the percentage of the function CPU time to total CPU time. The total function time includes processing times of the function and all of its children (*callees*).

The flame graph is a graphical representation of the data contained in the tabular **Top-Down** view.

- **Zoom/Select Action:** To learn more about a function, click on a box to zoom in horizontally. You will then see any child functions it contains. Ancestor frames (below the selected box) display in faded colors because their width is only partially visible. Changes in stack pane data reflect any zoom or selection action you take in the flame graph area.
- **Filter toolbar:** The flame graph responds to changes to the Global Filter setting in the Filter toolbar. Use this toolbar to filter data in the following ways:
 - Process
 - Thread
 - Module
 - Function Type
 - Time

- **Function colors:** The flame graph uses a color scheme to identify these function types:
 - **User:** A function from the application module of the user
 - **System:** A function from the System or Kernel module
 - **Synchronization:** A synchronization function from the Threading Library (like OpenMP Barrier)
 - **Overhead:** An overhead function from the Threading library (like OpenMP Fork or OpenMP Dispatcher)

2

Details Area:

Hover over a flame graph element to get CPU Time as well as the percentage of Total Time taken by the selected stack-frame.

3

Tooltips:

When you hover over a flame graph element, a tool tip displays these details for the selected bar or stack frame:

- CPU Time
- Function name
- Module name
- Source file
- Function type

4

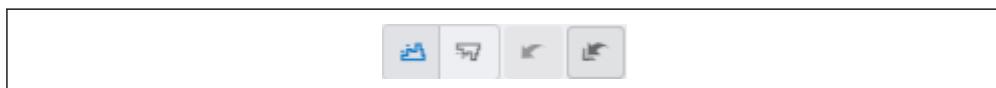
Legend:

The legend describes the types of functions included in the flame graph.

5

Navigation Bar:

Use these controls in the navigation bar to manage the flame graph display:



- : Select the **Flame Graph** mode.
- : Select the **Icicle Graph** mode. This inverts the flame graph display.
- : Undo the last zoom action.
- : Restore the flame graph to its original view.

6

Search:

Search for any functions in the flame graph. You can use regular expressions in the search string. When the results display, the CPU Time and percentage of Total Time include the times for all of the matched functions.



Analyze Flame Graph Data

Use these tips to analyze the application information contained in your flame graph:

- For hot code paths in your application, analyze the time spent on each function and its callees. The function bar displays as a fraction of CPU time.
- Choose between the **Flame Graph** and **Icicle Graph** visualizations to help with your analysis.
- Filter data through the Filter bar and/or Timeline.
- Optimize your application starting with the lowest function in the flame graph and working your way up.
- Pay close attention to the hottest user and synchronization functions. In the flame graph, they appear as the widest functions.
- Use the stack pane to dive into the source code of a function.

Related information

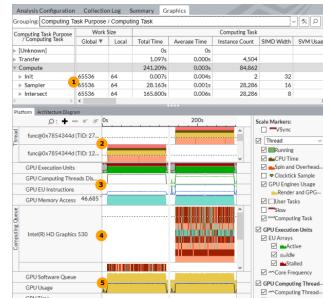
- An explanation of Flame Graphs
- Hotspots View
- Java Code Analysis

Window: Graphics - GPU Compute/Media Hotspots

Use this window for GPU analysis with Intel® VTune™ Profiler to identify GPU tasks with high GPU utilization and estimate the effectiveness of this utilization. This view is particularly useful for analysis of OpenCL™, SYCL, and Intel Media SDK applications doing substantial computation work on the GPU.

To access this window: Select the **GPU Compute/Media Hotspots** viewpoint and click the **Graphics** sub-tab in the result tab.

Along with the regular bottom-up analysis and stack data, the **Graphics** window correlates CPU / GPU busyness and displays the distribution of the GPU metrics over time:



1

Grid. Analyze basic performance metrics per program unit and identify the most time-consuming units. If your application uses the OpenCL software technology and you ran the analysis with the **Trace GPU Programming APIs** option enabled, the grid is grouped by **Computing Task Purpose** granularity by default.

Analyze and optimize hot kernels with the longest Total Time values first. These include kernels characterized by long Average Time values and kernels whose Average Time values are not long, but they are invoked more frequently than the others (see Instance Count values). Both groups deserve attention. For more details, see [GPU OpenCL™ Application Analysis](#).

To understand the CPU activity (which module/function was executed and its CPU time) while the GPU execution units were idle, queued, or busy executing some code, use the **GPU Render and EU Engine State** grouping level:

Analysis Configuration				Collection Log	Summary	Graphics	Platform	Bottleneck
Groupping: GPU Render and EU Engine State				Function / Call Stack				
CPU Render and EU Engine...				CPU Time	Module	Function/Full	Source File	
GPU Render and EU Engine...								
Idle	2.80s							
Execution	0.020s							
Queue	0.043s							
> sphere.intersect	0.001s	analyze_locks	sphere.intersect	analyze.cpp				
> grid.intersect	0.012s	analyze_locks	grid.intersect	grid.cpp				
> grid.bounds.intersect	0.002s	analyze_locks	grid.bounds..._intersect	grid.cpp				

2

Thread. Explore CPU and GPU utilization by a particular thread. The **Platform** tab displays the thread name as a name of the module where the thread function resides. For example, if you have a `myFoo` function that belongs to `MyMegaFoo` (Linux*) or `MyMegaFoo.dll` (Windows*) function, the thread name is displayed as `MyMegaFoo` (Linux*) or `MyMegaFoo.dll` (Windows*). This approach helps easily identify the location of the thread code producing the work displayed on the timeline.

Windows* targets only: Correlate CPU and GPU usage and estimate whether your application is GPU bound. GPU Engines Usage bars show DMA packets on CPU threads originating GPU tasks. The bars are colored according to the type of used GPU engine (yellow bars in the example above correspond to the Render and GPGPU engine).

3

GPU hardware metrics. If you enabled the **Analyze Processor Graphics hardware events** option for GPU analysis on the processors with the Intel® HD and Intel® Iris® Graphics, the VTune Profiler displays the statistics for the selected group of metrics over time.

For example, for the default **Overview** group of metrics, you may start with **GPU Execution Units: EU Array Idle** metric. Idle cycles are wasted cycles. No threads are scheduled and the EU's precious computational resources are not being utilized. If EU Array Idle is zero, the GPU is reasonably loaded and all EUs have threads scheduled on them.

In most cases the optimization strategy is to minimize the **EU Array Stalled** metric and maximize the **EU Array Active**. The exception is memory bandwidth-bound algorithms and workloads where optimization should strive to achieve a memory bandwidth close to the peak for the specific platform (rather than maximize **EU Array Active**).

Memory accesses are the most frequent reason for stalls. The importance of memory layout and carefully designed memory accesses cannot be overestimated. If the **EU Array Stalled** metric value is non-zero and correlates with the **GPU L3 Misses**, and if the algorithm is not memory bandwidth-bound, you should try to optimize memory accesses and layout.

Sampler accesses are expensive and can easily cause stalls. Sampler accesses are measured by the **Sampler Is Bottleneck** and **Sampler Busy** metrics.

NOTE

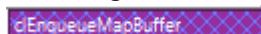
To analyze Intel Graphics hardware events on a GPU, make sure to [set up your system for GPU analysis](#).

4

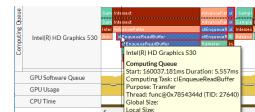
Computing Queue. Analyze details on OpenCL kernels submission, in particular distinguish the order of submission and execution, and identify the time spent in the queue, zoom in and explore the Computing Queue data. VTune Profiler displays kernels with the same name and global/local size in the same color. Synchronization tasks are marked with vertical hatching



. Data transfers are marked with cross-diagonal hatching



You can click a kernel task to highlight the whole queue to the execution displayed at the top layer. Hover over an object in the queue to see kernel execution parameters.



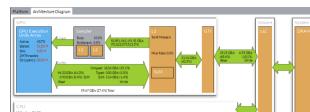
Windows targets only: Switch to the [Platform window](#) to explore how the execution path of the OpenCL device queue correlates to the DMA packets software queue.

5

GPU Usage metrics. GPU usage bars are colored according to the type of used GPU engine.

Theoretically, if the **Platform** tab shows that the GPU is busy most of the time and having small idle gaps between busy intervals and the GPU software queue is rarely decreased to zero, your application is GPU bound. If the gaps between busy intervals are big and the CPU is busy during these gaps, your application is CPU bound. But such obvious situations are rare and you need a detailed analysis to understand all dependencies. For example, an application may be mistakenly considered GPU bound when GPU engines usage is serialized (for example, when GPU engines responsible for video processing and for rendering are loaded in turns). In this case, an ineffective scheduling on the GPU results from the application code running on the CPU.

For further OpenCL kernel analysis, select a computing task you are interested in (for example, `AdvancedPaths`) and switch to the **Architecture Diagram** tab. VTune Profiler displays performance data per GPU hardware metrics for the time range when the selected kernel was executed:



Flagged values signal a performance issue. In this example, ~50% of the GPU time was spent in stalls. This means that performance is limited by memory or sampler accesses.

See Also

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

[Intel® Media SDK Program Analysis](#)

[Pane: Timeline](#)

Window: Graphics C/P States - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **Graphics C/P States** sub-tab in the result tab.

Use the **Graphics C/P States** window to:

- Review the GPU sleep states (C-state) and processor frequency (P-state) by device.
- Analyze GPU sleep states and processor frequency changes over time.

- Identify which device spent time at a particular frequency.
- View which devices were active at a specific time.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

Graphics C/P States Pane

Grouping: Device Name

Device Name	Graphics P-States Time by Core Frequency		Graphics C-States Time by Graphics Device Sleep State	
	0GHz	0.45GHz	RC0	RC6
GT	32.643s	7.391s	7.391s	32.923s
Highlighted 0 row(s):				

Shows the time spent in each state or frequency, organized by device. Click the expand



/collapse

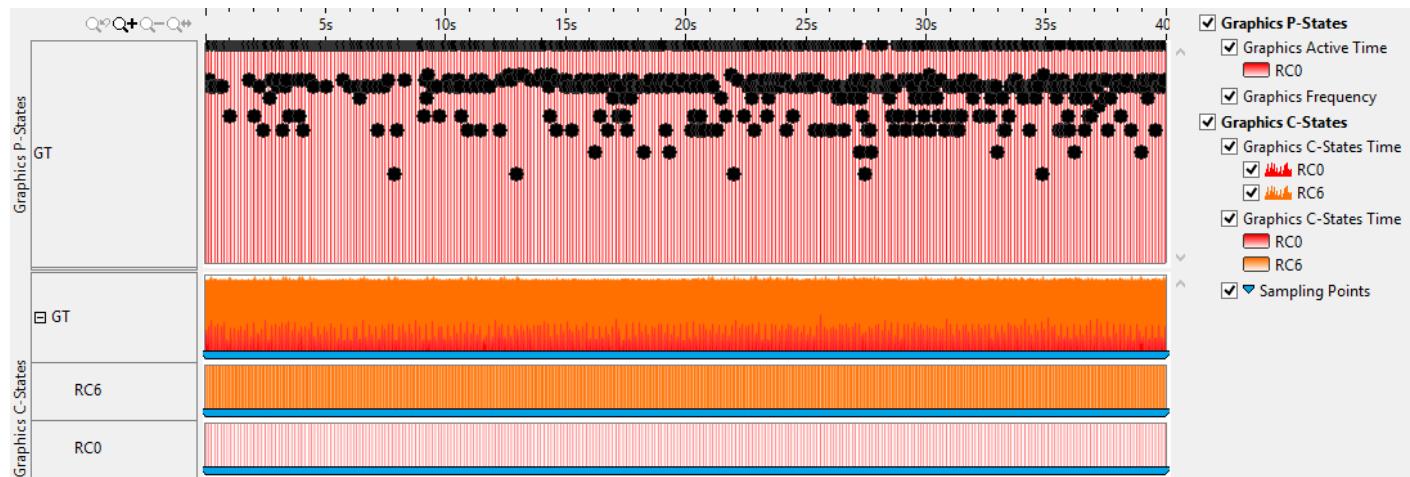


buttons in the data columns to expand the column and show data for different C-States and P-States in each device. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit. For example, select **Show Data As > Time and Bar** to view a visual representation of the percent of collection time spent in each state.

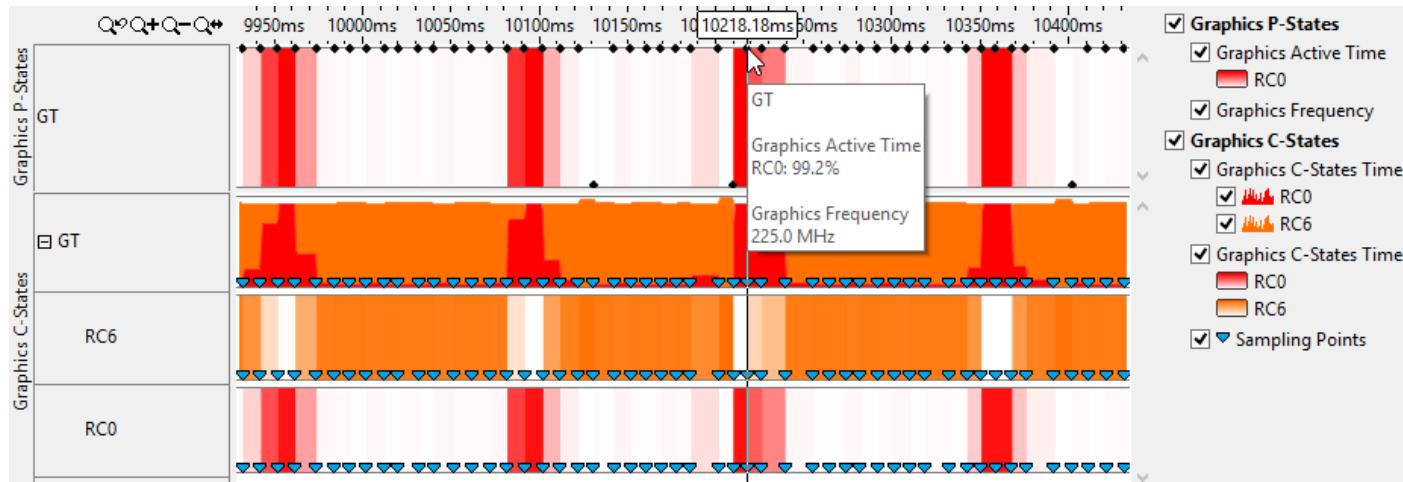
Device Name	Graphics P-States Time by Core Frequency		Graphics C-States Time by Graphics Device Sleep State	
	0GHz	0.45GHz	RC0	RC6
GT	32.643s	7.391s	40.314s	32.923s

Timeline Pane

Displays the C-states and P-states of each device at each point in time. The states are shown in a different color as identified by the legend to the right of the timeline. The frequency graph uses data points to indicate that the data has been read from the hardware at discrete sampling points instead of from a residency counter. Hover over a blue marker to see the time when the sampling point occurred.



Time spent in each state is represented by a heat map. The heat map data may not be visible when viewing the full timeline. Zoom in on a section of interest to view the heat map and details about the data points. The heat map, represented in the example below with shades of red in the **Graphics P-States** timeline, illustrates how active the device was since the previous sample. The deeper the red color, the longer it was in the active state. The exact transitions between active and idle are not known. Hover over a point to view the percentage of time in the active state. In the example below, the device was active for 99.2% of the time between the two sampling points and the color is the deepest shade of red. The bars with lighter shades indicate less time in the active state.



Use the timeline to identify times when there was a higher frequency for a longer period of time and ensure that it matches expectations. If it does not, you can look at the **CPU C/P States** tab to show CPU activity at the same time. You can also view the **Bandwidth** tab to see if a similar spike in activity occurs in that tab. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

[Managing Timeline View](#)

Window: NC Device States - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **NC Device States** sub-tab in the result tab.

Use the **NC Device States** window to:

- Identify the time spent in D0ix states by each device.
- Analyze the trend of D0ix state residency over time.
- Review the percent of time a device spent in a particular D0ix state.

The North complex contains the compute intensive components (for example, video decode, image processing, and others). D0ix states are low-power states used on system on a chip (SoC) platforms.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

North Complex Device States Pane

Grouping: North Complex Devices

North Complex Devices	Device ID	Sample Counts by D-State			
		D0i0	D0i1	D0i2	D0i3
Video Decoder	Video Decoder	0	0	0	2,062
Display Controller	Display Controller	2,062	0	0	0
ISP	ISP	0	0	0	2,062
GUnit	GUnit	2,062	0	0	0
Render	Render	52	0	0	2,010
Media	Media	0	0	0	2,062
Display DPIO	Display DPIO	0	0	0	2,062
CMNLN	CMNLN	0	0	0	2,062
TX0	TX0	0	0	0	2,062
TX1	TX1	0	0	0	2,062
TX2	TX2	0	0	0	2,062
TX3	TX3	0	0	0	2,062
RX0	RX0	0	0	0	2,062
RX1	RX1	0	0	0	2,062

The North Complex Device States pane shows the list of devices in the North complex and displays the sample counts for each device. Click the expand



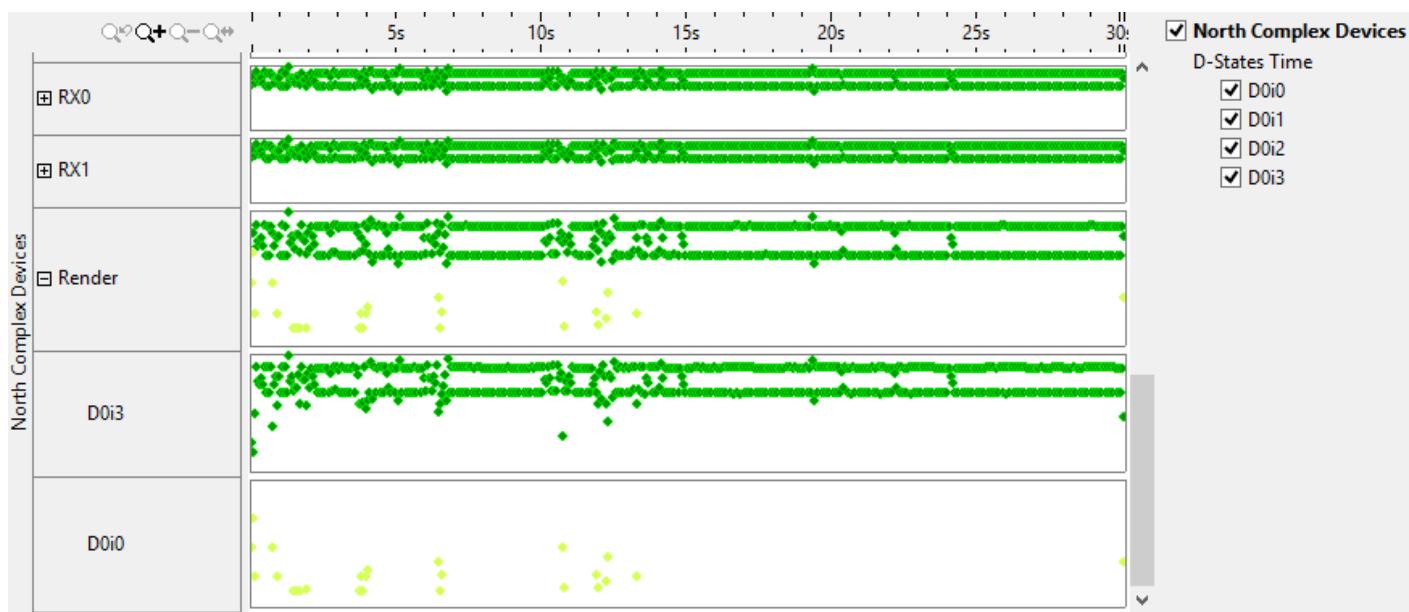
/collapse



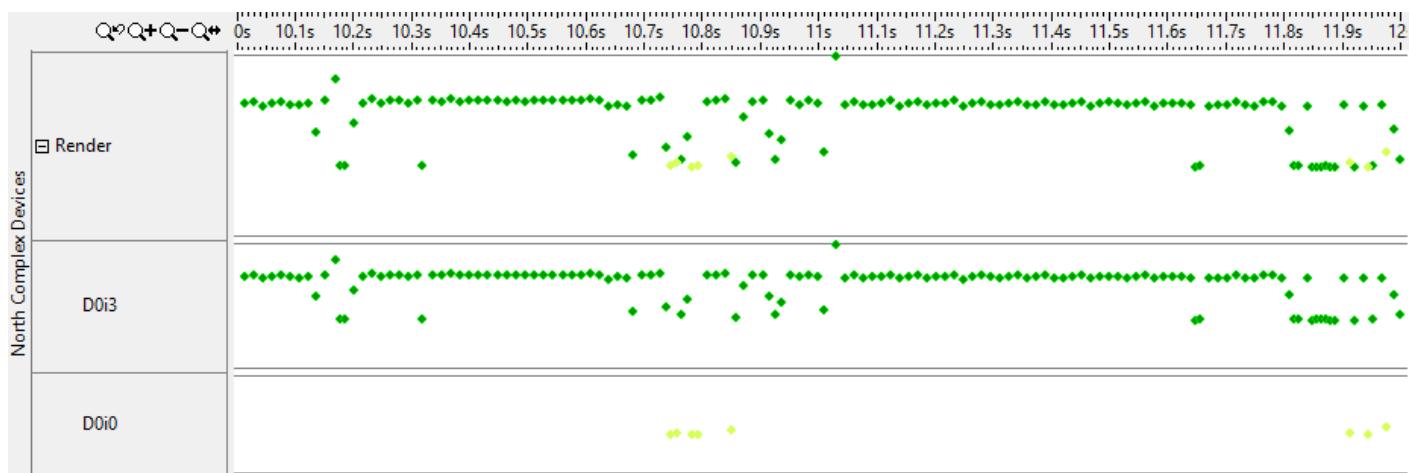
buttons in the data columns to expand the column and show data for different D0ix states in each device. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit. For example, you could select **Show Data As > Percent** to view the percent of collection time a particular device spent in the active state.

A device remaining in the active state (D0i0) can prevent the system from entering a deep sleep state (S0ix). Compare device time spent in the active state with **System Sleep States** or **Graphics C/P State**.

Timeline Pane



The Timeline pane displays the D0ix states of each device at each point when the data was read. Each state is shown in a different color. Use the legend on the right to add or remove D0ix states from the timeline. Hover over a data point to see the percentage of time spent in each state. Zoom in or out on the timeline to view trends in more detail. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab. The following example shows a zoomed-in view of the result above to show individual data points.



See Also

[Interpreting Energy Analysis Data](#)
[Viewing Energy Analysis Data](#)
[Viewpoint](#)

[Grouping Data](#)
[Managing Timeline View](#)

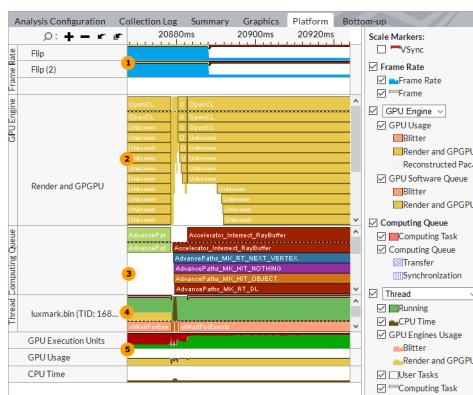
Window: Platform

To access this window: Click the **Platform** sub-tab in the result tab.

Depending on the metrics collected during the analysis, use the **Platform** window to:

- Inspect CPU and GPU utilization, frame rate and memory bandwidth.
- Explore your application performance for user tasks such as Intel ITT API tasks, Ftrace*/Systrace* event tasks, SYCL and OpenCL™ API tasks, and so on.
- Correlate CPU and GPU activity and identify whether your application/some phases of it are GPU or CPU bound.
- Analyze CPU/GPU interactions and software queue for GPU engines at each moment of time.

The **Platform** window represents a distribution of the performance data over time. For example, on Linux the **Platform** window displays the following data:

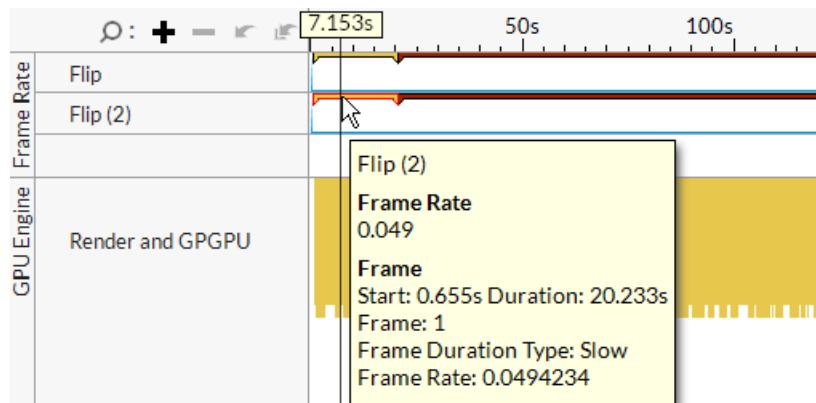


1

Frame Rate. Identify bounds for GPU and CPU frames (Windows only), where:

- *CPU Frame X (Present)* is the time range between the moment frame X-1 is queued for presentation and the moment frame X is queued for presentation.
- *GPU Frame X (Flip)* is the time range between the moment frame X-1 is rendered on the screen and the moment frame X is rendered on the screen.

Hover over a frame object to view a summary including data on frame duration, frame rate, and others:



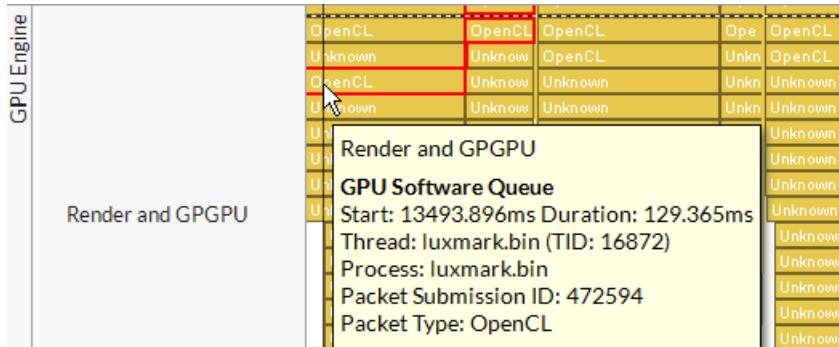
CPU and GPU frames with the same ID are displayed in the same color.

2

GPU Engine. Explore overall GPU utilization per GPU engine at each moment of time. By default, the **Platform** window displays GPU Utilization and software queues per GPU engine. Hover over an object executed on the GPU (in yellow) to view a short summary on GPU utilization, where *GPU Utilization* is the time when a GPU engine was executing a

workload. You can explore the top GPU Utilization band in the chart to estimate the percentage of GPU engine utilization (yellow areas vs. white spaces) and options to submit additional work to the hardware.

To view and analyze GPU software queues, select an object (packet) in the queue and the VTune Profiler highlights the corresponding software queue bounds:



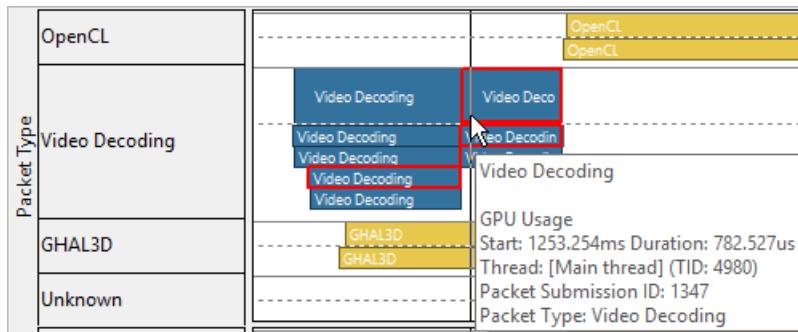
Full software queue prevents packet submissions and causes waits on a CPU side in the user-mode driver until there is space in the queue. To check whether such a stall decreases your performance, you may decrease a workload on the hardware and switch to the **Graphics** window to see if there are less waits on the CPU in threads that spawn packets. Another option could be to additionally load the queue by tasks and see whether the queue length increases.

Each packet in the **Platform** window has its own ID that helps track its life cycle in a software queue. The ID does not correspond to the rendered frames. You may identify where a packet came from by the thread name (corresponding to the name of the module where a thread entry point resides) specified in the tooltip.

Horizontal hatching is used for data that may be not accurate due to collection issues (for example, missing event from the Intel® Graphics Driver). This type of data is identified as Reconstructed packets in the Legend.

Windows only:

For Windows targets, you may select the **Packet Type** drop-down menu option in the Legend area to explore GPU utilization and software queues per DMA packet domain:



Presents on Windows targets are displayed in a red hatch.

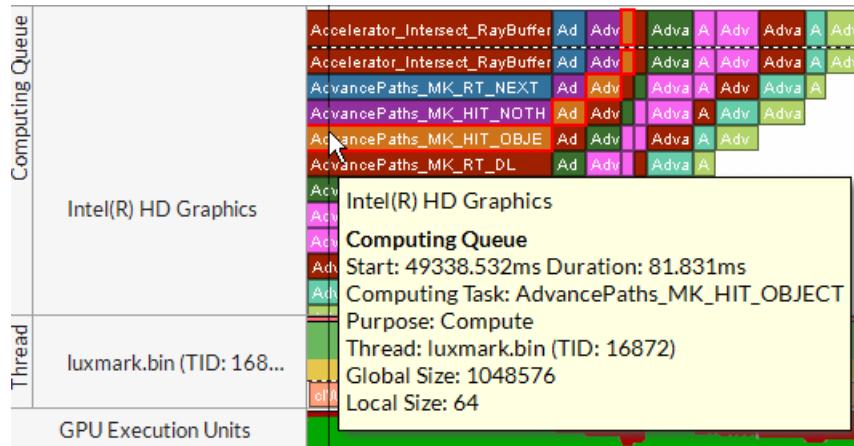
Computing Queue. Analyze details on OpenCL™ kernels submission, in particular distinguish the order of submission and execution, and identify the time spent in the queue, zoom in and explore the Computing Queue data. VTune Profiler displays kernels with the same name and global/local size in the same color. On Windows, synchronization tasks are marked with vertical hatching



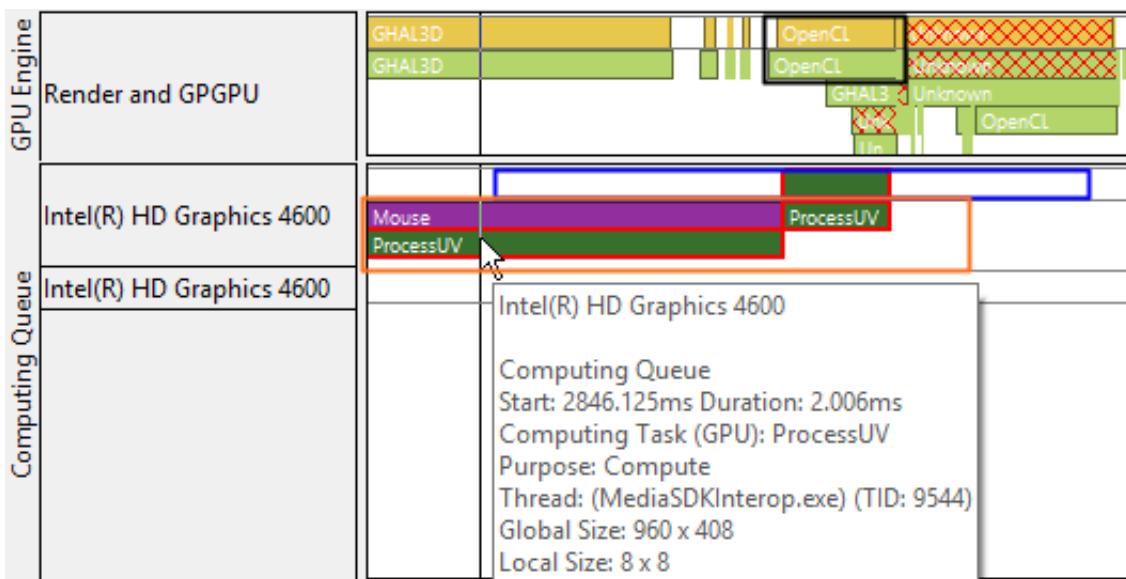
. Data transfers are marked with cross-diagonal hatching



You can click a kernel task to highlight the whole queue to the execution displayed at the top layer. Hover over an object in the queue to see kernel execution parameters.



Windows only:

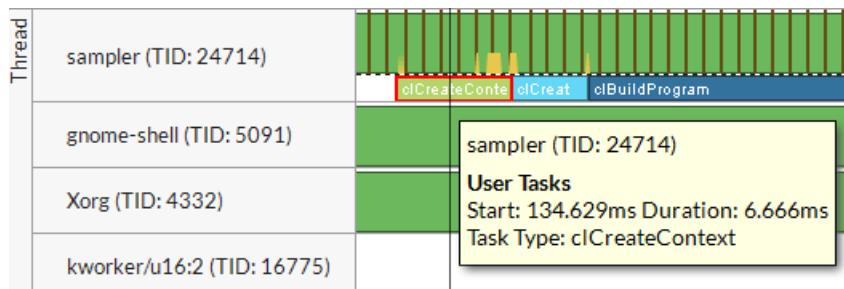


On Windows, you can explore how the execution path (marked in blue) of the OpenCL device queue (in orange) correlates with the DMA packets software queue (in black). The OpenCL kernel queue expedites kernels to the driver where DMA packets of different types are get multiplexed in the single DMA queue. In the example above, the **Render and GPGPU** queue serves both graphics (GHAL3D) and compute (OpenCL)-originated packets.

4

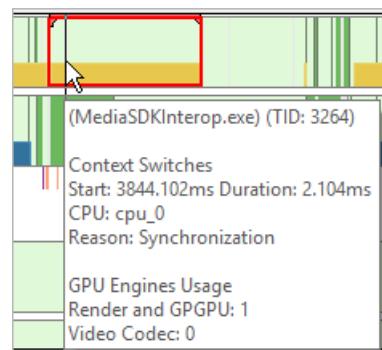
Thread. Explore CPU utilization by thread. The **Platform** window displays the thread name as a name of the module where the thread function resides. For example, if you have a `myFoo` function that belongs to `MyMegaFoo` function, the thread name is displayed as `MyMegaFoo`. This approach helps easily identify the location of the thread code producing the work displayed on the timeline.

If your code used the Task API to mark the tasks regions or you enabled any system tasks for monitoring specific events, the task objects show up on the timeline and you can hover over such an object for details:



Windows only:

Hover over a context switch area to see the details on its duration, reason, and affected CPU. Dark-green context switches show time slices when a thread was busy with a workload while light-green context switch objects show areas where a thread was waiting for a synchronization object. Gray areas show inactivity periods caused by preemption when the operating system task scheduler switched a thread off a processor to run another, higher-priority thread.



Correlate CPU and GPU utilization and estimate whether your application is CPU or GPU bound. GPU Engines utilization bars show DMA packets on CPU threads originating GPU tasks. The bars are colored according to the type of used GPU engine (yellow bars in the example below correspond to the Render and GPGPU engine). If the **GPU Engine** area of the **Platform** window shows aggregated GPU utilization for all threads and processes in the system, the GPU Engines Utilization bars in the **Thread** area show GPU engine utilization by a particular thread.

5

GPU Metrics. Correlate the data on GPU activity per [GPU metrics](#) with the CPU utilization data. The **GPU Utilization** bars are colored according to the type of used GPU engine.

To analyze CPU and GPU utilization per thread, switch to the **Graphics** window.

NOTE

To analyze Intel HD Graphics and Intel® Iris® Graphics hardware events on a GPU, make sure to [set up your system](#) for GPU analysis.

6

Core Frequency. Explore the ratio between the actual and the nominal CPU frequencies. Values above 1.0 indicate that CPU is operating in a turbo boost mode.

NOTE

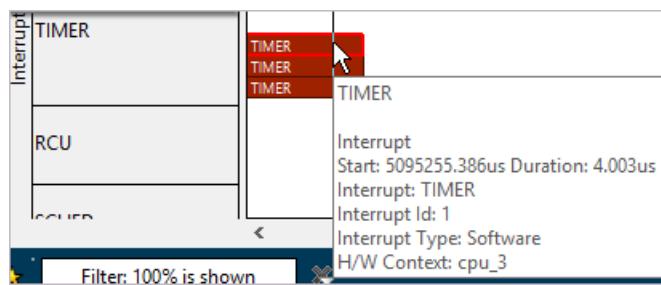
This data is available only for the hardware event-based sampling analysis results.

DRAM Bandwidth. Explore the application performance per Uncore to DRAM Bandwidth metrics over time.

NOTE

This data is available only for the hardware event-based sampling analysis results with the bandwidth events collection enabled.

Interrupt. Identify the intervals where system interrupts occurred. Hover over an interrupt object to view full details in the tooltip:

**NOTE**

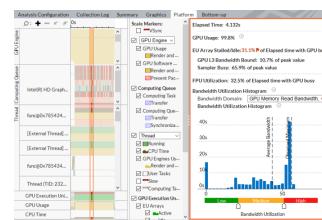
This type of data shows up for the [custom data collection](#) results if you enabled the corresponding [Ftrace events collection](#) during the analysis type configuration.

NOTE

To monitor general GPU utilization over time on Windows OS, run the VTune Profiler as an Administrator.

Platform Context Summary

Explore the **Context Summary** provided to the right of the Timeline pane in the Platform window. It displays the summary statistics for the context selected on the timeline. By default, the Context Summary shows data for the whole run. To narrow down the analysis, select an area of interest on the timeline, right-click and select **Filter In by Selection**:



The **EU Stalled/Idle** metric shows the time when execution units were stalled or idle. High values are flagged as a performance issue with a negative impact on the compute-bound applications.

See Also

[GPU Compute/Media Hotspots Analysis \(Preview\)](#)

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

[Task Analysis](#)

[Analyze Interrupts](#)

Window: Platform Power Analysis

Use the Platform Power Analysis viewpoint to review, visualize, and interpret power and energy data collected using Intel® SoC Watch.

Energy analysis data collected by Intel SoC Watch version 2.3 or later on an Android* or Linux* device can be imported into Intel® VTune™ Profiler and visualized with the Platform Power Analysis viewpoint. The **Summary** window is always present, but other windows within the viewpoint will vary depending on the metrics collected with Intel SoC Watch. For example, the **DDR Bandwidth** metrics are visualized on the **DDR Bandwidth** window. The metrics available to you will depend on your device hardware and operating system. Review the *Intel SoC Watch User's Guide* for your operating system for detailed information on each metric.

Collection and Visualization Method

Energy data is collected and visualized using the following mechanisms:

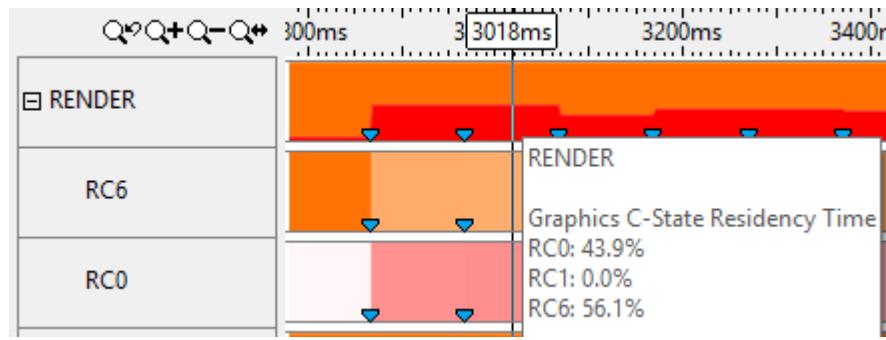
- Sampled Value Data: The value is gathered by sampling energy data over regular or irregular intervals. There can either be a set range of values (as with North Complex D0ix States) or the actual value can be measured at the sampling point (as with Thermal temperature). The values between sampling points are not known. The values are visualized with sampling points in the timeline pane.

For example, North Complex D0ix State values are sampled over regular intervals. At every sample point, a D-State value is returned at the time the sample was taken and is visualized with a set color in the timeline pane.



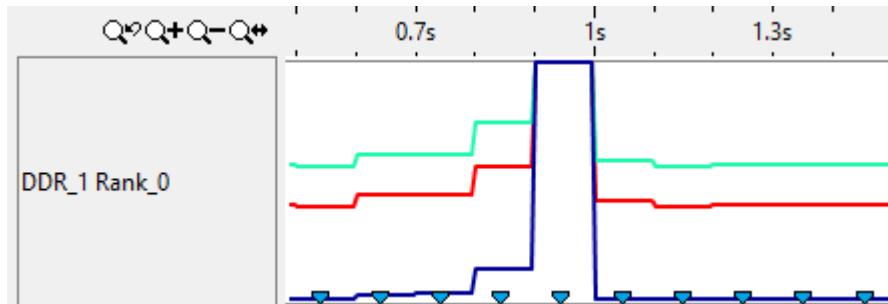
- Sampled Residency Data: The value is gathered by sampling data over regular intervals. There is a set range of values. The exact time of transition between values is not known, but the percentage of time spent in each value is calculated and displayed as a heat map in the timeline pane.

For example, the Graphics C-State status is collected at regular intervals. The value transitioned in and out of different C-States during the collection time, but the exact transition time is not tracked. Instead, a heat map shows that more time was spent in one state than the other. Hover over the graph to see the exact percentage of time spent in each state.



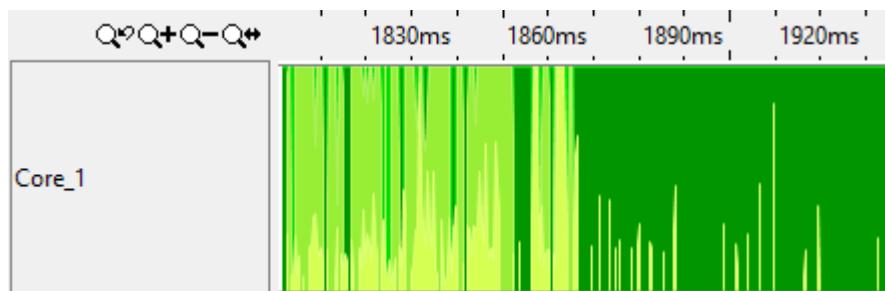
- Sampled Counter Data: The value is gathered by measuring a count since the previous sampling point. The data is then calculated into a rate per second to show the changes over time and visualized as a line graph in the timeline pane.

For example, the DDR Bandwidth data is displayed as a line graph with different lines for read, write, read partials, and write partials. Sampling points show when the counts were collected.



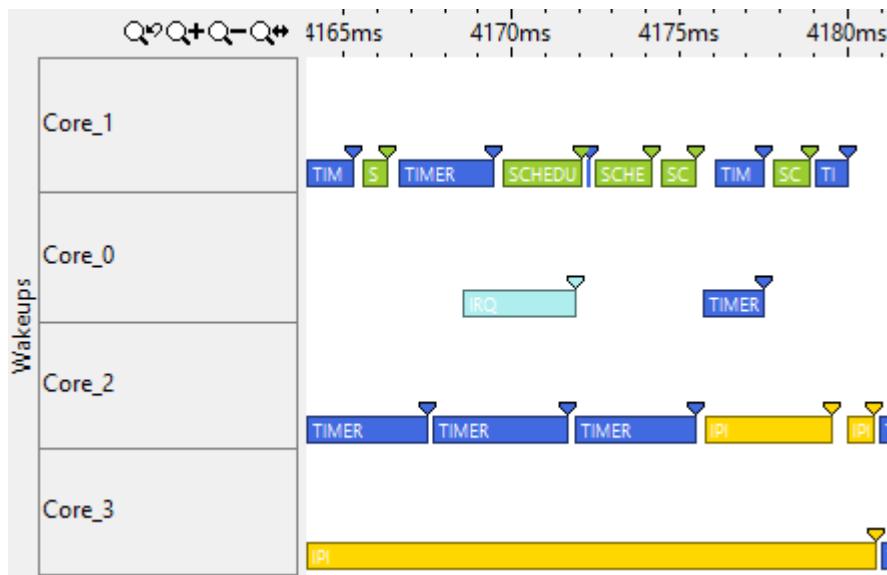
- Traced Residency Data: The value is gathered when the state changes from one value to another. The time spent in the previous state is known and can be displayed in the timeline pane. In some cases an additional metric is tracked, such as the frequency values for the Core P-State Residency metric.

For example, for the Core-C-State Residency Metric, a processor is in a certain C-State at any given time. C0 is the active state and Cn is a sleep state where a larger number means a deeper sleep state. When the processor transitions from one C-State to another, an event is emitted and the transition and time spent in the previous state is logged. The values are visualized as colored bars indicating the time in a certain state in the timeline pane. Drag and select an area of the timeline and then select the **Zoom In on Selection** option from the menu that appears to show finer granularity in the timeline pane. For more information, see [Managing Timeline View](#).



- Traced Event Data: The value is gathered when a new event occurs. Each event is displayed on the timeline with an event marker showing the exact time that the event occurred. Events of the same type are shown with the same color marker. The legend to the right of the timeline shows what color marker corresponds to each event type collected.

Unlike other traced event data, Wakeup and Abort events are displayed as bars and triangle event points on the timeline pane. Each event is color-coded by event type (timer, scheduled, etc.). The bar length shows how the event corresponds with the CPU sleep state, even though the event is instantaneous. The exact time of the wakeup or abort event is shown with the triangle.



See Also

[Window: Summary - Platform Power Analysis](#)

[Window: Bandwidth - Platform Power Analysis](#)

[Window: Core Wake-ups - Platform Power Analysis](#)

[Window: Correlate Metrics - Platform Power Analysis](#)

[Window: CPU C\P States - Platform Power Analysis](#)

[Window: Graphics C\P States - Platform Power Analysis](#)

[Window: NC Device States - Platform Power Analysis](#)

[Window: SC Device States - Platform Power Analysis](#)

[Window: System Sleep States - Platform Power Analysis](#)

[Window: Temperature - Platform Power Analysis](#)

[Window: Timer Resolution - Platform Power Analysis](#)

[Window: Wakelocks - Platform Power Analysis](#)

Window: Sample Count - Hardware Events

Use the **Sample Count** window to analyze the actual number of samples collected for a processor event.

To access this window: Select the **Hardware Events** viewpoint and click the **Sample Count** sub-tab in the result tab. Depending on the analysis type, the **Sample Count** window may include the following panes:

- Sample Count pane
- Timeline pane
- Context Summary pane

Sample Count Pane

The **Sample Count** pane attributes the **Hardware Event Sample Count by Hardware Event Type** to program units. The **Hardware Event Sample Count** metric provides the actual number of samples collected for an event.

By default, the data in the grid is sorted by the Instruction Retired event.

Analysis Configuration Collection Log Summary Event Count Sample Count Caller/Callee				
Function / Call Stack		Hardware Event Sample Count by Hardware Event Type		
		INST_RETIRE	CPU_CLK_UNHALTED	CPU_CLK_UNHALTED
grid_intersect_M_upsendl		149	52	60
> memcmp		79	37	35
> ATOMIC_CompareAndSwap64		70	33	20
> memmove		68	15	19
> strstr		44	10	6
> memset		42	22	15
> ATOMIC_CompareAndSwap32		37	58	33
> LEVEL_BASE_SWMALLOC_Allocate		32	12	15
> _free				2

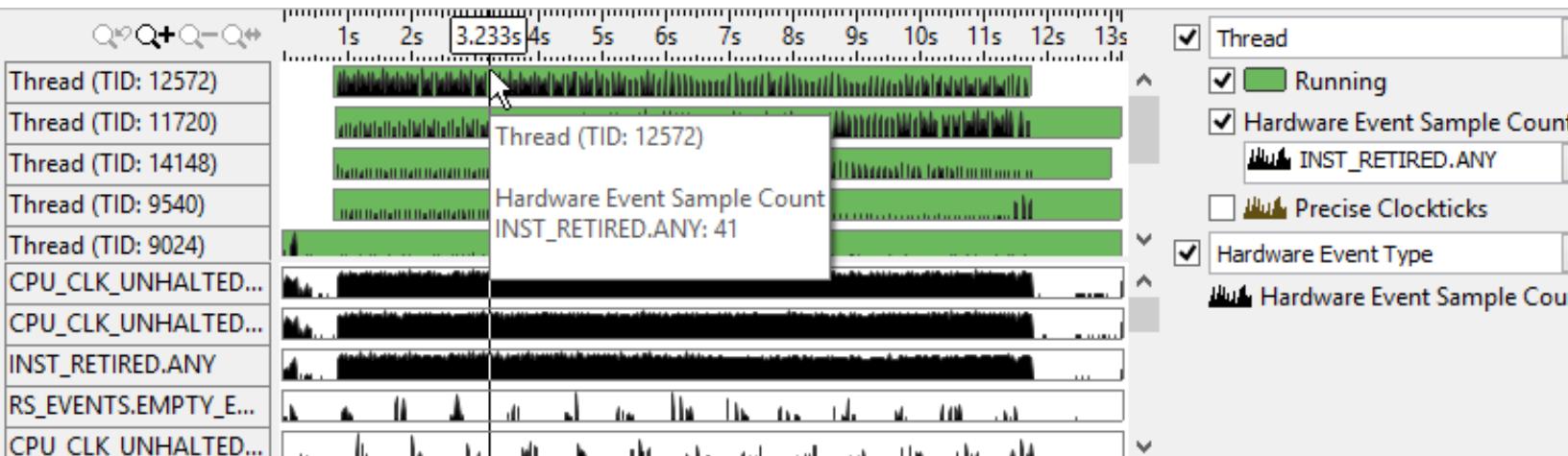
The list of hardware events depends on the analysis type. You may right-click an event column and select the **What's This Column** context menu option to open the description of the selected event.

When you explore the hardware events statistics for a result, you may drag and drop the columns in the grid for your convenience. VTune Profiler automatically saves your preferences and keeps the columns order for subsequent result views.

Timeline Pane

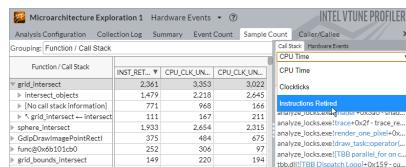
The **Timeline pane** is synchronized with the **Sample Count** pane. The **Thread** area of the **Timeline** pane shows the number of samples collected for the selected event (INST_RETIRE.DY in the example below) while a thread was running. You may use the **Hardware Event Sample Count** drop-down menu in the legend area to choose a different event.

The **Hardware Event Type** area shows the application-level performance per each event.



Call Stack Pane

If you selected the **Collect stacks** option for the hardware event-based sampling analysis, the VTune Profiler provides the **Call Stack pane**. Use this pane to navigate between stacks and analyze the distribution of the sample count for the object selected in the **Sample Count** pane. For the example below, you select the Instructions Retired to see stacks leading to the `grid_intersect` function and contributing to this event. You can use this data to identify the most performance-critical stacks with the highest contribution to the object's Instructions Retired value.



See Also

[Intel Processor Events Reference](#)

[Window: Summary - Hardware Events](#)

[Switch Viewpoints](#)

[Hardware Events Report](#)

from command line

[Window: SC Device States - Platform Power Analysis](#)

To access this window: Select the Platform Power Analysis viewpoint and click the **SC Device States** sub-tab in the result tab.

Use the **SC Device States** window to:

- Identify the time spent in D0ix states by each device.
- Analyze the trend of D0ix state residency over time.
- Review the percent of time a device spent in a particular D0ix state.

The South Complex contains low-intensity computing sub-systems, such as I/O and system management components. D0ix states are low-power states used on system on a chip (SoC) platforms. The South Complex devices are represented using logical sub-system (LSS) identifiers specific to the platform on which the collection was run.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android®, Windows®, or Linux® devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

South Complex Device Pane

Grouping: South Complex Devices		D-States Time by D-State					Device ID
South Complex Devices		D0i0	D0i1	D0i2	D0i3	D0ix	
LSS-02		0.529s	0s	0s	9.544s	0.000s	LSS-02
LSS-00		0s	0s	0s	10.073s	0s	LSS-00
LSS-01		0s	0s	0s	10.073s	0s	LSS-01
LSS-03		0s	0s	0s	10.073s	0s	LSS-03
LSS-04		0s	0s	0s	10.073s	0s	LSS-04
LSS-05		0s	0s	0s	10.073s	0s	LSS-05
LSS-06		0s	0s	0s	10.073s	0s	LSS-06
LSS-07		0s	0s	0s	10.073s	0s	LSS-07
LSS-08		0s	0s	0s	10.073s	0s	LSS-08
LSS-09		10.073s	0s	0s	0s	0s	LSS-09
LSS-10		10.073s	0s	0s	0s	0s	LSS-10
LSS-15		10.073s	0s	0s	0s	0s	LSS-15
LSS-16		0s	0s	0s	10.073s	0s	LSS-16
LSS-17		0s	0s	0s	10.073s	0s	LSS-17

The South Complex Device States pane shows the list of devices in the South Complex and displays estimated sample counts for each device. The sample counts are not a precise measure of the length of time each device spent in a state, but can be used as a guideline to determine if a device spent a greater amount of time in a particular state than was expected.

Click the expand



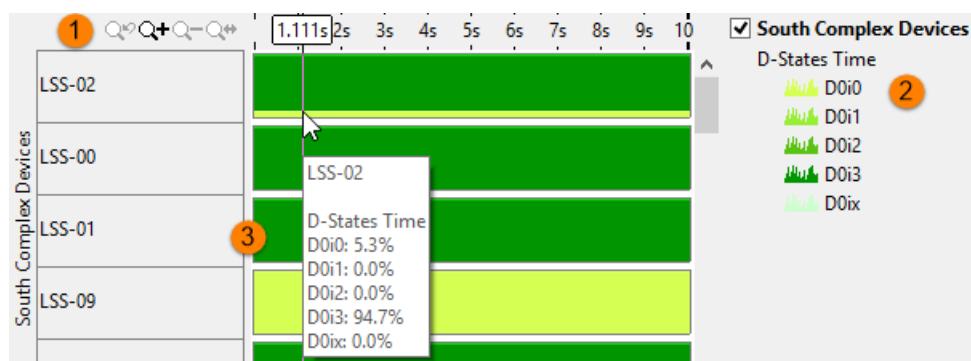
/collapse



buttons in the data columns to expand the column and show data for different D-States in each device. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit. For example, you could select **Show Data As > Percent** to view the percent of collection time a particular device spent in the active state.

Timeline Pane

The Timeline pane displays the D0ix states of each device, at each point in time. You can rearrange the order of the devices in the timeline by dragging and dropping.



1	Toolbar	Navigation control to zoom in/out on the view on areas of interest. For more details on the Timeline control, see Managing Timeline View .
2	Legend	Types of data presented on the timeline. Filter in/out any type of data presented on the timeline by selecting/deselecting corresponding check boxes.
3	South Complex Devices	Graphical representation of the time spent in a D-State. Each state is a different color, which can be filtered using the legend. Hover over the timeline for a device to view the total percentage of time spent in a particular state.
		Zoom in or out on the timeline to view trends in more detail. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

Window: Summary

Use the **Summary** window as a starting point for your analysis in the following viewpoints:

Window: Summary - Input and Output Summary

Use the **Summary** window as your starting point of the performance analysis with the Intel® VTune™ Profiler. To access this window, select the **Input and Output viewpoint** and click the **Summary** sub-tab in the result tab.

Depending on your analysis target, the **Summary** window provides the following application and system-level statistics in the **Disk Input and Output** viewpoint:

- Analysis metrics
- SPDK Info
- SPDK Throughput
- Bandwidth Utilization Histogram
- Top Hotspots
- Disk Input and Output Histogram
- Collection and Platform Info

NOTE

- Click a metric or an object name represented in the Summary window as a hyperlink to open the **Bottom-up** window with the grid data sorted by the selected metric or the selected object highlighted. By default, the grid data is grouped by **Thread/Page Faults**, which helps you easier
- Click the



Copy to Clipboard button to copy the content of the selected summary section to the clipboard.

Analysis Metrics

Explore the list of [CPU metrics](#) to understand high-level statistics of an overall application execution.

For Linux* targets, Intel® VTune™ Profiler introduces the **I/O Wait Time** metric that helps you estimate whether your application is I/O-bound:

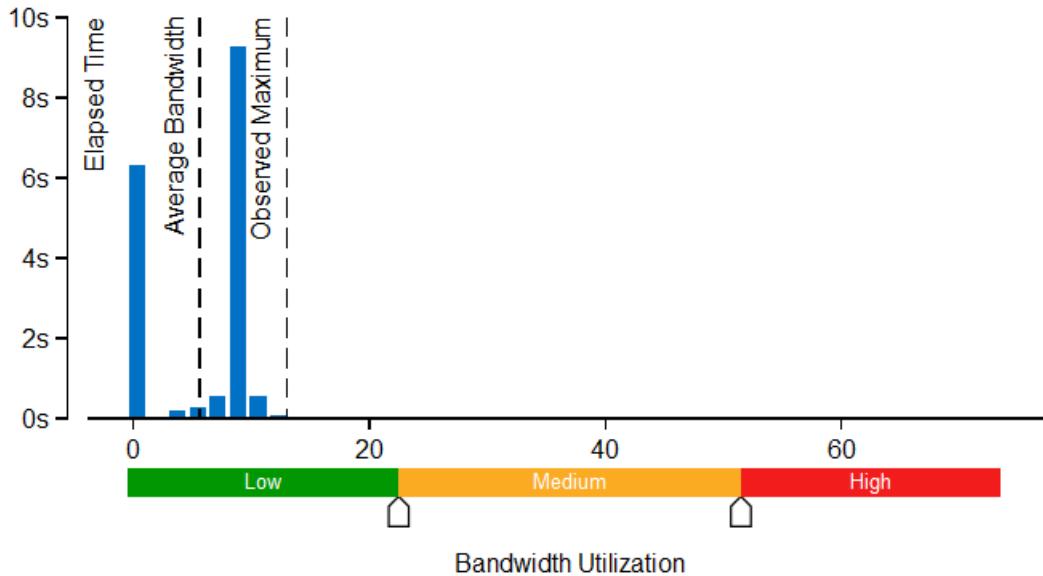
Elapsed Time	7.222s
I/O Wait Time	0.183s
CPU Time	3.958s
Instructions Retired	8,459,636,000
CPI	0.798
Total Thread Count	245
Paused Time	0s

The I/O Wait Time metric represents a portion of time when threads reside in I/O wait state while there are idle cores on the system. For every moment of time the number of counted threads does not exceed the number of idling cores on a system. This aggregated I/O Wait Time metric is an integral function of I/O Wait metric that is available in the Timeline pane of the Bottom-up view. If you see that the I/O Wait Time is a substantial part of the application Elapsed Time, as in the example above, switch to the [Platform window](#) to have a closer look at all the metrics on the timeline and understand what caused high I/O Wait time.

VTune Profiler analyzes metrics, compares their values with the threshold values provided by Intel architects, and, if the threshold is exceeded, it flags the metric value as a performance issue for an application as a whole. Mouse over the flagged value to read an issue description and tuning recommendation.

Bandwidth Utilization Histogram

This histogram shows how much time the system bandwidth was utilized by a certain value (Bandwidth Domain) and provides thresholds to categorize bandwidth utilization as High, Medium and Low. You can set the threshold by moving sliders at the bottom.



NOTE

This histogram is available if you collected results with the **Analyze memory bandwidth** option enabled.

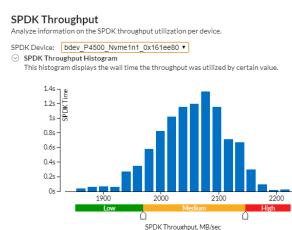
SPDK Info

Explore **SPDK Info** section for overall IO performance statistics. To see how each device performed per operation or metric, expand a corresponding block and identify potential IO performance imbalance among SSDs:

SPDK Info	
② Reads:	284,368
bdev_P4500_NameIn1_0x0f0700:	1
bdev_P4500_NameIn1_0x1da7600:	1
bdev_P3700_NameIn1_0x1da8f700:	1
bdev_P3700_NameIn1_0x1da9e700:	55,585
bdev_P3700_NameIn1_0x1daec700:	55,585
bdev_P3700_NameIn1_0x1daed700:	59,904
bdev_P4500_NameIn1_0x1daef700:	62,010
③ Read Bytes:	35545.6 MB
④ Writes:	282,478
⑤ Written Bytes:	35334.8 MB

SPDK Throughput

Explore the **SPDK Throughput** histogram and table to identify how long your workload has been under-utilizing the throughput of the selected SPDK device (**Low** utilization level):



Top Hotspots

VTune Profiler displays the most performance-critical functions and their CPU Time in the **Top Hotspots** section. Optimizing these functions typically results in improving overall application performance. Clicking a function in the list opens the **Bottom-up** window with this function selected.

Function	Module	CPU Time 
grid_intersect	analyze_locks.exe	3.225s
sphere_intersect	analyze_locks.exe	2.747s
RtlEnterCriticalSection	ntdll.dll	1.341s 
GdipDrawImagePointRectI	gdiplus.dll	1.012s
PeekMessageA	user32.dll	0.771s
[Others]		2.501s 

The grayed-out [Others] module, if provided, displays the total value for all other functions in the application that are not included into this table.

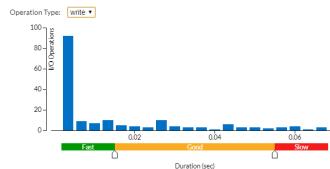
NOTE

You can control the number of objects in this list and displayed metrics via the viewpoint configuration file.

Disk Input and Output Histogram

The **Disk Input and Output** histogram shows how quickly storage requests are served by the kernel subsystem and helps quickly estimate latency distribution and identify slow I/O requests.

The X-axis shows the time it took to satisfy a storage request and the Y-axis shows the number of I/O requests in this category. Use the **Operation type** drop-down menu to select the type of an I/O operation you are interested in. For example, for the **write** type of I/O operations, type of I/O operations, 30 storage requests in all executed for more than 0.03 seconds are qualified by the VTune Profiler as slow:



To get more details on this type I/O request, switch to the **Timeline** pane in the **Bottom-up** window.

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.
Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.

Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collector type	Type of the data collector used for the analysis. The following types are possible: <ul style="list-style-type: none"> • Driver-based sampling • Driver-less Perf*-based sampling: per-process or system-wide • User-mode sampling and tracing
CPU Information	
Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.
Logical CPU Count	Logical CPU count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.
GPU Information	
Name	Name of the Graphics installed on the system.
Vendor	GPU vendor.
Driver	Version of the graphics driver installed on the system.
Stepping	Microprocessor version.
EU Count	Number of execution units (EUs) in the Render and GPGPU engine. This data is Intel® HD Graphics and Intel® Iris® Graphics (further: Intel Graphics) specific.
Max EU Thread Count	Maximum number of threads per execution unit. This data is Intel Graphics specific.
Max Core Frequency	Maximum frequency of the Graphics processor. This data is Intel Graphics specific.
Graphics Performance Analysis	<p>GPU metrics collection is enabled on the hardware level. This data is Intel Graphics specific.</p> <p>NOTE Some systems disable collection of extended metrics such as L3 misses, memory accesses, sampler busyness, SLM accesses, and others in the BIOS. On some systems you can set a BIOS option to enable this collection. The presence or absence of the option and its name are BIOS vendor specific. Look for the Intel® Graphics Performance Analyzers option (or similar) in your BIOS and set it to Enabled.</p>

See Also

[Input and Output Analysis](#)

Comparison Summary

Window: Summary - Microarchitecture Exploration

Use the **Summary** window as your starting point of the performance analysis with the Intel® VTune™ Profiler. To access this window, select the **Microarchitecture Exploration** viewpoint and click the **Summary** sub-tab in the result tab.

Depending on the analysis type, the **Summary** window provides the following application-level statistics in the **Microarchitecture Exploration** viewpoint:

- Microarchitecture metric diagram
- Analysis metrics
- CPU Utilization Histogram
- Collection and Platform Info

NOTE

You may click the

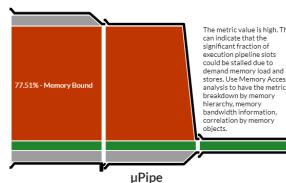


Copy to Clipboard button to copy the content of the selected summary section to the clipboard.

Microarchitecture Metric Diagram

Start your analysis with the [hardware metric diagram](#) representing CPU inefficiencies based on the Top-Down Microarchitecture Analysis Method (TMA).

Treat the diagram as a pipe with an output flow equal to the ratio: **Actual Instructions Retired/Possible Maximum Instruction Retired** (pipe efficiency). If there are pipeline stalls decreasing retiring, the pipe shape gets narrow.

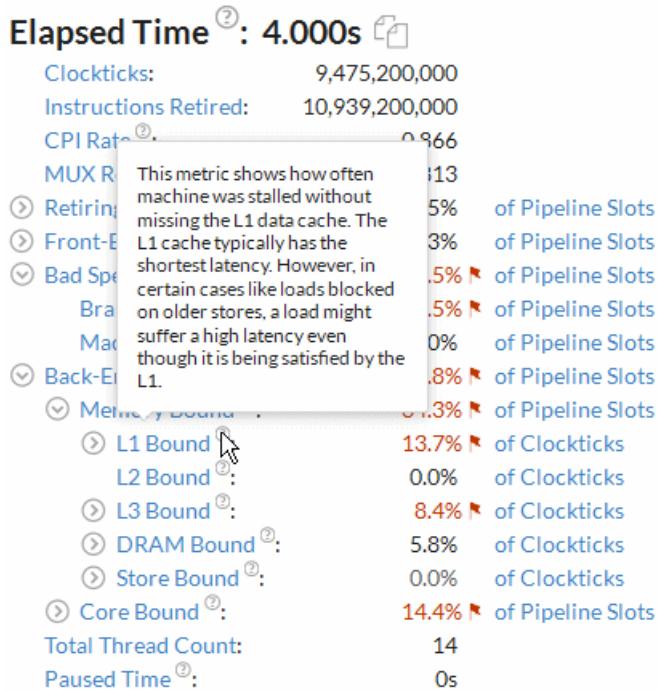


Analysis Metrics

The first section displays the summary statistics on the overall application execution per hardware-related metrics measured in [Pipeline Slots or Clockticks](#). Metrics are organized by execution categories in a list and also represented as a [μPipe diagram](#). To view a metric description, mouse over the help icon

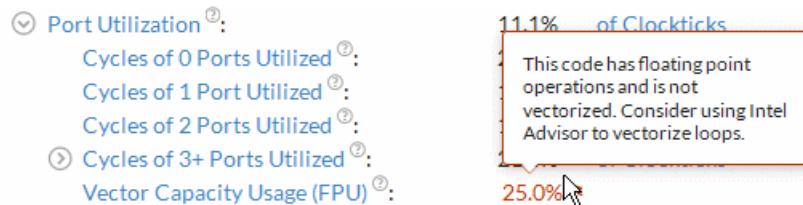


:



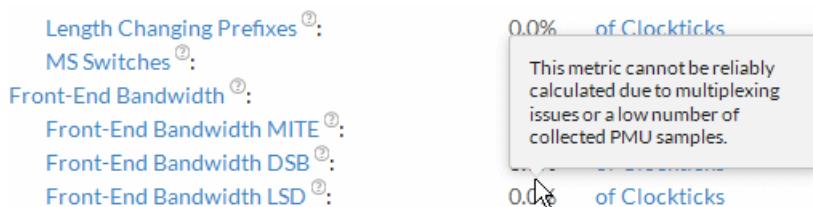
In the example above, mousing over the **L1 Bound** metric displays the metric description in the tooltip.

A flagged metric value signals a performance issue for the whole application execution. Mouse over the flagged value to read the issue description:



You may use the performance issues identified by the VTune Profiler as a baseline for comparison of versions before and after optimization. Your primary performance indicator is the Elapsed time value.

Grayed out metric values indicate that the data collected for this metric is unreliable. This may happen, for example, if the number of samples collected for PMU events is too low. In this case, when you hover over such an unreliable metric value, the VTune Profiler displays a message:

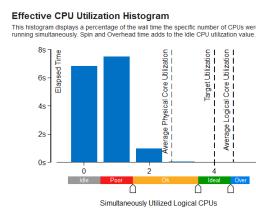


You may either ignore this data, or rerun the collection with the data collection time, sampling interval, or workload increased.

By default, the VTune Profiler collects Microarchitecture Exploration data in the **Detailed** mode. In this mode, all metric names in the Summary view are hyperlinks. Clicking such a hyperlink opens the **Bottom-up** window and sorts the data in the grid by the selected metric. The lightweight **Summary** collection mode is limited to the Summary view statistics.

CPU Utilization Histogram

Explore the **CPU Utilization Histogram** to analyze the percentage of the wall time the specific number of CPUs were running simultaneously.



Use This	To Do This
Vertical bars	Hover over the bar to identify the amount of Elapsed time the application spent using the specified number of logical CPUs.
Target Utilization	Identify the target CPU utilization. This number is equal to the number of logical CPUs. Consider this number as your optimization goal.
Average CPU Utilization	Identify the average number of CPUs used aggregating the entire run. It is calculated as CPU time / Elapsed time. CPU utilization at any point in time cannot surpass the available number of logical CPUs. Even when the system is oversubscribed, and there are more threads running than CPUs, the CPU utilization is the same as the number of CPUs. Use this number as a baseline for your performance measurements. The closer this number to the number of logical CPUs, the better, except for the case when the CPU time goes to spinning.
Utilization Indicator bar	Analyze how the various utilization levels map to the number of simultaneously utilized logical CPUs.

NOTE
In the CPU Utilization histogram, the VTune Profiler treats the [Spin](#) and [Overhead time](#) as Idle CPU utilization. Different analysis types may recognize Spin and Overhead time differently depending on availability of call stack information. This may result in a difference of CPU Utilization graphical representation per analysis type.

NOTE

The **Effective CPU Utilization Histogram** is available for Microarchitecture Exploration results collected in the **Detailed** mode only.

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.

Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collector type	Type of the data collector used for the analysis. The following types are possible: <ul style="list-style-type: none"> • Driver-based sampling • Driver-less Perf*-based sampling: per-process or system-wide • User-mode sampling and tracing
CPU Information	
Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.
Logical CPU Count	Logical CPU count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.
GPU Information	
Name	Name of the Graphics installed on the system.
Vendor	GPU vendor.
Driver	Version of the graphics driver installed on the system.
Stepping	Microprocessor version.
EU Count	Number of execution units (EUs) in the Render and GPGPU engine. This data is Intel® HD Graphics and Intel® Iris® Graphics (further: Intel Graphics) specific.
Max EU Thread Count	Maximum number of threads per execution unit. This data is Intel Graphics specific.
Max Core Frequency	Maximum frequency of the Graphics processor. This data is Intel Graphics specific.
Graphics Performance Analysis	GPU metrics collection is enabled on the hardware level. This data is Intel Graphics specific.

NOTE

Some systems disable collection of extended metrics such as L3 misses, memory accesses, sampler busyness, SLM accesses, and others in the BIOS. On some systems you can set a BIOS option to enable this collection. The presence or absence of the option and its name are BIOS vendor specific. Look for the **Intel® Graphics Performance Analyzers** option (or similar) in your BIOS and set it to **Enabled**.

See Also

[Microarchitecture Exploration View](#)

[Top-Down Microarchitecture Analysis Method](#)

[Comparison Summary](#)

[Change Threshold Values](#)

Window: Summary - GPU Analysis

Use the **Summary** window as your starting point of the GPU Offload or GPU Compute/Media Hotspots performance analysis of the Intel® VTune™ Profiler. To access this window, click the **Summary** sub-tab in the result tab.

Use the [Elapsed Time](#) metric as your primary indicator and a baseline for comparison of results before and after optimization. Note that for multithreaded applications, the CPU Time is different from the Elapsed Time since the CPU Time is the sum of CPU time for all application threads.

Depending on the selected GPU analysis type, the following statistics is available in the **Summary** window:

- [GPU Utilization section](#) helps identify whether the GPU was properly utilized.
- [EU Array Stalled/Idle section](#) helps explore the most typical reasons of the EU waits for compute-bound applications.
- [FPU Utilization section](#) helps identify kernels over-utilizing both FPUs for FPU-bound applications.
- [Bandwidth Utilization section](#) provides statistics for memory-bound applications.

NOTE

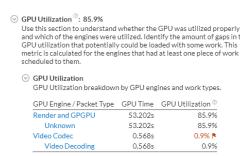
Click the



[Copy to Clipboard](#) button to copy the content of the selected summary section to the clipboard.

GPU Utilization

If your system satisfies [configuration requirements for GPU analysis](#) (i915 ftrace event collection is supported), VTune Profiler displays detailed **GPU Utilization** analysis data across all engines that had at least one DMA packet executed. By default, the VTune Profiler flags the GPU utilization less than 80% as a performance issue. In the example below, 85.9% of the application elapsed time was utilized by GPU engines.

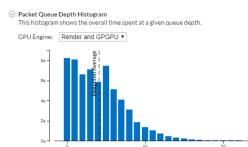


Depending on the target platform used for GPU analysis, the **GPU Utilization** section in the Summary window shows the time (in seconds) used by GPU engines. Note that GPU engines may work in parallel and the total time taken by GPU engines does not necessarily equal the application Elapsed time.

You may correlate GPU Time data with the Elapsed Time metric. The GPU Time value shows a share of the Elapsed time used by a particular GPU engine. If the GPU Time takes a significant portion of the Elapsed Time, it clearly indicates that the application is GPU-bound.

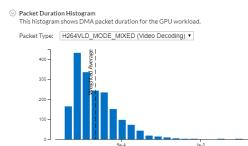
If your system does not support i915 ftrace event collection, all the GPU Utilization statistics will be calculated based on the hardware events and attributed to the **Render and GPGPU** engine.

The **Summary** view provides the **Packet Queue Depth Histogram** that helps you estimate the GPU software queue depth per GPU engine during the target run:



Ideally, your goal is an effective GPU engine utilization with evenly loaded queues and minimal duration for the zero queue depth.

For a high-level view of the DMA packet execution during the target run, review the **Packet Duration Histogram**:



Select a required packet type from the drop-down menu and identify how effectively these packets were executed on the GPU. Having high Packet Count values for the minimal duration is optimal.

To get detailed information on the packet queues and execution, switch to the [Platform tab](#) and analyze the GPU software queue on the timeline.

For OpenCL™ applications, explore the **Hottest GPU Computing Tasks** section that helps you understand which OpenCL kernels had performance issues:

[Hottest GPU Computing Tasks](#)

This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

Computing Task	Total Time	Average Time	Instance Count
Intersect ↗	83.530s	0.006s	13,882
AdvancePaths ↗	23.197s	0.002s	13,882
Sampler ↗	13.815s	0.001s	13,882
clEnqueueReadBuffer ↗	0.550s	0.000s	2,250
Init ↗	0.003s	0.003s	1
[Others]	0.000s	0.000s	1

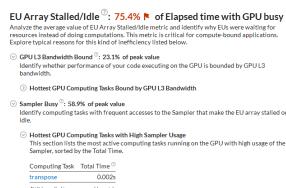
*N/A is applied to non-summable metrics.

Mouse over a flagged computing task for details on a performance issue. For example, for the `Intersect` computing task a significant portion of the GPU time was spent in stalls, which may result from frequent sampler or memory accesses. Click a hot GPU computing task to open the [Graphics window](#) with this computing task pre-selected for your convenience.

EU Array Stalled/Idle

For the compute-bound workloads, explore the **EU Array Stalled/Idle** section that shows the most typical reasons why the execution units could be waiting. This section shows up for the analysis that collects Intel® HD Graphics and Intel® Iris® Graphics hardware events for the GPU Compute/Media Hotspots.

Depending on the event preset you used for the configuration, the VTune Profiler analyzes metrics for stalled/idle executions units. The GPU Compute/Media Hotspots analysis by default collects the [Overview preset](#) including the metrics that track general GPU memory accesses, such as Sampler Busy and Sampler Is Bottleneck, and GPU L3 bandwidth. As a result, the **EU Array Stalled/Idle** section displays the Sampler Busy section with a list of GPU computing tasks with frequent access to the Sampler and hottest GPU computing tasks bound by GPU L3 bandwidth:



If you select the [Compute Basic](#) preset during the analysis configuration, VTune Profiler analyzes metrics that distinguish accessing different types of data on a GPU and displays the **Occupancy** section. See information about GPU tasks with low occupancy and understand how you can achieve peak occupancy:

Welcome x r084gh x

GPU Compute/Media Hotspots (preview) GPU Compute/Media Hotspots (preview) ? !

Analysis Configuration Collection Log **Summary** Graphics

Elapsed Time ?: 5.700s

If your application target was run more than once during the collection, this value includes elapsed time for all the runs.

GPU Time ?: 0.009s

EU Array Stalled/Idle ?: 84.3% ! !

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of being critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

- (GPU L3 Bandwidth Bound ?: 0.0% !
- (Occupancy ?: 53.2% ! !

Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the accesses and barriers may affect the maximum possible occupancy.

Hottest GPU Computing Tasks with Low Occupancy !

This section lists the most active computing tasks running on the GPU with the lowest occupancy.

Computing Task	Total Time ?	Global Size ?	Local Size ?	SIMD Width ?
workload	0.008s	4096	32	

*N/A is applied to non-summable metrics.

Occupancy, %

25.5%

0% 26%

red flag zone tuning potential

The normalized sum of all cycles on all slots when a slot has a thread scheduled value).

! Ineffective work scheduling can cause spikes in the occupancy metric.

- (Sampler Busy ?: 78.2% !

FPU Utilization ?: 1.9%

Bandwidth Utilization Histogram

If the **peak occupancy** is flagged as a problem for your application, inspect factors that limit the use of all the threads on the GPU. Consider modifying your code with corresponding solutions:

Factor responsible for Low Peak Occupancy	Solution
SLM size requested per workgroup in a computing task is too high	Decrease the SLM size or increase the Local size
Global size (the number of working items to be processed by a computing task) is too low	Increase Global size

Factor responsible for Low Peak Occupancy	Solution
Barrier synchronization (the sync primitive can cause low occupancy due to a limited number of hardware barriers on a GPU subslice)	Remove barrier synchronization or increase the Local size

EU Array Stalled/Idle: 79.9% ↗

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computations. This metric is critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

GPU L3 Bandwidth Bound: 9.5% ↗

Occupancy: 78.8% ↗

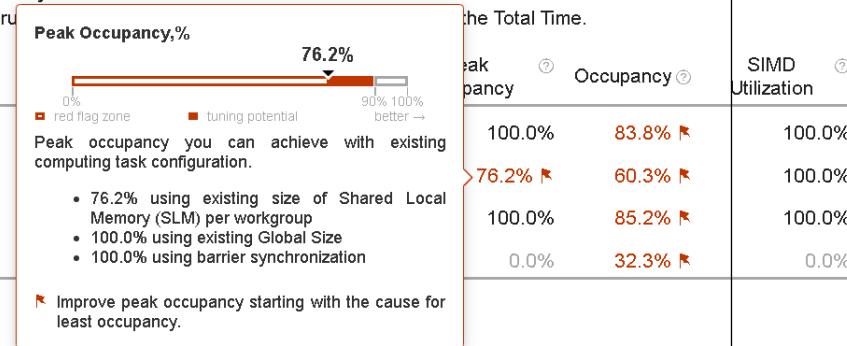
Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the scheduler. Note that frequent SLM accesses and barriers may affect the maximum possible occupancy.

Hottest GPU Computing Tasks with Low Occupancy

This section lists the most active computing tasks running on the GPU with a low Occupancy, sorted by the Total Time.

Computing Task	Total Time
kernel_ocl_path_trace_shader_evaluation	0.457s
kernel_ocl_path_trace_shader_sort	0.323s
kernel_ocl_path_trace_lamp_emission	0.208s
[Others]	0.033s

*N/A is applied to non-summable metrics.



If the **occupancy** is flagged as a problem for your application, change your code to improve hardware thread scheduling. These are some reasons that may be responsible for ineffective thread scheduling:

- A tiny computing task could cause considerable overhead when compared to the task execution time.
- There may be high imbalance between the threads executing a computing task.

EU Array Stalled/Idle: 79.9% ↗

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computations. This metric is critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

GPU L3 Bandwidth Bound: 9.5% ↗

Occupancy: 78.8% ↗

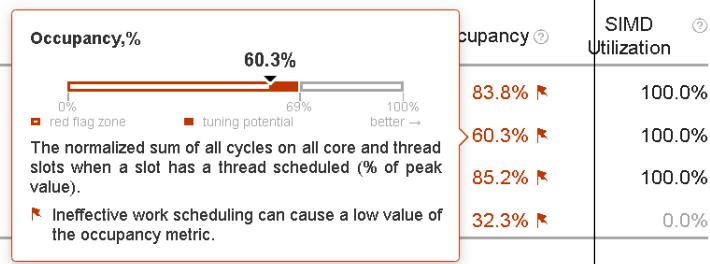
Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the scheduler. Note that frequent SLM accesses and barriers may affect the maximum possible occupancy.

Hottest GPU Computing Tasks with Low Occupancy

This section lists the most active computing tasks running on the GPU with a low Occupancy, sorted by the Total Time.

Computing Task	Total Time	Global Size
kernel_ocl_path_trace_shader_evaluation	0.457s	640 x 652
kernel_ocl_path_trace_shader_sort	0.323s	640 x 652
kernel_ocl_path_trace_lamp_emission	0.208s	640 x 652
[Others]	0.033s	

*N/A is applied to non-summable metrics.



The **Compute Basic** preset also enables an analysis of the DRAM bandwidth usage. If the GPU workload is DRAM bandwidth-bound, the corresponding metric value is flagged. You can explore the table with GPU computing tasks heavily using the DRAM bandwidth during execution.

If you select the **Full Compute** preset and **multiple run mode** during the analysis configuration, the VTune Profiler will use both **Overview** and **Compute Basic** event groups for data collection and provide all types of reasons for the EU array stalled/idle issues in the same view.

NOTE

To analyze Intel® HD Graphics and Intel® Iris® Graphics hardware events, make sure to [set up your system for GPU analysis](#)

FPU Utilization

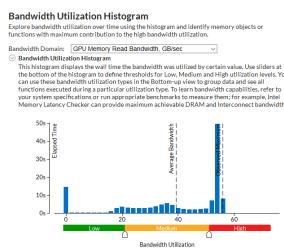
If your application execution takes more than 80% of collection time heavily utilizing both floating point units (FPUs), the VTune Profiler highlights such a value as an issue and lists the kernels that overutilized the FPUs:



Click a flagged kernel to switch to the **Graphics** tab > **Timeline** pane, explore the distribution of the **GPU EU Instructions** metric that shows the FPU usage during the analysis run, and identify time ranges with the highest metric values. To address high FPU utilization issue for your code, consider reducing computations.

Bandwidth Utilization

For memory-bound applications, explore the **Bandwidth Utilization Histogram** section that includes statistics on the average system bandwidth and a Bandwidth Utilization histogram that shows how intensively your application was using each bandwidth domain:



Collection and Platform Info

Explore the platform information including GPU and CPU data. The last four GPU characteristics are specific to Intel® HD Graphics and Intel® Iris® Graphics.

GPU OpenCL™ Application Analysis

GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics

GPU Compute/Media Hotspots Analysis (Preview)

Window: Summary - Hardware Events

Use the **Summary** window as your starting point of the performance analysis with the Intel VTune Profiler. To access this window, select the **Hardware Events** viewpoint and click the **Summary** sub-tab in the result tab.

The **Hardware Events** viewpoint is enabled for all hardware event-based sampling results and is targeted primarily for the analysis of monitored hardware events: estimated count and/or the number of samples collected. In the **Summary** window, explore the following data:

- [Analysis metrics](#)
- [Hardware Events](#)
- [Uncore Event Count](#)
- [Top Tasks](#)
- [Collection and Platform Info](#)

NOTE

You may click the



[**Copy to Clipboard**](#) button to copy the content of the selected summary section to the clipboard.

Analysis Metrics

The **Summary** window displays a list of [CPU metrics](#) that help you estimate an overall application execution. For a metric description, hover over the corresponding question mark icon



to read the pop-up help.

Use the Elapsed Time metric as your primary indicator and a baseline for comparison of results before and after optimization. Note that for multithreaded applications, the CPU Time is different from the Elapsed Time since the CPU Time is the sum of CPU time for all application threads.

Hardware Events

This section provides a list of hardware events monitored for this analysis and the statistics collected:

Hardware Event Type	Event name provided as a hyperlink. Clicking an event name opens the Event Count window sorted by the selected event. You can identify a function with the highest event/sample count and double-click it to open the Source view and identify which code line generated the highest count for the event of interest.
Hardware Event Count	Estimated number of times this event occurred during the collection.
Hardware Event Sample Count	Actual number of samples collected for this event.
Events per Sample	Number of events collected at one sample (Sample After Value).

Uncore Event Count

This section provides a list of uncore hardware events monitored for this analysis and the statistics collected:

Uncore Event Type	Event name provided as a hyperlink. Clicking an event name opens the Uncore Event Count window sorted by the selected event.
Uncore Event Count	The number of times this uncore event occurred during the collection.

Top Tasks

This section provides a list of tasks that took most of the time to execute, where *tasks* are either code regions marked with Task API, or system tasks enabled to monitor Ftrace* events, Atrace* events, Intel Media SDK programs, OpenCL™ kernels, and so on.

Clicking a task type in the table opens the grid view (for example, Bottom-up or Event Count) grouped by the **Task Type** granularity. See [Task Analysis](#) for more information.

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.
Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collector type	Type of the data collector used for the analysis. The following types are possible: <ul style="list-style-type: none"> • Driver-based sampling • Driver-less Perf*-based sampling: per-process or system-wide • User-mode sampling and tracing
CPU Information	
Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.
Logical CPU Count	Logical CPU count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.
GPU Information	
Name	Name of the Graphics installed on the system.
Vendor	GPU vendor.
Driver	Version of the graphics driver installed on the system.

Stepping	Microprocessor version.
EU Count	Number of execution units (EUs) in the Render and GPGPU engine. This data is Intel® HD Graphics and Intel® Iris® Graphics (further: Intel Graphics) specific.
Max EU Thread Count	Maximum number of threads per execution unit. This data is Intel Graphics specific.
Max Core Frequency	Maximum frequency of the Graphics processor. This data is Intel Graphics specific.
Graphics Performance Analysis	<p>GPU metrics collection is enabled on the hardware level. This data is Intel Graphics specific.</p> <p>NOTE Some systems disable collection of extended metrics such as L3 misses, memory accesses, sampler busyness, SLM accesses, and others in the BIOS. On some systems you can set a BIOS option to enable this collection. The presence or absence of the option and its name are BIOS vendor specific. Look for the Intel® Graphics Performance Analyzers option (or similar) in your BIOS and set it to Enabled.</p>

See Also

[Sample After Value](#)

[Intel Processor Events Reference](#)

Window: Summary - Hotspots by CPU Utilization

Use the **Summary** window as your starting point of the performance analysis with the Intel® VTune™ Profiler. To access this window, select the **Hotspots by CPU Utilization** viewpoint and click the **Summary** sub-tab in the result tab.

Depending on the analysis type, the **Summary** window provides the following application-level statistics in the **Hotspots by CPU Utilization** viewpoint:

- Analysis metrics
- Top Hotspots
- Top Tasks
- Effective CPU Utilization Histogram
- Frame Rate Histogram
- Collection and Platform Info

NOTE

You may click the



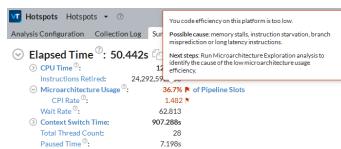
[Copy to Clipboard](#) button to copy the content of the selected summary section to the clipboard.

Analysis Metrics

The **Summary** window displays a list of **CPU metrics** that help you estimate an overall application execution. For a metric description, hover over the corresponding question mark icon



to read the pop-up help. For metric values flagged as performance issues, hover over such a value for details:



Use the Elapsed Time metric as your primary indicator and a baseline for comparison of results before and after optimization. Note that for multithreaded applications, the CPU Time is different from the Elapsed Time since the CPU Time is the sum of CPU time for all application threads.

For some analysis types, the Effective CPU Time is classified per CPU utilization as follows:

Utilization Type	Description
Idle	Idle utilization. By default, if the CPU Time is insignificant (less than 50% of 1 CPU), such CPU utilization is classified as idle.
Poor	Poor utilization. By default, poor utilization is when the number of simultaneously running CPUs is less than or equal to 50% of the target CPU utilization.
OK	Acceptable (OK) utilization. By default, OK utilization is when the number of simultaneously running CPUs is between 51-85% of the target CPU utilization.
Ideal	Ideal utilization. By default, Ideal utilization is when the number of simultaneously running CPUs is between 86-100% of the target CPU utilization.

The [Overhead and Spin Time metrics](#), if provided (depend on the analysis), can tell you how your application's use of synchronization and threading libraries is impacting the CPU time. Review the metrics within these categories to learn where your application might be spending additional time making calls to synchronization and threading libraries such as system synchronization API, Intel® oneAPI Threading Building Blocks(oneTBB), and OpenMP*. VTune Profiler provides the following types of inefficiencies in your code taking CPU time:

Imbalance or Serial Spinning Time	Imbalance or Serial Spinning time is CPU time when working threads are spinning on a synchronization barrier consuming CPU resources. This can be caused by load imbalance, insufficient concurrency for all working threads or waits on a barrier in the case of serialized execution.
Lock Contention Spin Time	Lock Contention time is CPU time when working threads are spinning on a lock consuming CPU resources. High metric value may signal inefficient parallelization with highly contended synchronization objects. To avoid intensive synchronization, consider using reduction, atomic operations or thread local variables where possible.
Other Spin Time	This metric shows unclassified Spin time spent in a threading runtime library.
Creation Overhead Time	Creation time is CPU time that a runtime library spends on organizing parallel work.
Scheduling Overhead Time	Scheduling time is CPU time that a runtime library spends on work assignment for threads. If the time is significant, consider using coarse-grain work chunking.

Reduction Overhead Time	Reduction time is CPU time that a runtime library spends on loop or region reduction operations.
Atomics Overhead Time	Atomics time is CPU time that a runtime library spends on atomic operations.
Other Overhead Time	This metric shows unclassified Overhead time spent in a threading runtime library.

Depending on the analysis type, the VTune Profiler may analyze a metric, compare its value with the threshold value provided by Intel architects, and highlight the metric value in pink as a performance issue for an application as a whole. The issue description for such a value may be provided below the critical metric or when you hover over the highlighted metric.

Each metric in the list shows up as a hyperlink. Clicking a hyperlink opens the **Bottom-up** window and sorts the grid by the selected metric or highlights the selected object in the grid.

Top Hotspots

VTune Profiler displays the most performance-critical functions and their CPU Time in the **Top Hotspots** section. Optimizing these functions typically results in improving overall application performance. Clicking a function in the list opens the **Bottom-up** window with this function selected.

Function	Module	CPU Time
grid_intersect	analyze_locks.exe	3.225s
sphere_intersect	analyze_locks.exe	2.747s
RtlEnterCriticalSection	ntdll.dll	1.341s
GdipDrawImagePointRectI	gdiplus.dll	1.012s
PeekMessageA	user32.dll	0.771s
[Others]		2.501s

The grayed-out [Others] module, if provided, displays the total value for all other functions in the application that are not included into this table.

NOTE

You can control the number of objects in this list and displayed metrics via the viewpoint configuration file.

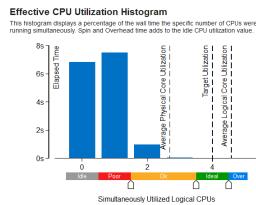
Top Tasks

This section provides a list of tasks that took most of the time to execute, where *tasks* are either code regions marked with Task API, or system tasks enabled to monitor Ftrace* events, Atrace* events, Intel Media SDK programs, OpenCL™ kernels, and so on.

Clicking a task type in the table opens the grid view (for example, Bottom-up or Event Count) grouped by the **Task Type** granularity. See [Task Analysis](#) for more information.

Effective CPU Utilization Histogram

Explore the **Effective CPU Utilization Histogram** to analyze the percentage of the wall time the specific number of logical CPUs were running simultaneously. Spin and Overhead Time adds to the Idle CPU Utilization value.



Use This	To Do This
Vertical bars	Hover over the bar to identify the amount of Elapsed time the application spent using the specified number of logical CPU cores.
Target Utilization	Identify the target CPU utilization. This number is equal to the number of logical CPU cores. Consider this number as your optimization goal.
Average Effective CPU Utilization	Identify the average number of CPUs used aggregating the entire run. It is calculated as CPU time / Elapsed time. CPU utilization at any point in time cannot surpass the available number of logical CPU cores. Even when the system is oversubscribed, and there are more threads running than CPUs, the CPU utilization is the same as the number of CPUs. Use this number as a baseline for your performance measurements. The closer this number to the number of logical CPU cores, the better, except for the case when the CPU time goes to spinning.
Utilization Indicator bar	Analyze how the various utilization levels map to the number of simultaneously utilized logical CPU cores.

NOTE
In the CPU Utilization histogram, the VTune Profiler treats the **Spin** and **Overhead** time as Idle CPU utilization. Different analysis types may recognize Spin and Overhead time differently depending on availability of call stack information. This may result in a difference of CPU utilization graphical representation per analysis type.

Frame Rate Histogram

If you used the Frame API to mark the start and finish of the code regions executed repeatedly (*frames*) in your graphics application, the VTune Profiler analyzes this data and helps you identify regions that ran slowly. Explore the **Frame Rate Histogram** section and [identify slow and fast frame domains](#).

Use This	To Do This
Domain drop-down menu	Choose a frame domain to analyze with the frame rate histogram. If only one domain is available, the drop-down menu is grayed out. Then, you can switch to the Bottom-up window grouped by Frame Domain , filter the data by slow frames and switch to the Function grouping to identify functions in the slow frame domains. Try to optimize your code to keep the frame rate constant (for example, from 30 to 60 frames per second).

Use This	To Do This
Vertical bars	Hover over a bar to see the total number of frames in your application executed with a specific frame rate. High number of slow or fast frames signals a performance bottleneck.
Frame rate bar	Use the sliders to adjust the frame rate threshold (in frames per second) for the currently open result and all subsequent results in the project.

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.
Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collector type	Type of the data collector used for the analysis. The following types are possible: <ul style="list-style-type: none"> • Driver-based sampling • Driver-less Perf*-based sampling: per-process or system-wide • User-mode sampling and tracing

CPU Information

Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.
Logical CPU Count	Logical CPU count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.

GPU Information

Name	Name of the Graphics installed on the system.
Vendor	GPU vendor.
Driver	Version of the graphics driver installed on the system.

Stepping	Microprocessor version.
EU Count	Number of execution units (EUs) in the Render and GPGPU engine. This data is Intel® HD Graphics and Intel® Iris® Graphics (further: Intel Graphics) specific.
Max EU Thread Count	Maximum number of threads per execution unit. This data is Intel Graphics specific.
Max Core Frequency	Maximum frequency of the Graphics processor. This data is Intel Graphics specific.
Graphics Performance Analysis	<p>GPU metrics collection is enabled on the hardware level. This data is Intel Graphics specific.</p> <p>NOTE Some systems disable collection of extended metrics such as L3 misses, memory accesses, sampler busyness, SLM accesses, and others in the BIOS. On some systems you can set a BIOS option to enable this collection. The presence or absence of the option and its name are BIOS vendor specific. Look for the Intel® Graphics Performance Analyzers option (or similar) in your BIOS and set it to Enabled.</p>

See Also

[Comparison Summary](#)

[Thread Concurrency](#)

[CPU Utilization](#)

[Changing Threshold Values](#)

Window: Summary - HPC Performance Characterization

Use the **Summary** window as your starting point of the performance analysis with the Intel® VTune™ Profiler. To access this window, click the **Summary** sub-tab in the result tab.

The VTune Profiler may analyze a metric, compare its value with the threshold value provided by Intel architects, and highlight the metric value in pink as a performance issue for an application as a whole. The issue description for such a value may be provided below the critical metric or when you hover over the highlighted metric.

The **Summary** window provides the following application-level statistics in the **HPC Performance Characterization** viewpoint:

- [Analysis Metrics](#)
- [CPU Utilization](#)
- [Memory Bound](#)
- [Vectorization](#)
- [Collection and Platform Info](#)

NOTE

You may click the



Copy to Clipboard button to copy the content of the selected summary section to the clipboard.

Analysis Metrics

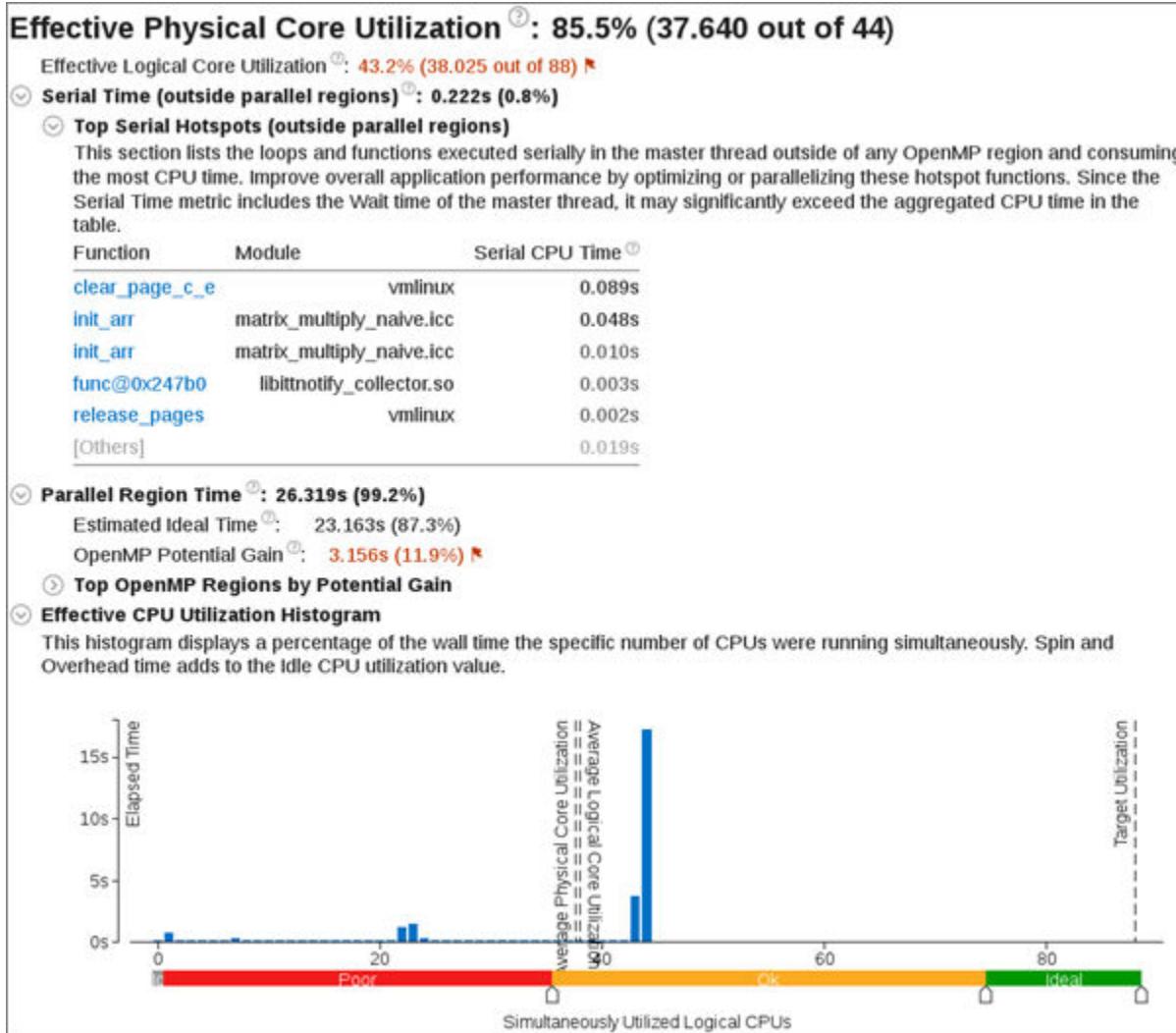
The **Summary** window displays metrics that help you estimate an overall application execution. For a metric description, hover over the corresponding question mark icon  to read the pop-up help.

Use the Elapsed Time, GFLOPS, or GFLOPS Upper Bound (Intel® Xeon Phi™ processor only) metric as your primary indicator and a baseline for comparison of results before and after optimization.

Elapsed Time : 19.285s
GFLOPS : 19.020

CPU Utilization

The CPU Utilization section displays metrics for CPU usage during the collection time.



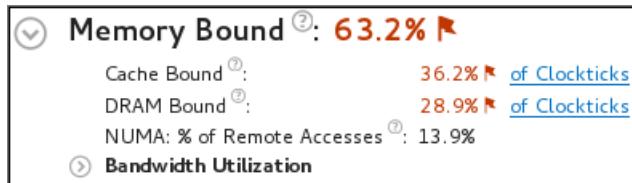
Metrics can include:

- **OpenMP Analysis Collection Time:** Displays metrics for the duration of serial (outside of any parallel region) and parallel portions of the program. If the **Serial time** is significant, review the **Top Serial Hotspots** section and consider options to minimize serial execution, either by introducing more

parallelism or by doing algorithm or microarchitecture tuning for sections that seem unavoidably serial. For high thread-count machines, serial sections have a severe negative impact on potential scaling (Amdahl's Law) and should be minimized as much as possible.

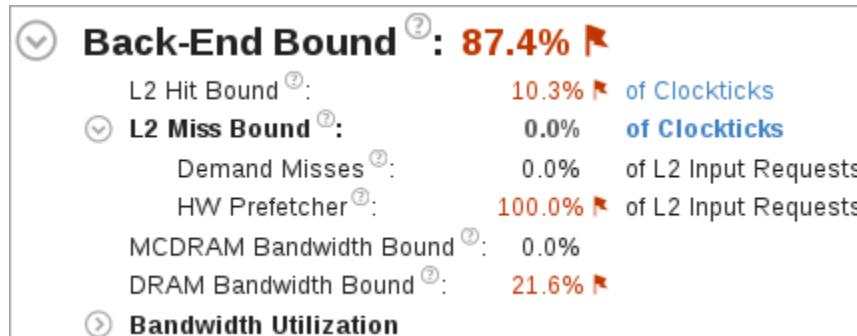
- **Top OpenMP Regions by Potential Gain:** Displays the efficiency of Intel OpenMP* parallelization in the parallel part of the code and checks for an MPI imbalance. The **Potential Gain** metric estimates the elapsed time between the actual measurement and an idealized execution of parallel regions, assuming perfectly balanced threads and zero overhead of the OpenMP runtime on work arrangement. Use this data to understand the maximum time that you may save by improving parallel execution. If Potential Gain for a region is significant, you can go deeper and select the link on a region name to navigate to the **Bottom-up** window employing an **OpenMP Region** dominant grouping and the region of interest selection.
- **Effective CPU Utilization Histogram:** Graphical representation of the percentage of wall time the specific number of CPUs the application was running simultaneously. The CPU usage does not contain spin and overhead time that does not perform actual work. Hover over a vertical bar to identify the amount of Elapsed Time the application spent using the specified number of logical CPU cores. Use the Average Physical Core Utilization and Average Logical Core Utilization numbers as a baseline for your performance measurements. The CPU usage at any point cannot surpass the available number of logical CPU cores.

Memory Bound



A high Memory Bound value might indicate that a significant portion of execution time was lost while fetching data. The section shows a fraction of cycles that were lost in stalls being served in different cache hierarchy levels (L1, L2, L3) or fetching data from DRAM. For last level cache misses that lead to DRAM, it is important to distinguish if the stalls were because of a memory bandwidth limit since they can require specific optimization techniques when compared to latency bound stalls. VTune Profiler shows a hint about identifying this issue in the DRAM Bound metric issue description. This section also offers the percentage of accesses to a remote socket compared to a local socket to see if memory stalls can be connected with NUMA issues.

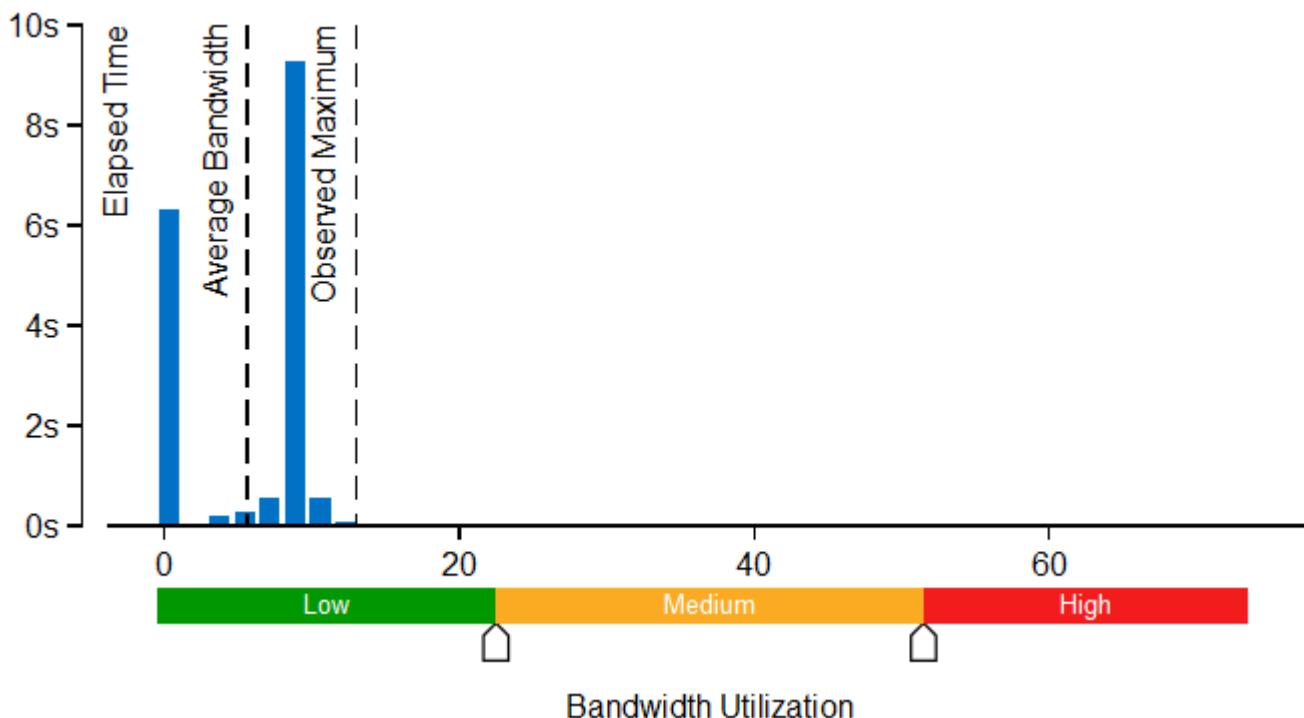
For Intel® Xeon Phi™ processors formerly code named Knights Landing, there is no way to measure memory stalls to assess memory access efficiency in general. Therefore Back-end Bound stalls that include memory-related stalls as a high-level characterization metric are shown instead. The second level metrics are focused particularly on memory access efficiency.



- A high **L2 Hit Bound** or **L2 Miss Bound** value indicates that a high ratio of cycles were spent handling L2 hits or misses.
- The **L2 Miss Bound** metric does not take into account data brought into the L2 cache by the hardware prefetcher. However, in some cases the hardware prefetcher can generate significant DRAM/MCDRAM traffic and saturate the bandwidth. The **Demand Misses** and **HW Prefetcher** metrics show the percentages of all L2 cache input requests that are caused by demand loads or the hardware prefetcher.

- A high **DRAM Bandwidth Bound** or **MCDRAM Bandwidth Bound** value indicates that a large percentage of the overall elapsed time was spent with high bandwidth utilization. A high **DRAM Bandwidth Bound** value is an opportunity to run the [Memory Access](#) analysis to identify data structures that can be allocated in high bandwidth memory (MCDRAM), if it is available.

The **Bandwidth Utilization Histogram** shows how much time the system bandwidth was utilized by a certain value (Bandwidth Domain) and provides thresholds to categorize bandwidth utilization as High, Medium and Low. The thresholds are calculated based on benchmarks that calculate the maximum value. You can also set the threshold by moving the sliders at the bottom of the histogram. The modified values are applied to all subsequent results in the project.



If your application is memory bound, consider running a [Memory Access](#) analysis to identify deeper memory issues and examine memory objects in more detail.

Vectorization

Top Loops/Functions with FPU Usage by CPU Time						
Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
[Loop at line 526 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std::miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@518]	119.475s	15.7%	55.8%	44.2% ■	AVX(128); FMA(128) ■	Body
[Loop at line 204 in miniFE::daxpy<miniFE::Vector<double, int, int>>>\$omp\$parallel_for@204]	19.962s				AVX(256); FMA(256) ?	Body
[Loop at line 526 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std::miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@518]	16.779s	25.3%	98.9%	1.1%		Remainder
[Loop at line 519 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std::miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@518]	10.309s	26.4%	93.2%	6.8%	AVX(128); AVX512F_128(128); FMA(128)	Body
[Loop at line 248 in miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std::miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>>>\$omp\$parallel_for@248]	8.054s	10.7%	100.0%	0.0%	AVX(256); FMA(256)	Body
[Others]	21.160s	58.5%	8.2%	91.8% ■		

*NA is applied to non-summable metrics.

NOTE

Vectorization and GFLOPS metrics are supported on Intel® microarchitectures formerly code named Ivy Bridge, Broadwell, and Skylake. Limited support is available for Intel® Xeon Phi™ processors formerly code named Knights Landing. The metrics are not currently available on 4th Generation Intel processors. Expand the **Details** section on the analysis configuration pane to view the processor family available on your system.

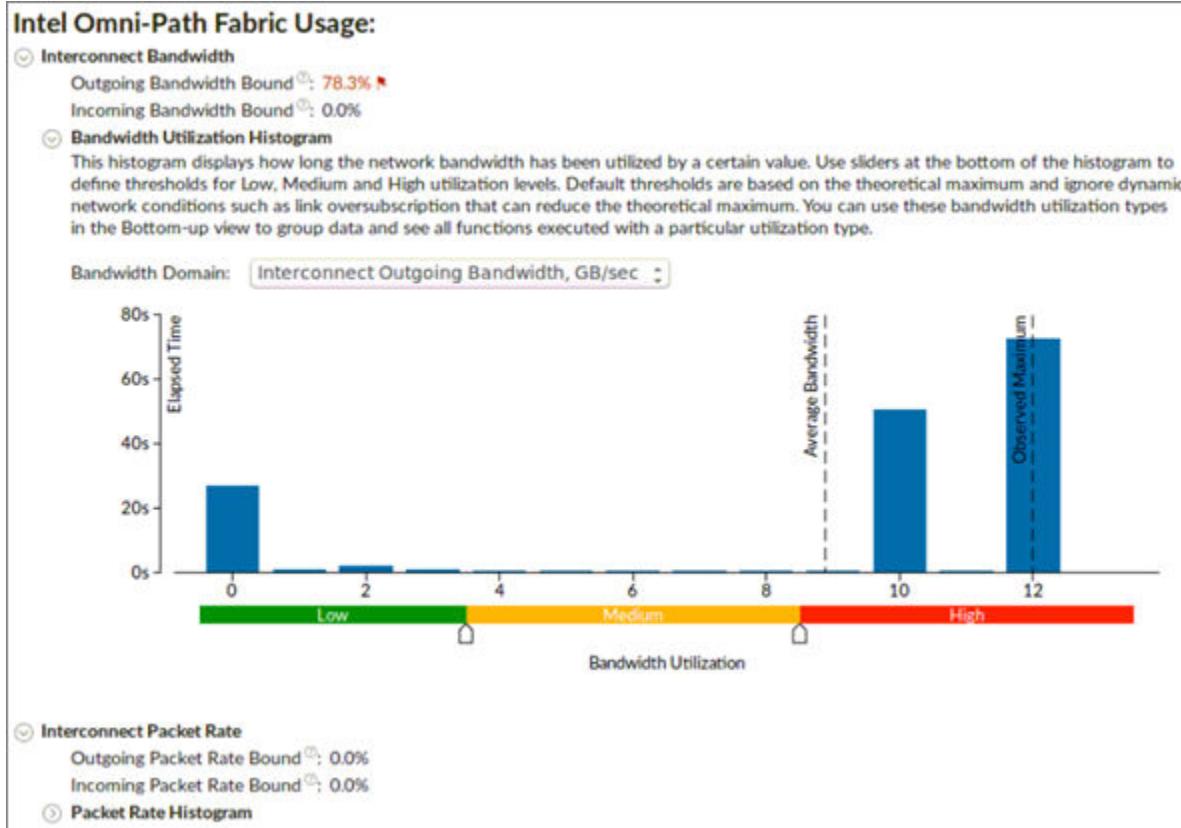
This metric shows how efficiently the application is using floating point units for vectorization. Expand the **GFLOPS** or **GFLOPS Upper Bound** (Intel Xeon Phi processors only) section to show the number of Scalar and Packed GFLOPS. This section provides a quick estimate of the amount of FLOPs that were not vectorized.

The **Top Loops/Functions with FPU Usage by CPU Time** table shows the top functions that contain floating point operations sorted by CPU time and allows for a quick estimate of the fraction of vectorized code, the vector instruction set used in the loop/function, and the loop type.

For example, if a floating point loop (function) is bandwidth bound, use the **Memory Access** analysis to resolve the bandwidth bound issue. If a floating point loop is vectorized, use the Intel Advisor to improve the vectorization. If the loop is also bandwidth bound, the bandwidth bound issue should be resolved prior to improving vectorization. Click one of the function names to switch to the **Bottom-up** window and evaluate if the function is memory bound.

Intel® Omni-Path Fabric Usage

Intel® Omni-Path Fabric (Intel® OP Fabric) metrics are available for analysis of compute nodes equipped with Intel OP Fabric interconnect. They help to understand if MPI communication has bottlenecks connected with reaching interconnect hardware limits. The section shows two aspects interconnect usage: bandwidth and packet rate. Both bandwidth and packet rate split the data into outgoing and incoming data because the interconnect is bi-directional. A bottleneck can be connected with one of the directions.



- **Outgoing and Incoming Bandwidth Bound** metrics shows the percent of elapsed time that an application spent in communication closer to or reaching interconnect bandwidth limit.
- **Bandwidth Utilization Histogram** shows how much time the interconnect bandwidth was utilized by a certain value (Bandwidth Domain) and provides thresholds to categorize bandwidth utilization as High, Medium, and Low.
- **Outgoing and Incoming Packet Rate** metrics shows the percent of elapsed time that an application spent in communication closer to or reaching interconnect packet rate limit.
- **Packet Rate Histogram** shows how much time the interconnect packet rate was reached by a certain value and provides thresholds to categorize packet rate as High, Medium, and Low.

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.
Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
CPU Information	
Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.
Logical CPU Count	Logical CPU core count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.

See Also

[HPC Performance Characterization View](#)

[Reference](#)

[Comparison Summary](#)

[Change Threshold Values](#)

Window: Summary - Memory Consumption

Use the **Summary** window as your starting point of the Memory Consumption analysis with the Intel® VTune™ Profiler and identify top memory-consuming functions and memory allocation sizes. To access this window, select the **Memory Consumption** viewpoint and click the **Summary** sub-tab in the result tab.

Depending on the analysis type, the **Summary** window provides the following application-level statistics in the **Memory Consumption** viewpoint:

- Analysis metrics
- Top Memory-Consuming Objects
- Collection and Platform Info

NOTE

Click the



Copy to Clipboard button to copy the content of the selected summary section to the clipboard.

Analysis Metrics

The first section displays the summary statistics on the overall application execution:

Elapsed Time : 1.521s	
Allocation Size:	8 GB
Deallocation Size:	8 GB
Allocations:	12,716
Total Thread Count:	1
Paused Time :	0s

All metric names are hyperlinks. Clicking such a hyperlink opens the **Bottom-up** window and sorts the data in the grid by the selected metric.

Top Memory-Consuming Objects

This section displays a list of top memory-consuming functions. For example, the `foo` function has the highest Memory Consumption metric value and could be a candidate for optimization:

Top Memory-Consuming Functions					
This section lists the most memory-consuming functions in your application.					
Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module	
<code>foo</code>	8 GB	0 B	20,000	<code>textbay</code>	
<code>_nl_load_locale_from_archive</code>	94 MB	94 MB	2	<code>libc.so.6</code>	
<code>list_resize</code>	4 MB	0 B	323	<code>python2.7</code>	
<code>data_stash_grow</code>	3 MB	0 B	3,374	<code>python2.7</code>	
<code>new_arena</code>	1 MB	1 MB	5	<code>python2.7</code>	
[Others]	3 MB	305 KB	3,567		

N/A is applied to non-callable methods.

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.

Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collector type	Type of the data collector used for the analysis. The following types are possible: <ul style="list-style-type: none"> • Driver-based sampling • Driver-less Perf*-based sampling: per-process or system-wide • User-mode sampling and tracing
CPU Information	
Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.
Logical CPU Count	Logical CPU count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.
GPU Information	
Name	Name of the Graphics installed on the system.
Vendor	GPU vendor.
Driver	Version of the graphics driver installed on the system.
Stepping	Microprocessor version.
EU Count	Number of execution units (EUs) in the Render and GPGPU engine. This data is Intel® HD Graphics and Intel® Iris® Graphics (further: Intel Graphics) specific.
Max EU Thread Count	Maximum number of threads per execution unit. This data is Intel Graphics specific.
Max Core Frequency	Maximum frequency of the Graphics processor. This data is Intel Graphics specific.
Graphics Performance Analysis	GPU metrics collection is enabled on the hardware level. This data is Intel Graphics specific.

NOTE

Some systems disable collection of extended metrics such as L3 misses, memory accesses, sampler busyness, SLM accesses, and others in the BIOS. On some systems you can set a BIOS option to enable this collection. The presence or absence of the option and its name are BIOS vendor specific. Look for the **Intel® Graphics Performance Analyzers** option (or similar) in your BIOS and set it to **Enabled**.

See Also

[Memory Consumption Analysis](#)

[Memory Consumption and Allocations View](#)

Window: Summary - Memory Usage

Use the **Summary** window as your starting point of the performance analysis with the Intel® VTune™ Profiler. To access this window, select the **Memory Usage** viewpoint and click the **Summary** sub-tab in the result tab.

Depending on the analysis type, the **Summary** window provides the following application-level statistics in the **Memory Usage** viewpoint:

- [Analysis metrics](#)
- [System Bandwidth](#)
- [Bandwidth Utilization Histogram](#)
- [Top Memory Objects](#)
- [Top Tasks](#)
- [Latency Histogram](#)
- [Collection and Platform Info](#)

NOTE

You may click the



Copy to Clipboard button to copy the content of the selected summary section to the clipboard.

Analysis Metrics

The **Summary** window displays a list of memory-related [CPU metrics](#) that help you estimate an overall memory usage during application execution. For a metric description, hover over the corresponding question mark icon



to read the pop-up help:

Elapsed Time [?]: 5.056s

CPU Time [?] :	44.458s
Memory Bound [?] :	73.5% ↗ of Pipeline Slots
L1 Bound [?] :	2.1% of Clockticks
L2 Bound [?] :	0.0% of Clockticks
L3 Bound [?] :	0.0% of Clockticks
DRAM Bound [?] :	72.6% ↗ of Clockticks
DRAM Bandwidth Bound [?] :	59.7% ↗ of Elapsed Time
Memory Latency :	
Remote / Local DRAM Ratio [?] :	0.000
Loads :	17,873,336,184
Stores :	8,390,525,856
LLC Miss Count [?] :	2,135,328,112
Average Latency (cycles) [?] :	116
Total Thread Count :	16
Paused Time [?] :	0s

Memory Bound metrics are measured either as [Clockticks](#) or as [Pipeline Slots](#). Metrics measured in Clockticks are less precise compared to the metrics measured in Pipeline Slots since they may overlap and their sum at some level does not necessarily match the parent metric value. But such metrics are still useful for identifying the dominant performance bottleneck in the code.

Mouse over a flagged value with the performance issue and read the recommendation for further analysis. For example, a high Memory Bound value typically indicates that a significant fraction of the execution pipeline slots could be stalled due to a demand memory load and stores. For further details, you may switch to the **Bottom-up** window and explore metric data per memory object.

A high DRAM Bandwidth Bound metric value indicates that your system spent much time heavily utilizing the DRAM bandwidth. The calculation of this metric relies on the accurate maximum system DRAM bandwidth measurement provided in the **System Bandwidth** section below.

System Bandwidth

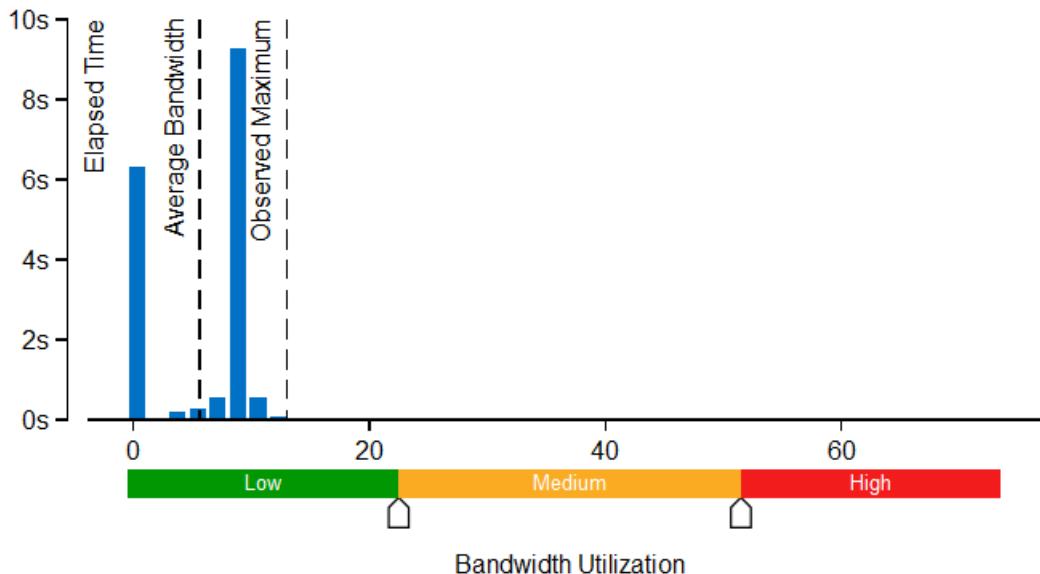
This section provides various system bandwidth-related properties detected by the product. Depending on the number of sockets on your system, the following types of system bandwidth are measured:

Max DRAM System Bandwidth	Maximum DRAM bandwidth measured for the whole system (across all packages) by running a micro-benchmark before the collection starts. If the system has already been actively loaded at the moment of collection start (for example, with the attach mode), the value may be less accurate.
Max DRAM Single-Package Bandwidth	Maximum DRAM bandwidth for single package measured by running a micro-benchmark before the collection starts. If the system has already been actively loaded at the moment of collection start (for example, with the attach mode), the value may be less accurate.

These values are used to define default High, Medium and Low bandwidth utilization thresholds for the **Bandwidth Utilization Histogram** and to scale over-time bandwidth graphs in the **Bottom-up** view. By default, for Memory Analysis results the system bandwidth is measured automatically. To enable this functionality for custom analysis results, make sure to select the **Evaluate max DRAM bandwidth** option.

Bandwidth Utilization Histogram

This histogram shows how much time the system bandwidth was utilized by a certain value (Bandwidth Domain) and provides thresholds to categorize bandwidth utilization as High, Medium and Low. You can set the threshold by moving sliders at the bottom.



If you switch to the **Bottom-up** window and group the grid data by **../Bandwidth Utilization Type/..**, you can identify functions or memory objects with high bandwidth utilization in the specific bandwidth domain.

If you select the **Interconnect** domain, you will be able to check whether the performance of your application is limited by the bandwidth of Interconnect links (inter-socket connections). Then, you may switch to the **Bottom-up** window and identify code and memory objects with NUMA issues.

Single-Package domains are displayed for the systems with two or more CPU packages and the histogram for them shows the distribution of the elapsed time per maximum bandwidth utilization *among all packages*. Use this data to identify situations where your application utilizes bandwidth only on a subset of CPU packages. In this case, the whole system bandwidth utilization represented by domains like DRAM may be low whereas the performance is in fact limited by bandwidth utilization.

NOTE

- Interconnect bandwidth analysis is supported by the VTune Profiler for Intel microarchitecture code name Ivy Bridge EP and later.
 - To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and Interconnect bandwidth.
-

Top Memory Objects by Latency (Linux* Targets Only)

If you enabled the **Analyze memory object** configuration option for the **Memory Access analysis**, the **Summary** window in the **Memory Usage** viewpoint displays memory objects (variables, data structures, arrays) that introduced the highest latency to the execution of your application.

NOTE

- Memory objects identification is supported only for Linux targets and only for processors based on Intel microarchitecture code named Haswell and newer architectures.
- Only metrics based on DLA-capable hardware events are applicable to the memory objects analysis. For example, the CPU Time metric is based on a non DLA-capable Clockticks event, so cannot be applied to memory objects. Examples of applicable metrics are Loads, Stores, LLC Miss Count, and Average Latency.

Clicking an object in the table opens the **Bottom-up** window with the grid data grouped by **Memory Object/Function/Allocation Stack**. The selected hotspot object is highlighted.

Top Tasks

This section provides a list of tasks that took most of the time to execute, where *tasks* are either code regions marked with Task API, or system tasks enabled to monitor Ftrace* events, Atrace* events, Intel Media SDK programs, OpenCL™ kernels, and so on.

Clicking a task type in the table opens the grid view (for example, Bottom-up or Event Count) grouped by the **Task Type** granularity. See [Task Analysis](#) for more information.

Latency Histogram

This histogram shows a distribution of loads per latency (in cycles).

Collection and Platform Info

This section provides the following data:

Application Command Line	Path to the target application.
Operating System	Operating system used for the collection.
Computer Name	Name of the computer used for the collection.
Result Size	Size of the result collected by the VTune Profiler.
Collection start time	Start time (in UTC format) of the external collection . Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collection stop time	Stop time (in UTC format) of the external collection. Explore the Timeline pane to track the performance statistics provided by the custom collector over time.
Collector type	Type of the data collector used for the analysis. The following types are possible: <ul style="list-style-type: none"> Driver-based sampling Driver-less Perf*-based sampling: per-process or system-wide User-mode sampling and tracing
CPU Information	
Name	Name of the processor used for the collection.
Frequency	Frequency of the processor used for the collection.

Logical CPU Count	Logical CPU count for the machine used for the collection.
Physical Core Count	Number of physical cores on the system.
User Name	User launching the data collection. This field is available if you enabled the per-user event-based sampling collection mode during the product installation.
GPU Information	
Name	Name of the Graphics installed on the system.
Vendor	GPU vendor.
Driver	Version of the graphics driver installed on the system.
Stepping	Microprocessor version.
EU Count	Number of execution units (EUs) in the Render and GPGPU engine. This data is Intel® HD Graphics and Intel® Iris® Graphics (further: Intel Graphics) specific.
Max EU Thread Count	Maximum number of threads per execution unit. This data is Intel Graphics specific.
Max Core Frequency	Maximum frequency of the Graphics processor. This data is Intel Graphics specific.
Graphics Performance Analysis	<p>GPU metrics collection is enabled on the hardware level. This data is Intel Graphics specific.</p> <p>NOTE Some systems disable collection of extended metrics such as L3 misses, memory accesses, sampler busyness, SLM accesses, and others in the BIOS. On some systems you can set a BIOS option to enable this collection. The presence or absence of the option and its name are BIOS vendor specific. Look for the Intel® Graphics Performance Analyzers option (or similar) in your BIOS and set it to Enabled.</p>

See Also

[Memory Usage View](#)

[HPC Performance Characterization View](#)

[Comparison Summary](#)

[CPU Metrics Reference](#)

[Change Threshold Values](#)

Window: Summary - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **Summary** sub-tab in the result tab.

Use the **Summary** window to:

- Begin analyzing data collected by Intel SoC Watch.
- Review the overall statistics for the collected period.
- Understand the high-level indicators of energy inefficiency.
- View basic graphs of time spent in active and sleep states.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

Depending on the options selected when running the Intel SoC Watch collector and the operating system or platform on which the analysis was run, the **Summary** window provides the following statistics in the **Platform Power Analysis** viewpoint:

- [Wake-ups/sec per Core](#)
- [Top 5 Frequencies](#)
- [Top 5 Causes of Core Wake-ups](#)
- [Top 5 Kernel Wakelocks](#)
- [Core Frequency Histogram](#)
- [Elapsed Time per Core Sleep State Histogram](#)
- [Elapsed Time per System Sleep State Histogram](#)
- [Elapsed Time per Graphics Device State Histogram](#)
- [Collection and Platform Information](#)

After reviewing the information on the **Summary** window, switch to the [Correlate Metrics](#) window to view all timeline data on one window. The **Correlate Metrics** window is another method of identifying energy trends in the collected data.

Tip

- Click the



- Copy to Clipboard** button to copy the content of the selected summary section to the clipboard.
- Click the **Details** link next to the table or graph title on the **Summary** tab to view more information about that metric in another window of the Platform Power Analysis viewpoint.

[Wake-ups/sec per Core](#)

The **Summary** window displays a list of CPU sleep state metrics that help you estimate overall energy efficiency during collection. For a metric description, hover over the corresponding question mark icon



to read the pop-up help.

Wake-ups/sec per Core : 519.231

Elapsed Time	40.125s
Available Core Time	80.249s
CPU Utilization (%)	62.290
Total Time in Non-C0 States	30.262s
Total Wake-up Count	41,668
Wake-up Count due to	Total number of core wake-ups over all cores.

A wake-up is a shift from an inactive state (Cn) to an active state (C0).

Elapsed Time	Total execution time for the collection.
Available Core Time	Total execution time across all cores (elapsed time X number of cores).
CPU Utilization (%)	Percentage of time spent in the active state (C0) during collection.
	A greater percentage time spent in the active state (C0) is an indication of higher energy consumption.
Total Time in Non-C0 States	Total time spent in sleep states (C1-Cn) across all cores. The larger the C-State number, the deeper the sleep state and the greater the energy savings.
Total Wake-up Count	Total number of wake-ups across all cores.
	A high number of wake-ups indicates that the device spent a lot of time switching from idle states to the active state. It is more energy efficient for the device to remain in either an active or inactive state than to continuously switch between idle states to the active state.
Wake-up count due to <reason>	Total number of wake-ups caused by a particular event type. For example, the total number of wake-ups caused by Clock Interrupt events.

See [Window: Core Wake-ups - Platform Power Analysis](#) for more information.

Top 5 Frequencies

View the total time and total percentage of time spent in each of the top 5 processor frequencies. The 0GHz frequency is time when the processor was inactive (in a sleep state). Switch to the **CPU C/P States** sub-tab to view more detailed information about core frequency. See [Window: CPU C/P States - Platform Power Analysis](#) for more information.

Core Frequency	Core P-States Time ^⑦	Time (%) ^⑦
0GHz	30.262s	37.7%
2.1GHz	12.698s	15.8%
2GHz	8.126s	10.1%
2.6GHz	5.117s	6.4%
1.9GHz	4.197s	5.2%
[Others]	19.850s	24.7%

Top 5 Causes of Core Wake-ups

Identifies the objects that caused the cores to wake-up the most. Objects include system events such as a clock interrupt, or a particular thread. Switch to the **Core Wake-ups** sub-tab to view more detailed information about the number of core wake-ups and wake-up reasons. See [Window: Core Wake-ups - Platform Power Analysis](#) for more information.

Wake-up Object	Total Wake-up Count ⓘ	Wake-ups %
Thread 7904	10,970	26.3%
Thread 796	8,724	20.9%
Thread 9684	3,263	7.8%
Thread 212	3,126	7.5%
Thread 436	2,719	6.5%
[Others]	12,866	30.9%

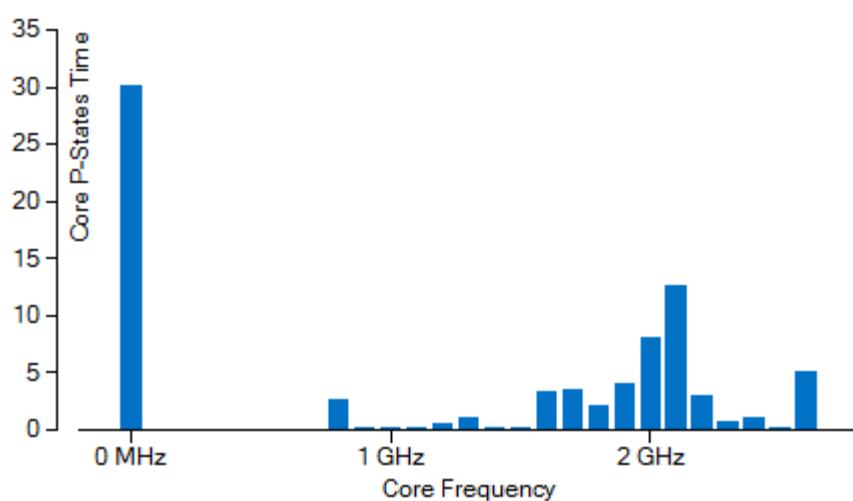
Top 5 Kernel Wakelocks

Identifies the locking processes with the most wakelock counts and longest duration. Switch to the **Wakelocks** sub-tab to view more detailed information about the kernel wakelocks, locking processes, and locking threads. See [Window: Wakelocks - Platform Power Analysis](#) for more information.

Locking Process	Kernel Wakelock Count ⓘ	Total Lock Duration %
swapper	5	100.0%
AudioTrack	1	0.0%

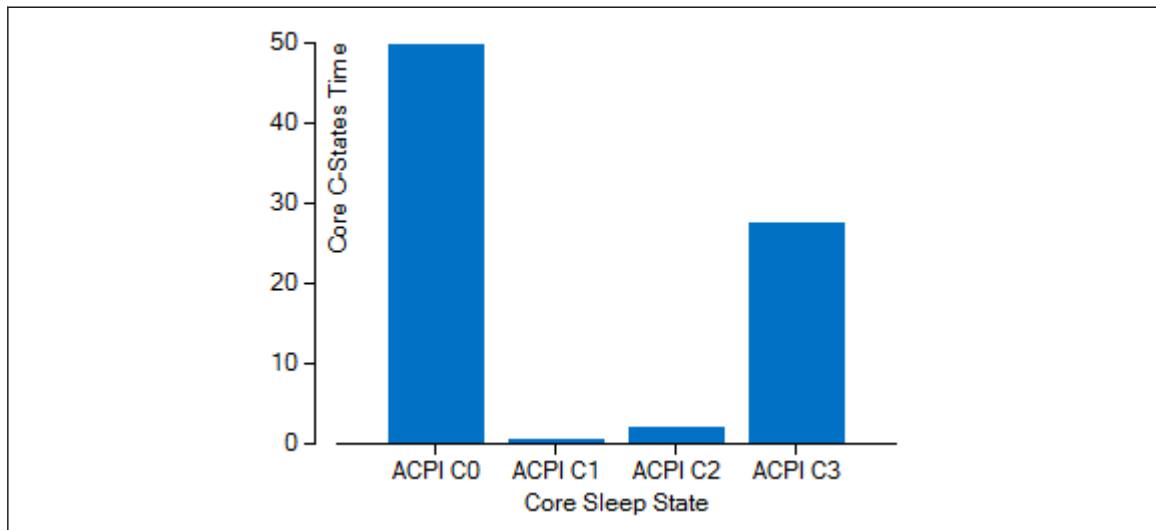
Core Frequency Histogram

Graphical representation of the amount of time spent at each frequency. Hover over a bar to see the number of seconds the CPU executed in a frequency. More than one frequency value may be represented by a single bar. Increased time spent in the high-frequencies leads to higher power consumption. Switch to the **CPU C/P States** sub-tab to view more detailed information about core frequency. See [Window: CPU C\P States - Platform Power Analysis](#) for more information.



Elapsed Time per Core Sleep State Histogram

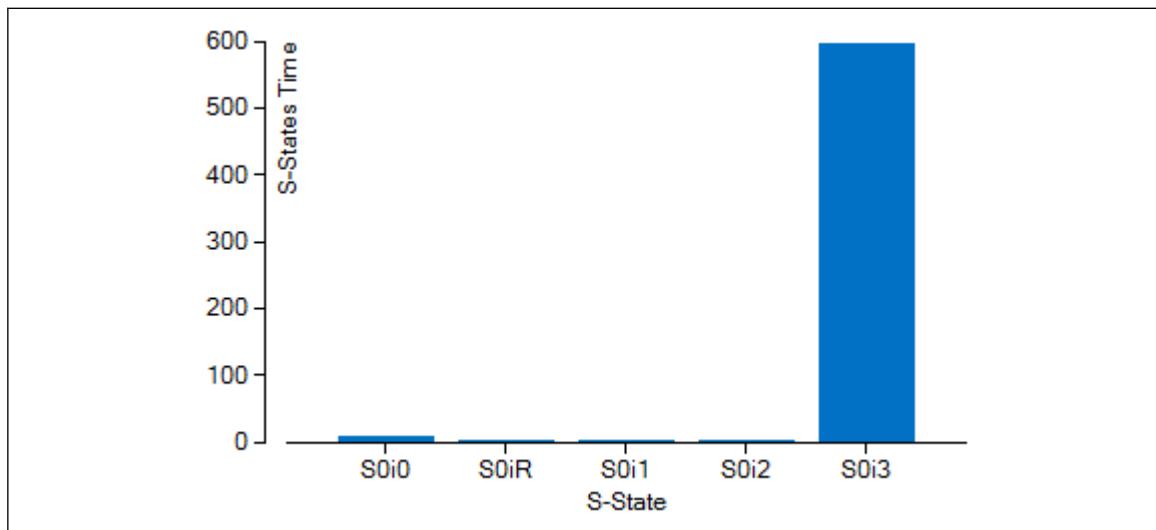
Graphical representation of the amount of time spent in each core sleep state (C-State). Hover over a bar to see the number of seconds the system spent in a specific sleep state.



Cn represents the inactive or sleep state during which the device consumes the least energy. The larger the C-State number, the deeper the sleep state. A greater amount of time spent in the C0 or active state is an indication of higher energy consumption. Switch to the **Core Wake-ups** sub-tab to view more detailed information about the reasons the cores spent time in active states and to view a timeline indicating when the cores were active. See [Window: Core Wake-ups - Platform Power Analysis](#) for more information.

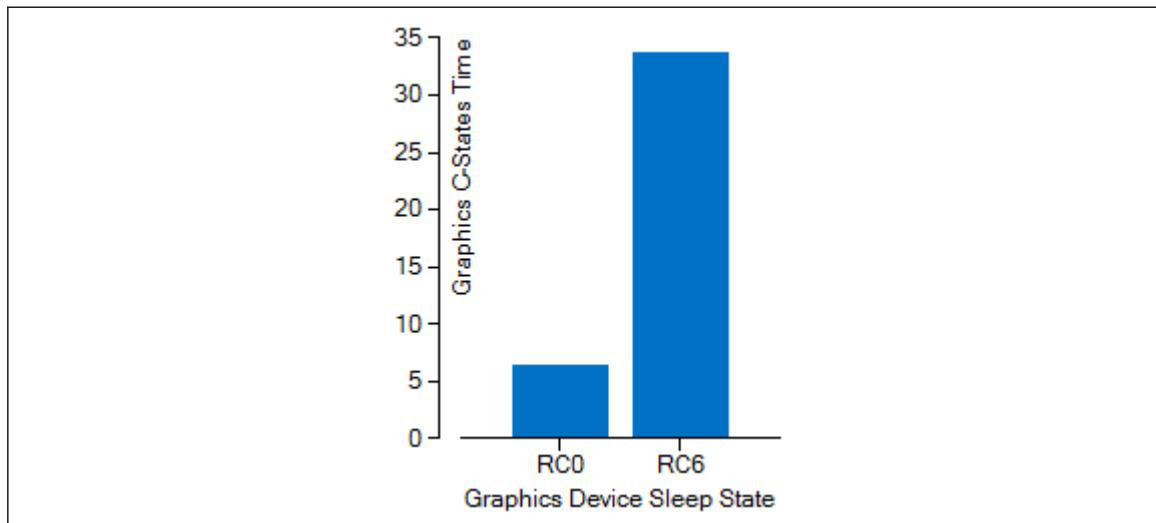
Elapsed Time per System Sleep State Histogram

Graphical representation of the amount of time spent in each system sleep state (S-State). Hover over a bar to see the number of seconds the system spent in a specific sleep state. Switch to the **System Sleep States** sub-tab to view more detailed information about the time spent in each state. See [Window: System Sleep States - Platform Power Analysis](#) for more information.



Elapsed Time per Graphics Device Sleep State Histogram

Graphical representation of the amount of time spent in each GPU sleep state (C-State). Hover over a bar to see the number of seconds your application executed in a specific sleep state. Switch to the **Graphics C/P States** sub-tab to view more detailed information about the time spent in each state, including graphics frequency changes over time. See [Window: Graphics C\P States - Platform Power Analysis](#) for more information.



Collection and Platform Info

Provides basic details about the data collected (result size) and the device on which it was collected (operating system, CPU count, core count).

Operating System:	ANDROID_5.1.1
Computer Name:	localhost
Result Size:	2 MB
Collection start time:	12:03:08 27/09/2015 UTC
Collection stop time:	12:03:38 27/09/2015 UTC
Collapsed	CPU
Frequency:	1.4 GHz
Logical CPU Count:	4
Physical Core Count:	4

See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Viewpoint](#)

[Energy Analysis Metrics](#)

[Window: Correlate Metrics - Platform Power Analysis](#)

Window: System Sleep States - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **System Sleep States** sub-tab in the result tab.

Use the **System Sleep States** window to:

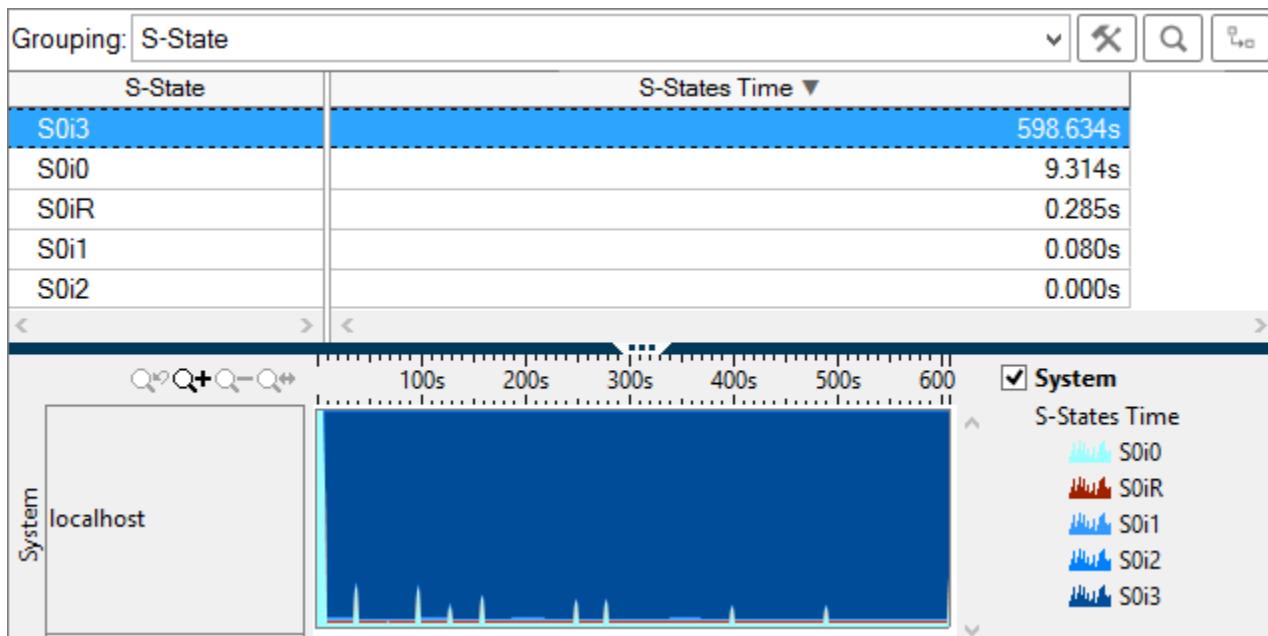
- Analyze the time spent in different ACPI-defined system sleep states (S-States).

System sleep states are available for system-on-a-chip (SoC) platforms when the display is disabled, but the system may still be active. As with other active and inactive states, the S0i0 state is the active state. The S0ix state is when the CPU is inactive, but another device is active. For example, the display may be turned

off, but audio playback is enabled, which does not allow the system to enter the deepest sleep state. The S0i3 is the deepest sleep state, which is achieved when the device is in power-saving mode (referred to as *connected standby* for Windows* devices).

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.



System Sleep States Pane

The System Sleep States pane lists the different S-States and the amount of time the system spent in each state. Click the expand

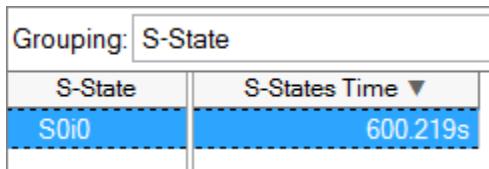


/collapse



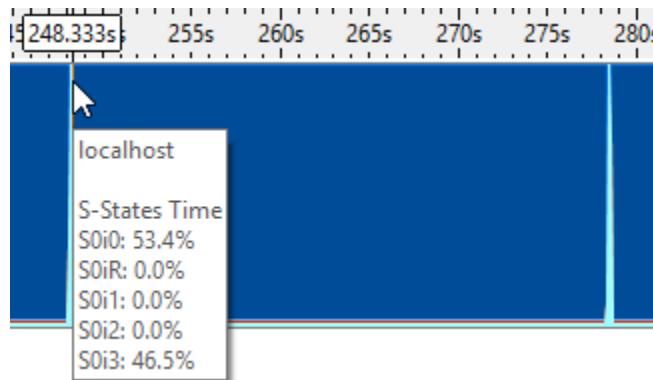
buttons in the data columns to expand the column and show data for different S-States in each device. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit. For example, you could select **Show Data As > Percent** to view the percent of collection time a particular device spent in the active state.

In the following example, the system never leaves the active S0i0 state. Either the CPU is active or one or more devices kept the system active during collection. The active devices can be identified by switching to the **NC Device States** tab or the **SC Device States** tab and looking for a device or devices that were active during the collection. Use the **CPU C/P States** window to check the CPU activity level.



Timeline Pane

The Timeline pane displays the S-States of the system at each point in time. Each state is shown in a different color. Use the legend on the right to see the colors related to the different states or features. Zoom in on the timeline to better view the transitions between inactive and active states. Hover over the timeline to view the percent of time spent in each state.



Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

[Window: Summary](#)

Window: Temperature/THERMAL Sample - Platform Power Analysis

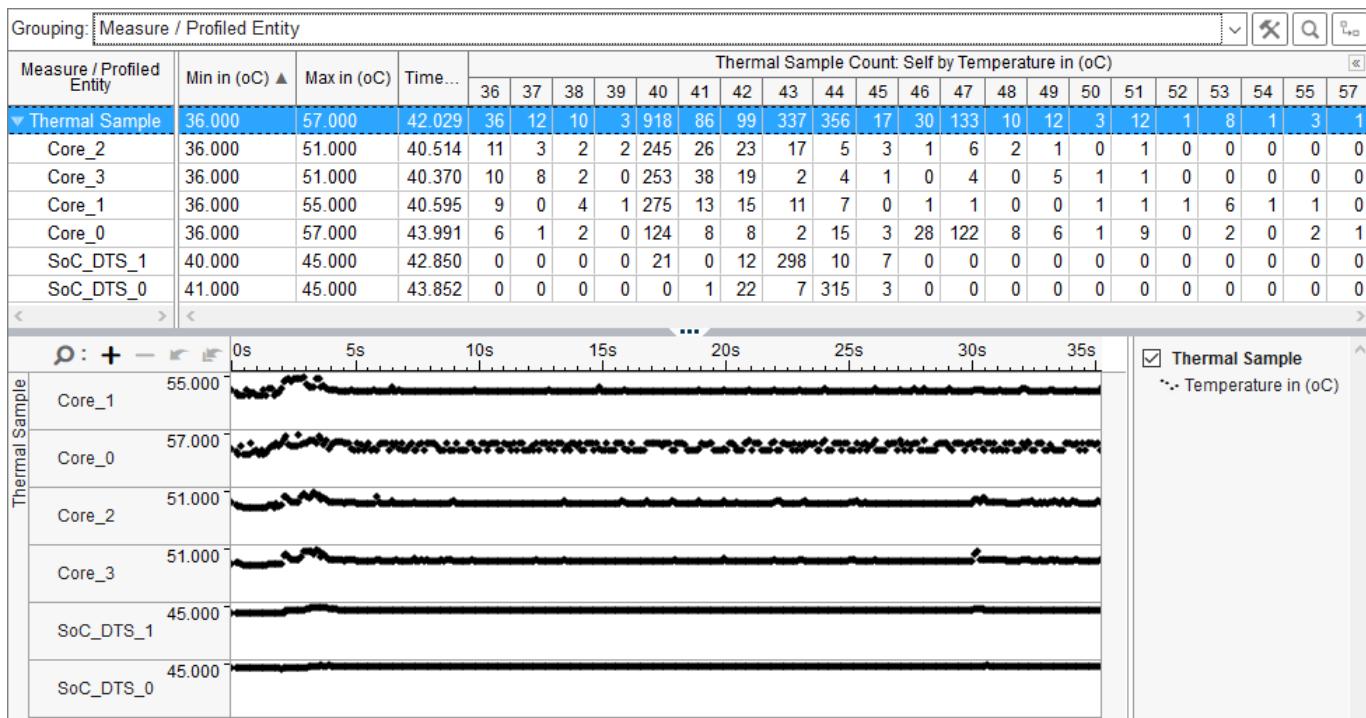
To access this window: Select the Platform Power Analysis viewpoint and click the **Temperature** (Windows*) or **Thermal Sample** (Linux*/Android*) sub-tab in the result tab.

Use this window to:

- Identify how much time each core spent in each temperature.
- Review the systems on a core (SOC) temperature samples (Intel Atom® cores only).

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.



Temperature Pane

The Temperature pane shows the sample counts at each temperature reading in degrees Celsius (°C) for each core or device. A greater number of sample counts indicates that the device or core spent more of the collection time at that temperature. Click the expand



/collapse

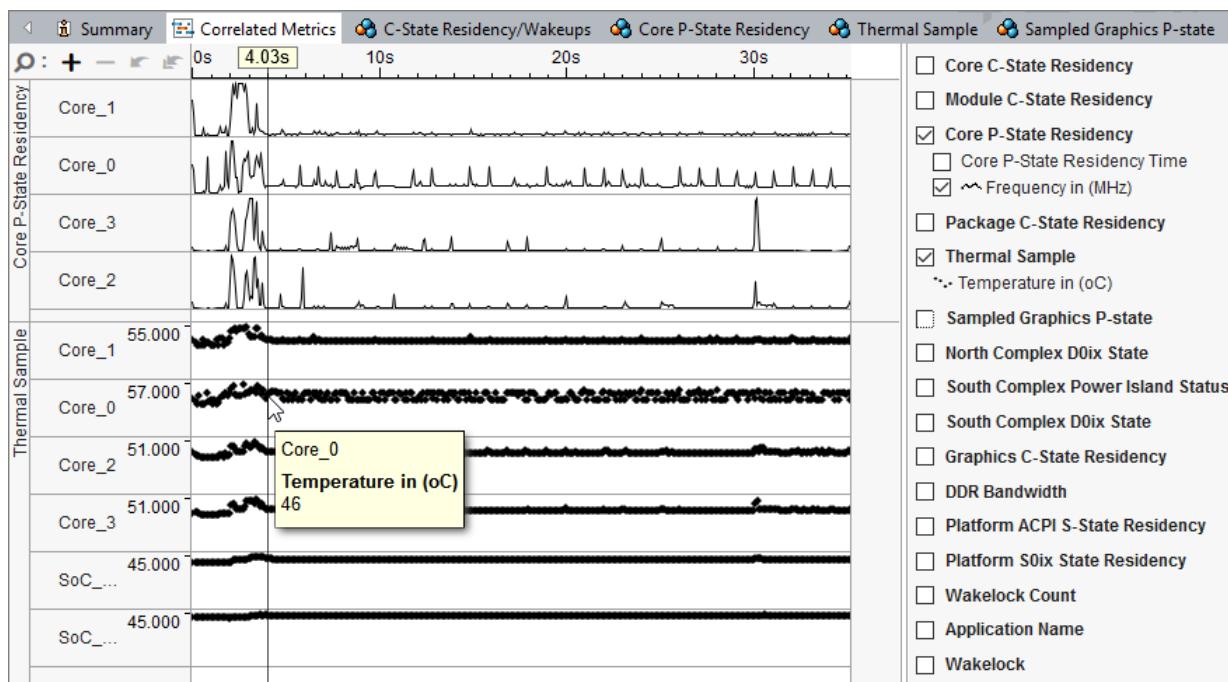


buttons in the data columns to expand the column and show data for different temperature readings in each device. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit. For example, you can display the sample counts as a percentage of the total sample counts.

Timeline Pane

The Timeline pane displays the temperatures of each core at each point in time during the collection. Expand the timeline rows vertically to view subtle temperature shifts. Zoom in on the timeline to view sampling points. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab.

Shifts in core temperature often mirror shifts in processor frequency. When the processor runs at a higher frequency, the temperature also rises. In the following example of the **Correlated Metrics** tab showing both Thermal Sample and Core P-State Frequency data, both the temperature and the frequency fluctuate for the first 4 seconds of collection and then remain fairly stable.



If the temperature is high, but the frequency is low, it could mean that the CPU is being throttled to lower core temperature.

See Also

[Interpreting Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

[Window: CPU C\P States - Platform Power Analysis](#)

Window: Timer Resolution - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **Timer Resolution** sub-tab in the result tab. The Timer Resolution window is only available for Windows* platforms.

Use the **Timer Resolution** to:

- View when timer resolution changes occurred during the collection.
- Analyze the time spent at each resolution.

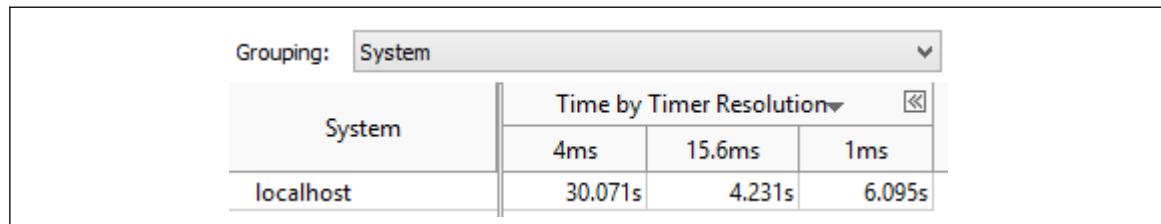
NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

Timer Resolution Pane

Total amount of time in seconds that the system spent at each timer resolution value. A value of 15.6 is the normal and recommended timer resolution in most cases. It is less energy efficient to spend a large amount of time at a lower timer resolution value because there are an increased number of wake-ups. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit. For example, you may want to change the unit to Percent to view the percent of time spent at each value and evaluate if it meets your expectations.

In the following example, most of the collection time was spent at a low timer resolution value of 4 ms. Applications generally request more frequent system timer wakeups like this to ensure a faster response. Such changes in the system timer should be restricted to critical regions in the application since it impacts the entire system. Use the Timeline pane to see which process or processes caused the change in timer resolution.



Timeline Pane

The Timeline pane shows a graphical representation of the timer resolution value changes and the duration each application spent at each resolution value.



1 Toolbar

Navigation control to zoom in/out on the view on areas of interest. Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab. For more details on the Timeline control, see [Managing Timeline View](#).

2 Legend

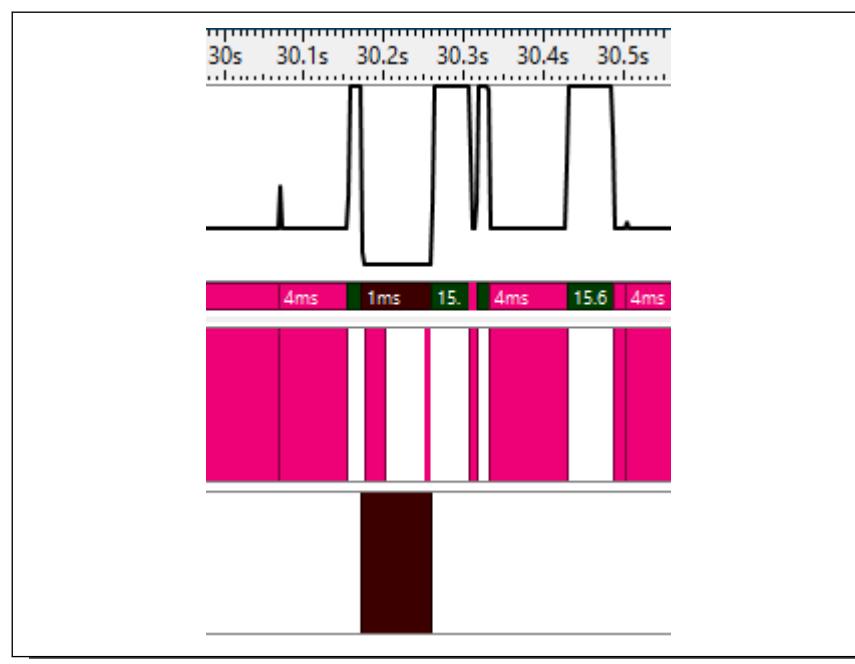
Types of data presented on the timeline. Filter in/out any type of data presented on the timeline by selecting/deselecting corresponding check boxes.

3 System Timer Resolution

Timer resolution value changes over time. The black line illustrates the change in timer resolution. The colored bar at the bottom illustrates the duration at each timer resolution value.

4 Requested/ Application Timer Resolution

Requests for timer resolution change and duration by application. Hover over the timeline to view a tooltip listing information about the request, including start time, duration, application requesting the change, and requested resolution value (ms). Zoom in on the timeline to view changes in timer resolution value.



See Also

[Interpreting Energy Analysis Data](#)

[Viewing Energy Analysis Data](#)

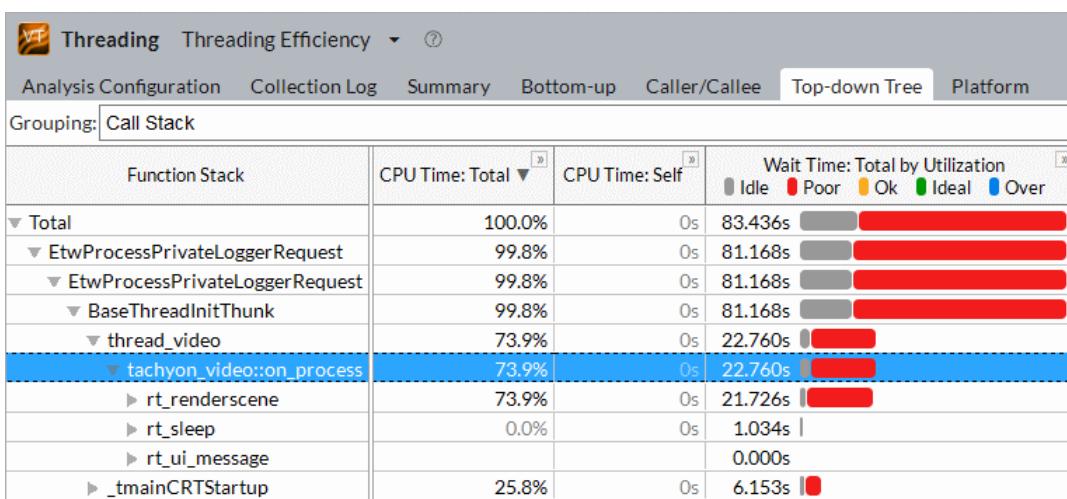
[Viewpoint](#)

[Grouping Data](#)

Window: Top-down Tree

Use the **Top-down Tree** window to explore the call sequence flow of the application and analyze the time spent in each program unit and on its callees.

To access this pane: Run a performance analysis type collecting stacks and click the **Top-down Tree** tab when the collection result opens.



Function Stack

The **Function Stack** column represents call sequences (stacks) detected during collection phase starting from the application root (usually, the `main()` function). The time value for a row is equal to the sum of all the nested items from that row. Use this data to see the impact of program units together with their callees. This type of investigation is known as a *top-down analysis*.

In this example above, the hotspot `thread_video` function has three callees, where `rt_renderscene` is the first candidate for optimization.

The call stacks are always available for the results of the user-mode sampling and tracing collection. They are also available for the results of the hardware event-based sampling collection, if you enabled the **Collect stacks** option during the analysis configuration. Otherwise, the **Function Stack** column for the event-based sampling results shows a flat list of the functions.

<Performance metrics>

Each data column in the **Top-Down Tree** grid corresponds to a [performance metric](#). The list of performance metrics varies with the analysis type and selected viewpoint. In the **Top-down Tree** window, the Intel® VTune™ Profiler provides two types of metrics:

- *Self* metrics show performance data collected within particular procedures and functions.
- *Total* metrics show performance data collected within functions AND children (callees).

By default, all program units are sorted in a descending order by the metric values in the first column (for example, **CPU Time: Total**) providing the most performance-critical program units first. You may click a column header to re-sort the table by the required metric.

NOTE

Mouse over a column header to see a metric description.

See Also[Manage Data Views](#)[User-Mode Sampling and Tracing Collection](#)[Control Window Synchronization](#)[Top-down Tree Comparison](#)**Window: Uncore Event Count - Hardware Events**

Use the **Uncore Event Count** window to analyze the event count for uncore events.

To access this window: Select the **Hardware Events** viewpoint and click the **Uncore Event Count** sub-tab in the result tab.

The **Uncore Event Count** window includes the following panes:

- [Uncore Events pane](#)
- [Timeline pane](#)

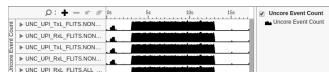
Uncore Events Pane

Uncore events happen on structures shared between all CPUs in a package (for example, 10 CPUs on a single package). This makes it impossible to associate any single uncore event with any code context and show call stacks.

By default, the uncore events are grouped by package:

Input and Output Hardware Events			
Analysis Configuration			
Collection Log			
Event Count			
Grouping	Sample Count	Uncore Event Count	
Package	UNC_M_CAS_COUNT[1] * [UNC_M_CAS_COUNTREUNIT1]	UNC_M_CAS_COUNTREUNIT2	
package_0	35,051,855	31,516,433	0
package_1			

Timeline Pane



NOTE

If there are no uncore events selected for the analysis, the **Timeline** pane is empty.

Window: Wakelocks - Platform Power Analysis

To access this window: Select the Platform Power Analysis viewpoint and click the **Wakelocks** sub-tab in the result tab. The **Wakelocks** window is available for platform power analysis on Android* target systems only.

Use the **Wakelocks** window to:

- View lock duration and locking/unlocking reasons.
- Review kernel, or system, wakelocks and understand how they change over time.
- Understand which applications cause locks, the user locking process, and the user wakelock tag.
- Analyze user and application wakelock changes over time.

NOTE

Platform Power Analysis viewpoint is available as part of energy analysis. Collecting energy analysis with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

Wakelock Pane

The Wakelock pane shows the list of wakelock objects for the user/application or kernel, depending on the grouping selected. Change the grouping selection to view data about either kernel or application/user wakelocks and the process that caused the lock or unlock. The following grouping levels and combinations of these grouping levels are available from the **Grouping** drop-down menu:

- Kernel wakelock
- Locking processes
- Application name
- User locking process
- User wakelock tag

The grid displays the sample counts for each object. Click the expand



/collapse



buttons in the data columns to expand the column and show data for different wakelock objects. By default, the table is sorted by the **Kernel Wakelock/Locking Process/Locking Thread** grouping in ascending order, which provides the objects with the highest total lock duration first. You can change the unit displayed by right-clicking a data cell and selecting the **Show Data As** option to select an alternate unit.

The following columns are available for kernel wakelocks.

Total Lock Duration	Duration of the lock in seconds.
Kernel Wakelock Count	Number of kernel wakelocks that occurred.
Wakelock Lock Count by Lock Reason	Number of wakelocks for the following reasons: Process, Existing Lock, Unknown. An Existing Lock was already started when the collection began.
Wakelock Unlock Count by Unlock Reason	The number of wakelock unlocks for the following reasons: Process, Timeout, Overwritten, Unknown. An Unknown wakelock unlock reason may mean that the wakelock continued after the collection ended.
Locking PID	Identifier of the process that caused the wakelock lock.
Locking TID	Identifier of the thread that caused the wakelock lock.

In the following example, the PowerManagerService.WakeLocks kernel wakelock had already started before the collection began and continued after the collection ended.

Grouping: Kernel Wakelock / Locking Process / Locking Thread											
Kernel Wakelock / Locking Process / Locking Thread	Total Lock Duration	Kernel Wakelock Count	Wakelock Lock Count by Lock Reason			Wakelock Unlock Count by Unlock Reason				Locking PID	Locking TID
			Process	Existing Lock	Unknown	Process	Timeout	Overwritten	Unknown		
PowerManagerService.WakeLocks	30.214s	1	0	1	0	0	0	0	1		
swapper	30.214s	1	0	1	0	0	0	0	1	0	
Thread (TID: 0)	30.214s	1	0	1	0	0	0	0	1	0	0
PowerManagerService.Display	30.214s	1	0	1	0	0	0	0	1		
vbus-intel-cht-otg.0	30.214s	1	0	1	0	0	0	0	1		
ctp_charger_wakelock	30.214s	1	0	1	0	0	0	0	1		
alarm	0.032s	1	1	0	0	1	0	0	0		
KeyEvents	0.000s	1	1	0	0	1	0	0	0		

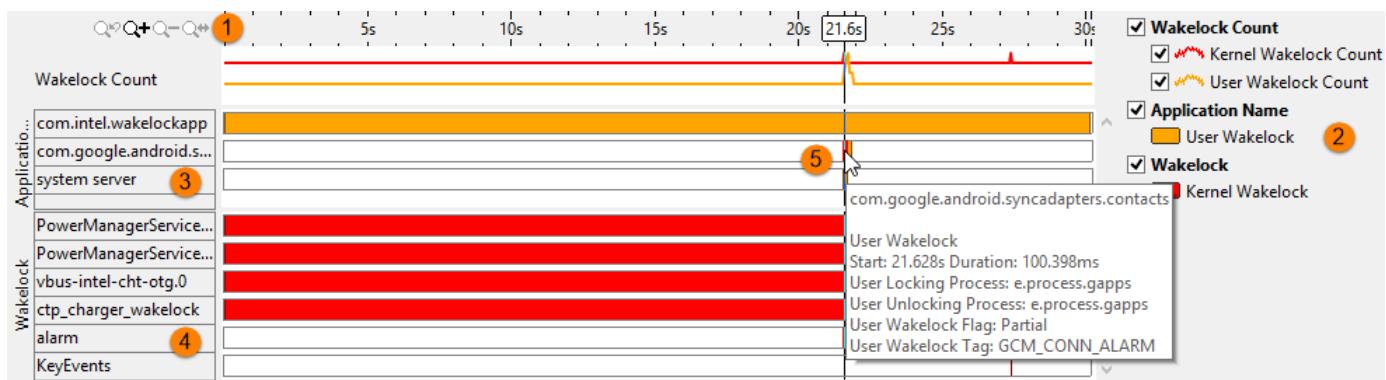
Application/user wakelocks show information for the APK name rather than the wakelock name like kernel wakelocks. The following columns are available for application/user wakelocks:

Total Lock Duration	Duration of the lock in seconds.
User Wakelock Count	Number of user wakelocks that occurred.
User Wakelock Flag	The type of wakelock.
User UID	Identifier of the application that caused the wakelock lock or unlock.
User Locking/Unlocking PID	Identifier of the application that caused the wakelock lock or unlock.

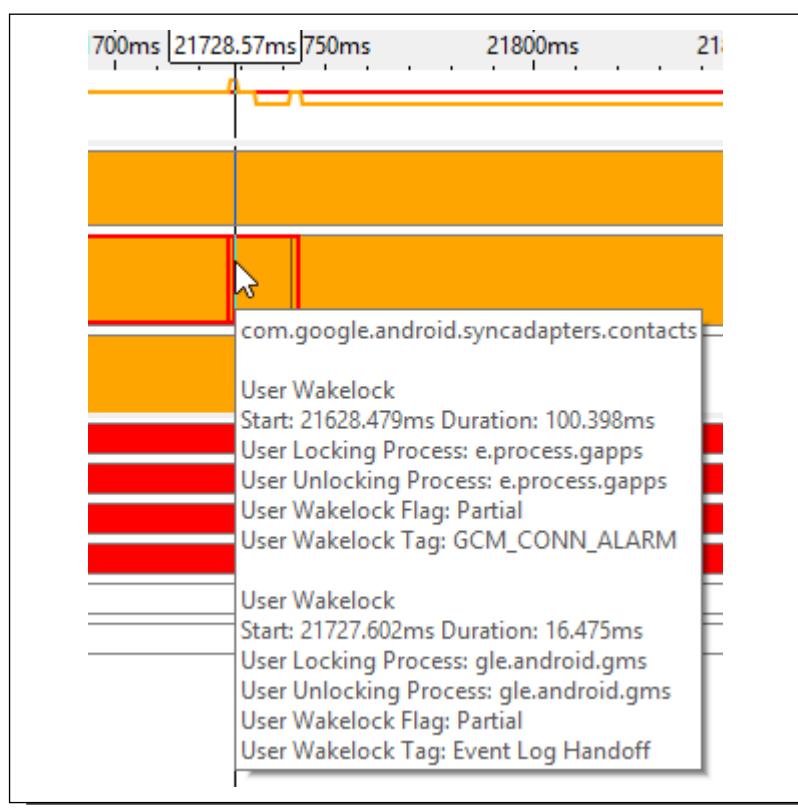
In the following example, two wakelocks originated from the com.intel.wakelockapp APK.

Grouping: Application Name / User Locking Process / User Wakelock Tag						
Application Name / User Locking Process / User Wakelock Tag	User Wakelock Count	Total Lock Duration	User UID	User Wakelock Flag	User Locking PID	
com.google.android.syncadapters.contacts	3	0.253s	10008			
com.intel.wakelockapp	2	60.428s	10112			
hwuiTask1	2	60.428s	10112		9876	
mFullWakeLock	1	30.214s	10112	Full	9876	
mPartialWakeLock	1	30.214s	10112	Partial	9876	
system server	1	0.144s	1000			

Timeline Pane



1	Toolbar	Navigation control to zoom in/out on the view on areas of interest. For more details on the Timeline control, see Managing Timeline View .
2	Legend	Types of data presented on the timeline. Filter in/out any type of data presented on the timeline by selecting/deselecting corresponding check boxes. For example, you may only be interested in the application/user wakelock data and want to remove the kernel wakelock timelines for an expanded view of the application/user wakelock data.
3	Application Name	Graphical representation of the wakelock duration for each application APK.
4	Wakelock	Graphical representation of the kernel wakelock duration through the collection time.
5	Wakelock Details	<p>Hover over the timeline of an application to view tooltips with details such as the wakelock type, start time, duration, locking and unlocking process name, and application name.</p> <p>Hover over the timeline of a kernel wakelock to view tooltips with details such as the wakelock type, start time, duration, locking and unlocking reasons, and locking process.</p> <p>Zoom in on the timeline to view the exact time when the wakelock started and when it was released. It is possible for one wakelock to begin before another ends, causing an overlap.</p>



Filters applied on a timeline in one window are applied on all other windows within the viewpoint. This is useful if you identify an issue on one tab and want to see how the issue impacts the metrics shown on a different tab. For more details on the timeline control, see [Managing Timeline View](#).

See Also

[Interpreting Energy Analysis Data](#)

[Viewpoint](#)

[Grouping Data](#)

CPU Metrics Reference

Assists

Metric Description

This metric estimates cycles fraction the CPU retired uops delivered by the Microcode_Sequencer as a result of Assists. Assists are long sequences of uops that are required in certain corner-cases for operations that cannot be handled natively by the execution pipeline. For example, when working with very small floating point values (so-called Denormals), the FP units are not set up to perform these operations natively. Instead, a sequence of instructions to perform the computation on the Denormals is injected into the pipeline. Since these microcode sequences might be hundreds of uops long, Assists can be extremely deleterious to performance and they can be avoided in many cases.

Possible Issues

A significant portion of execution time is spent in microcode assists.

Tips:

1. Examine the FP_ASSIST and OTHER_ASSISTS events to determine the specific cause.

2. Add options eliminating x87 code and set the compiler options to enable DAZ (denormals-are-zero) and FTZ (flush-to-zero).

Average Core Time

Metric Description

Total execution time over all cores.

Average Bandwidth

Metric Description

Average bandwidth utilization during the analysis.

Average CPU Frequency

Metric Description

Average actual CPU frequency. Values above nominal frequency indicate that the CPU is operating in a turbo boost mode.

Average CPU Usage

Metric Description

The metric shows average CPU utilization by computations of the application. Spin and Overhead time are not counted. Ideal average CPU usage is equal to the number of logical CPU cores.

Average Frame Time

Metric Description

Average amount of time spent within a frame.

Average Latency (cycles)

Metric Description

This metric shows average load latency in cycles

Average Logical Core Utilization

Metric Description

The metric shows average logical cores utilization by computations of the application. Spin and Overhead time are not counted. Ideal average CPU utilization is equal to the number of logical CPU cores.

Average Physical Core Utilization

Metric Description

The metric shows average physical cores utilization by computations of the application. Spin and Overhead time are not counted. Ideal average CPU utilization is equal to the number of physical CPU cores.

Average Task Time

Metric Description

Average amount of time spent within a task.

Back-End Bound

Metric Description

Back-End Bound metric represents a Pipeline Slots fraction where no uOps are being delivered due to a lack of required resources for accepting new uOps in the Back-End. Back-End is a portion of the processor core where an out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. For example, stalls due to data-cache misses or stalls due to the divider unit being overloaded are both categorized as Back-End Bound. Back-End Bound is further divided into two main categories: Memory Bound and Core Bound.

Possible Issues

A significant proportion of pipeline slots are remaining empty. When operations take too long in the back-end, they introduce bubbles in the pipeline that ultimately cause fewer pipeline slots containing useful work to be retired per cycle than the machine is capable of supporting. This opportunity cost results in slower execution. Long-latency operations like divides and memory operations can cause this, as can too many operations being directed to a single execution port (for example, more multiply operations arriving in the back-end per cycle than the execution unit can support).

Memory Bandwidth

Metric Description

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Contested Accesses (Intra-Tile)

Metric Description

Contested accesses occur when data written by one thread is read by another thread on a different core. Examples of contested accesses include synchronizations such as locks, true data sharing such as modified locked variables, and false sharing. Contested accesses metric is a ratio of the number of contested accesses to all demand loads and stores. This metrics only accounts for contested accesses between two cores on the same tile.

Possible Issues

There is a high number of contested accesses to cachelines modified by another core. Consider either using techniques suggested for other long latency load events (for example, LLC Miss) or reducing the contested accesses. To reduce contested accesses, first identify the cause. If it is synchronization, try increasing synchronization granularity. If it is true data sharing, consider data privatization and reduction. If it is false data sharing, restructure the data to place contested variables in distinct cachelines. This may increase the working set due to padding, but false sharing can always be avoided.

LLC Miss

Metric Description

The LLC (last-level cache) is the last, and longest-latency, level in the memory hierarchy before main memory (DRAM). Any memory requests missing here must be serviced by local or remote DRAM, with significant latency. The LLC Miss metric shows a ratio of cycles with outstanding LLC misses to all cycles.

Possible Issues

A high number of CPU cycles is being spent waiting for LLC load misses to be serviced. Possible optimizations are to reduce data working set size, improve data access locality, blocking and consuming data in chunks that fit in the LLC, or better exploit hardware prefetchers. Consider using software prefetchers but they can increase latency by interfering with normal loads, and can increase pressure on the memory system.

UTLB Overhead

Metric Description

This metric represents a fraction of cycles spent on handling first-level data TLB (or UTLB) misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce UTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data. This metric does not include store TLB misses.

Possible Issues

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce UTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Port Utilization

Metric Description

This metric represents a fraction of cycles during which an application was stalled due to Core non-divider-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports. Hint: Loop Vectorization - most compilers feature auto-Vectorization options today - reduces pressure on the execution ports as multiple elements are calculated with same uop.

Possible Issues

A significant fraction of cycles was stalled due to Core non-divider-related issues.

Tips

Use vectorization to reduce pressure on the execution ports as multiple elements are calculated with same uOp.

Port 0

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 0 (SNB+: ALU; HSW +:ALU and 2nd branch)

Port 1

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 1 (ALU)

Port 2

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 2 (Loads and Store-address)

Port 3

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 3 (Loads and Store-address)

Port 4

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 4 (Store-data)

Possible Issues

This metric represents Core cycles fraction CPU dispatched uops on execution port 4 (Store-data). Note that this metric value may be highlighted due to Split Stores issue.

Port 5

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 5 (SNB+: Branches and ALU; HSW+: ALU)

Port 6

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 6 (Branches and simple ALU)

Port 7

Metric Description

This metric represents Core cycles fraction CPU dispatched uops on execution port 7 (simple Store-address)

BACLEARs

Metric Description

This metric estimates a fraction of cycles lost due to the Branch Target Buffer (BTB) prediction corrected by a later branch predictor.

Possible Issues

A significant number of CPU cycles lost due to the Branch Target Buffer (BTB) prediction corrected by a later branch predictor. Consider reducing the amount of taken branches.

Bad Speculation (Cancelled Pipeline Slots)

Metric Description

Bad Speculation represents a Pipeline Slots fraction wasted due to incorrect speculations. This includes slots used to issue uOps that do not eventually get retired and slots for which the issue-pipeline was blocked due to recovery from an earlier incorrect speculation. For example, wasted work due to mispredicted branches is categorized as a Bad Speculation category. Incorrect data speculation followed by Memory Ordering Nukes is another example.

Possible Issues

A significant proportion of pipeline slots containing useful work are being cancelled. This can be caused by mispredicting branches or by machine clears. Note that this metric value may be highlighted due to Branch Resteers issue.

Bad Speculation (Back-End Bound Pipeline Slots)

Metric Description

Superscalar processors can be conceptually divided into the 'front-end', where instructions are fetched and decoded into the operations that constitute them; and the 'back-end', where the required computation is performed. Each cycle, the front-end generates up to four of these operations placed into pipeline slots that then move through the back-end. Thus, for a given execution duration in clock cycles, it is easy to determine the maximum number of pipeline slots containing useful work that can be retired in that duration. The actual number of retired pipeline slots containing useful work, though, rarely equals this maximum. This can be due to several factors: some pipeline slots cannot be filled with useful work, either because the front-end could not fetch or decode instructions in time ('Front-end bound' execution) or because the back-end was not prepared to accept more operations of a certain kind ('Back-end bound' execution). Moreover, even pipeline slots that do contain useful work may not retire due to bad speculation. Front-end bound execution may be due to a large code working set, poor code layout, or microcode assists. Back-end bound execution may be due to long-latency operations or other contention for execution resources. Bad speculation is most frequently due to branch misprediction.

Possible Issues

A significant proportion of pipeline slots are remaining empty. When operations take too long in the back-end, they introduce bubbles in the pipeline that ultimately cause fewer pipeline slots containing useful work to be retired per cycle than the machine is capable of supporting. This opportunity cost results in slower execution. Long-latency operations like divides and memory operations can cause this, as can too many operations being directed to a single execution port (for example, more multiply operations arriving in the back-end per cycle than the execution unit can support).

FP Arithmetic

Metric Description

This metric represents an overall arithmetic floating-point (FP) uOps fraction the CPU has executed (retired).

FP Assists

Metric Description

Certain floating point operations cannot be handled natively by the execution pipeline and must be performed by microcode (small programs injected into the execution stream). For example, when working with very small floating point values (so-called denormals), the floating-point units are not set up to perform these operations natively. Instead, a sequence of instructions to perform the computation on the denormal is injected into the pipeline. Since these microcode sequences might be hundreds of instructions long, these microcode assists are extremely deleterious to performance.

Possible Issues

A significant portion of execution time is spent in floating point assists.

Tips

Consider enabling the DAZ (Denormals Are Zero) and/or FTZ (Flush To Zero) options in your compiler to flush denormals to zero. This option may improve performance if the denormal values are not critical in your application. Also note that the DAZ and FTZ modes are not compatible with the IEEE Standard 754.

FP Scalar

Metric Description

This metric represents an arithmetic floating-point (FP) scalar uops fraction the CPU has executed. Analyze metric values to identify why vector code is not generated, which is typically caused by the selected algorithm or missing/wrong compiler switches.

FP Vector

Metric Description

This metric represents an arithmetic floating-point (FP) vector uops fraction the CPU has executed. Make sure vector width is expected.

FP x87

Metric Description

This metric represents a floating-point (FP) x87 uops fraction the CPU has executed. It accounts for instructions beyond X87 FP arithmetic operations; hence may be used as a thermometer to avoid X87 high usage and preferably upgrade to modern ISA. Consider compiler flags to generate newer AVX (or SSE) instruction sets, which typically perform better and feature vectors.

MS Assists

Metric Description

Certain corner-case operations cannot be handled natively by the execution pipeline and must be performed by the microcode sequencer (MS), where 1 or more uOps are issued. The microcode sequencer performs microcode assists (small programs injected into the execution stream), inserting flows, and writing to the instruction queue (IQ). For example, when working with very small floating point values (so-called denormals), the floating-point units are not set up to perform these operations natively. Instead, a sequence of instructions to perform the computation on the denormal is injected into the pipeline. Since these microcode sequences might be hundreds of instructions long, these microcode assists are extremely deleterious to performance.

Possible Issues

A significant portion of execution time is spent in microcode assists, inserted flows, and writing to the instruction queue (IQ). Examine the FP Assist and SIMD Assist metrics to determine the specific cause.

Branch Mispredict

Metric Description

When a branch mispredicts, some instructions from the mispredicted path still move through the pipeline. All work performed on these instructions is wasted since they would not have been executed had the branch been correctly predicted. This metric represents slots fraction the CPU has wasted due to Branch Misprediction. These slots are either wasted by uOps fetched from an incorrectly speculated program path, or stalls when the out-of-order part of the machine needs to recover its state from a speculative path.

Possible Issues

A significant proportion of branches are mispredicted, leading to excessive wasted work or Back-End stalls due to the machine need to recover its state from a speculative path.

Tips

1. Identify heavily mispredicted branches and consider making your algorithm more predictable or reducing the number of branches. You can add more work to 'if' statements and move them higher in the code flow for earlier execution. If using 'switch' or 'case' statements, put the most commonly executed cases first. Avoid using virtual function pointers for heavily executed calls.

2. Use profile-guided optimization in the compiler.

See the *Intel 64 and IA-32 Architectures Optimization Reference Manual* for general strategies to address branch misprediction issues.

Bus Lock

Metric Description

Intel processors provide a LOCK# signal that is asserted automatically during certain critical memory operations to lock the system bus or equivalent link. While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked. This metric measures the ratio of bus cycles, during which a LOCK# signal is asserted on the bus. The LOCK# signal is asserted when there is a locked memory access due to uncacheable memory, locked operation that spans two cache lines, and page-walk from an uncacheable page table.

Possible Issues

Bus locks have a very high performance penalty. It is highly recommended to avoid locked memory accesses to improve memory concurrency.

Tips

Examine the BUS_LOCK_CLOCKS.SELF event in the source/assembly view to determine where the LOCK# signals are asserted from. If they come from themselves, look at Back-end issues, such as memory latency or reissues. Account for skid.

Cache Bound

Metric Description

This metric shows how often the machine was stalled on L1, L2, and L3 caches. While cache hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data.

Possible Issues

A significant proportion of cycles are being spent on data fetches from caches. Check Memory Access analysis to see if accesses to L2 or L3 caches are problematic and consider applying the same performance tuning as you would for a cache-missing workload. This may include reducing the data working set size, improving data access locality, blocking or partitioning the working set to fit in the lower cache levels, or exploiting hardware prefetchers. Consider using software prefetchers, but note that they can interfere with normal loads, increase latency, and increase pressure on the memory system. This metric includes coherence penalties for shared data. Check Microarchitecture Exploration analysis to see if contested accesses or data sharing are indicated as likely issues.

Clears Resteers

Metric Description

This metric measures the fraction of cycles the CPU was stalled due to Branch Resteers as a result of Machine Clears.

Possible Issues

A significant fraction of cycles could be stalled due to Branch Resteers as a result of Machine Clears.

Clockticks per Instructions Retired (CPI)

Metric Description

Clockticks per Instructions Retired (CPI) event ratio, also known as Cycles per Instructions, is one of the basic performance metrics for the [hardware event-based sampling](#) collection, also known as Performance Monitoring Counter (PMC) analysis in the sampling mode. This ratio is calculated by dividing the number of unhalted processor cycles (Clockticks) by the number of instructions retired. On each processor the exact events used to count clockticks and instructions retired may be different, but VTune Profiler knows the correct ones to use.

What is the significance of CPI?

The CPI value of an application or function is an indication of how much latency affected its execution. Higher CPI values mean there was more latency in your system - on average, it took more clockticks for an instruction to retire. Latency in your system can be caused by cache misses, I/O, or other bottlenecks.

When you want to determine where to focus your performance tuning effort, the CPI is the first metric to check. A good CPI rate indicates that the code is executing optimally.

The main way to use CPI is by comparing a current CPI value to a baseline CPI for the same workload. For example, suppose you made a change to your system or your code and then ran the VTune Profiler and collected CPI. If the performance of the application decreased after the change, one way to understand what may have happened is to look for functions where CPI increased. If you have made an optimization that improved the runtime of your application, you can look at VTune Profiler data to see if CPI decreased. If it did, you can use that information to help direct you toward further investigations. What caused CPI to decrease? Was it a reduction in cache misses, fewer memory operations, lower memory latency, and so on.

How do I know when CPI is high?

The CPI of a workload depends both on the code, the processor, and the system configuration.

VTune Profiler analyzes the CPI value against the threshold set up by Intel architects. These numbers can be used as a general guide:

Good	Poor
0.75	4

A CPI < 1 is typical for instruction bound code, while a CPI > 1 may show up for a stall cycle bound application, also likely memory bound.

If a CPI value exceeds the threshold, the VTune Profiler highlights this value in pink.

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
► price_out_impl	78,854,118,281	62,556,093,834	1.261
► refresh_potential	64,014,096,021	17,836,026,754	3.589
► primal_bea_mpp	53,092,079,638	38,108,057,162	1.393
► update_tree	13,802,020,703	4,092,006,138	3.373
► sort_basket	12,696,019,044	12,246,018,369	1.037
► primal_iminus	11,436,017,154	5,324,007,986	2.148
► primal_net_simplex	656,000,984	266,000,399	2.466

A high value for this ratio (>1) indicates that over the current code region, instructions are taking a high number of processor clocks to execute. This could indicate a problem if most of the instructions are not predominately high latency instructions and/or coming from microcode ROM. In this case there may be opportunities to modify your code to improve the efficiency with which instructions are executed within the processor.

For processors with Intel® Hyper-Threading Technology, this ratio measures the CPI for the phases where the physical package is not in any sleep mode, that is, at least one logical processor in the physical package is in use. Clockticks are continuously counted on logical processors even if the logical processor is in a halted state (not executing instructions). This can impact the logical processors CPI ratio because the Clockticks

event continues to be accumulated while the Instructions Retired event is unchanged. A high CPI value still indicates a performance problem however a high CPI value on a specific logical processor could indicate poor CPU usage and not an execution problem.

If your application is threaded, CPI at all code levels is affected. The Clockticks event counts independently on each logical processors parallel execution is not accounted for.

For example, consider the following:

Function XYZ on logical processor 0 |-----| 4000 Clockticks / 1000 Instructions

Function XYZ on logical processor 1 |-----| 4000 Clockticks / 1000 Instructions

The CPI for the function XYZ is (8000 / 2000) 4.0. If parallel execution is taken into account in Clockticks the CPI would be (4000 / 2000) 2.0. Knowledge of the application behavior is necessary in interpreting the Clockticks event data.

What are the pitfalls of using CPI?

CPI can be misleading, so you should understand the pitfalls. CPI (latency) is not the only factor affecting the performance of your code on your system. The other major factor is the number of instructions executed (sometimes called path length). All optimizations or changes you make to your code will affect either the time to execute instructions (CPI) or the number of instructions to execute, or both. Using CPI without considering the number of instructions executed can lead to an incorrect interpretation of your results. For example, you vectorized your code and converted your math operations to operate on multiple pieces of data at once. This would have the effect of replacing many single-data math instructions with fewer multiple-data math instructions. This would reduce the number of instructions executed overall in your code, but it would likely raise your CPI because multiple-data instructions are more complex and take longer to execute. In many cases, this vectorization would increase your performance, even though CPI went up.

It is important to be aware of your total instructions executed as well. The number of instructions executed is generally called [INST_RETIRED](#) in the VTune Profiler. If your instructions retired is remaining fairly constant, CPI can be a good indicator of performance (this is the case with system tuning, for example). If both the number of instructions and CPI are changing, you need to look at both metrics to understand why your performance increased or decreased. Finally, an alternative to looking at CPI is applying the [top-down method](#).

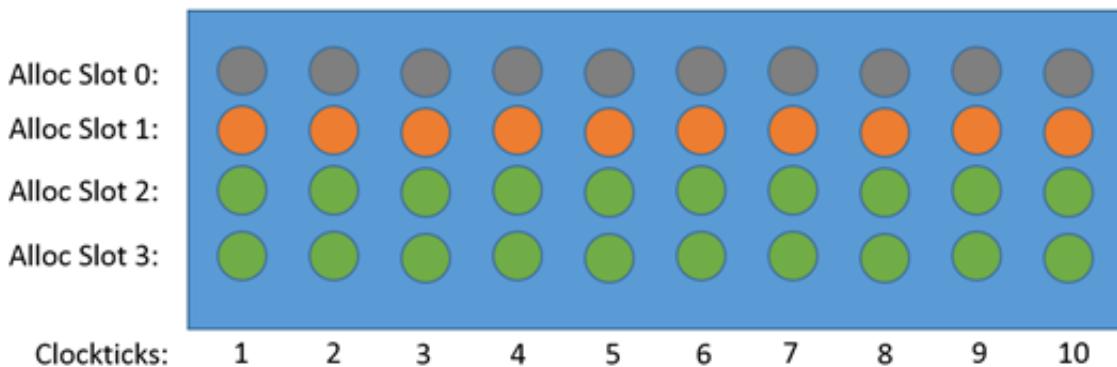
Clockticks Vs. Pipeline Slots Based Metrics

CPI Rate

Metric Description

Cycles per Instruction Retired, or CPI, is a fundamental performance metric indicating approximately how much time each executed instruction took, in units of cycles. Modern superscalar processors issue up to four instructions per cycle, suggesting a theoretical best CPI of 0.25. But various effects (long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end) tend to pull the observed CPI up. A CPI of 1 is generally considered acceptable for HPC applications but different application domains will have very different expected values. Nonetheless, CPI is an excellent metric for judging an overall potential for application performance tuning.

Intel® VTune™ Profiler provides hardware event-based metrics for the Microarchitecture Exploration analysis measured either as Clockticks or [Pipeline Slots](#). To understand the difference, consider the following example:



Here the two slots are wasted on each cycle, which is 50% in terms of Pipeline Slots. But in terms of Clockticks, the stall metrics will be 100% since on each cycle there is some stall. Moreover, on each cycle there may be stalls due to different reasons, which means that metrics measured in Clockticks may overlap.

So, metrics measured in Clockticks are less precise compared to the metrics measured in Pipeline Slots since they may overlap and their sum at some level does not necessarily match the parent metric value. But these metrics are still useful for identifying the dominant performance bottleneck in the code.

Possible Issues

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

CPI Rate (Intel Atom® processor)

Metric Description

Cycles per Instructions Retired is a fundamental performance metric indicating an average amount of time each instruction took to execute, in units of cycles. For Intel Atom processors, the theoretical best CPI per thread is 0.50, but CPI's over 2.0 warrant investigation. High CPI values may indicate latency in the system that could be reduced such as long-latency memory, floating-point operations, non-retired instructions due to branch mispredictions, or instruction starvation in the front-end. Beware that some optimizations such as SIMD will use less instructions per cycle (increasing CPI), and debug code can use redundant instructions creating more instructions per cycle (decreasing CPI).

Possible Issues

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

CPU Time

Metric Description

CPU Time is time during which the CPU is actively executing your application.

Core Bound

Metric Description

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations).

CPU Frequency

Metric Description

Frequency calculated with APERF/MPERF MSR registers captured on the clockcycles event.

It is a software frequency showing the average logical core frequency between two samples. The smaller the sampling interval is, the closer the metric is to the real HW frequency.

CPU Time

Metric Description

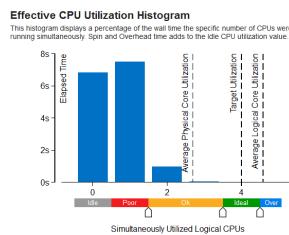
CPU Time is time during which the CPU is actively executing your application.

CPU Utilization

Metric Description

This metric evaluates the parallel efficiency of your application. It estimates the percentage of all the logical CPU cores in the system that is used by your application -- without including the overhead introduced by the parallel runtime system. 100% utilization means that your application keeps all the logical CPU cores busy for the entire time that it runs.

Depending on the analysis type, you can see the CPU Utilization data in the Bottom-up grid (HPC Performance Characterization), on the Timeline pane, and in the **Summary** window on the **Effective CPU Utilization** histogram:



Utilization Histogram

For the histogram, the Intel® VTune™ Profiler identifies a processor utilization scale, calculates the target CPU utilization, and defines default utilization ranges depending on the number of processor cores. You can change the utilization ranges by dragging the sliders, if required.

Utilization Type	Default color	Description
Idle	Grey	Idle utilization. By default, if the CPU Time on all threads is less than 0.5 of 100% CPU Time on 1 core, such CPU utilization is classified as idle. Formula: $\sum_{i=1, \text{ThreadsCount}} (\text{CPUTime}(T,i)/T) < 0.5$, where CPUTime(T,i) is the total CPU Time on thread i on interval T.
Poor	Red	Poor utilization. By default, poor utilization is when the number of simultaneously running CPUs is less than or equal to 50% of the target CPU utilization.
OK	Orange	Acceptable (OK) utilization. By default, OK usage is when the number of simultaneously running CPUs is between 51-85% of the target CPU utilization.
Ideal	Green	Ideal utilization. By default, Ideal utilization is when the number of simultaneously running CPUs is between 86-100% of the target CPU utilization.

VTune Profiler treats the [Spin](#) and [Overhead time](#) as Idle CPU utilization. Different analysis types may recognize Spin and Overhead time differently depending on availability of call stack information. This may result in a difference of CPU Utilization graphical representation per analysis type.

For the [HPC Performance Characterization](#) analysis, the VTune Profiler differentiates **Effective Physical Core Utilization** vs. **Effective Logical Core Utilization** for all systems other than Intel® Xeon Phi processors code named Knights Mill and Knights Landing.

For Intel® Xeon Phi processors code named Knights Mill and Knights Landing, as well as systems with Intel Hyper-Threading Technology (Intel HT Technology) OFF, only generic Effective CPU Utilization metric is provided.

CPU Utilization vs. Thread Efficiency

CPU Utilization may be higher than the Thread Efficiency (available for Threading analysis) if a thread is executing code on a CPU while it is logically waiting (that is, the thread is spinning).

CPU Utilization may be lower than the Thread Efficiency if:

1. The concurrency level is higher than the number of available cores (oversubscription) and, thus, reaching this level of CPU utilization is not possible. Generally, large oversubscription negatively impacts the application performance since it causes excessive context switching.
2. There was a period when the profiled process was swapped out. Thus, while it was not logically waiting, it was not scheduled for any CPU either.

Possible Issues

The metric value is low, which may signal a poor logical CPU cores utilization caused by load imbalance, threading runtime overhead, contended synchronization, or thread/process underutilization. Explore CPU Utilization sub-metrics to estimate the efficiency of MPI and OpenMP parallelism or run the Threading analysis to identify parallel bottlenecks for other parallel runtimes.

CPU Utilization (OpenMP)

Metric Description

This metric represents how efficiently the application utilized the CPUs available and helps evaluate the parallel efficiency of the application. It shows the percent of average CPU utilization by all logical CPUs on the system. Average CPU utilization contains only effective time and does not contain spin and overhead. A CPU utilization of 100% means that all of the logical CPUs were loaded by computations of the application.

Possible Issues

The metric value is low, which may signal a poor logical CPU cores utilization caused by load imbalance, threading runtime overhead, contended synchronization, or thread/process underutilization. Explore CPU Utilization sub-metrics to estimate the efficiency of MPI and OpenMP parallelism or run the Threading analysis to identify parallel bottlenecks for other parallel runtimes.

Cycles of 0 Ports Utilized

Metric Description

This metric represents a fraction of cycles with no uOps executed by the CPU on any execution port. Long-latency instructions like divides may contribute to this metric.

Possible Issues

CPU executed no uOps on any execution port during a significant fraction of cycles. Long-latency instructions like divides may contribute to this issue. Check the Assembly view and Appendix C in the Optimization Guide to identify instructions with 5 or more cycles latency.

Cycles of 1 Port Utilized

Metric Description

This metric represents cycles fraction where the CPU executed total of 1 uop per cycle on all execution ports. This can be due to heavy data-dependency among software instructions, or oversubscribing a particular hardware resource. In some other cases with high 1_Port_Utilized and L1 Bound, this metric can point to L1 data-cache latency bottleneck that may not necessarily manifest with complete execution starvation (due to the short L1 latency e.g. walking a linked list) - looking at the assembly can be helpful.

Possible Issues

This metric represents cycles fraction where the CPU executed total of 1 uop per cycle on all execution ports. This can be due to heavy data-dependency among software instructions, or oversubscribing a particular hardware resource. In some other cases with high 1_Port_Utilized and L1 Bound, this metric can point to L1 data-cache latency bottleneck that may not necessarily manifest with complete execution starvation (due to the short L1 latency e.g. walking a linked list) - looking at the assembly can be helpful. Note that this metric value may be highlighted due to L1 Bound issue.

Cycles of 2 Ports Utilized

Metric Description

This metric represents cycles fraction CPU executed total of 2 uops per cycle on all execution ports. Tip: Loop Vectorization - most compilers feature auto-Vectorization options today- reduces pressure on the execution ports as multiple elements are calculated with same uop.

Cycles of 3+ Ports Utilized

Metric Description

This metric represents Core cycles fraction CPU executed total of 3 or more uops per cycle on all execution ports.

Divider

Metric Description

Not all arithmetic operations take the same amount of time. Divides and square roots, both performed by the DIV unit, take considerably longer than integer or floating point addition, subtraction, or multiplication. This metric represents cycles fraction where the Divider unit was active.

Possible Issues

The DIV unit is active for a significant portion of execution time.

Tips

Locate the hot long-latency operation(s) and try to eliminate them. For example, if dividing by a constant, consider replacing the divide by a product of the inverse of the constant. If dividing an integer, consider using a right-shift instead.

(Info) DSB Coverage

Metric Description

Fraction of uOps delivered by the DSB (known as Decoded ICache or uOp Cache).

Possible Issues

A significant fraction of uOps was not delivered by the DSB (known as Decoded ICache or uOp Cache). This may happen if a hot code region is too large to fit into the DSB.

Tips

Consider changing the code layout (for example, via profile-guided optimization) to help your hot regions fit into the DSB.

See the "Optimization for Decoded ICache" section in the *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

DTLB Store Overhead

Metric Description

This metric represents a fraction of cycles spent on handling first-level data TLB store misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Effective CPU Utilization

Metric Description

How many of the logical CPU cores are used by your application? This metric helps evaluate the parallel efficiency of your application. It estimates the percentage of all the logical CPU cores in the system that is spent in your application -- without including the overhead introduced by the parallel runtime system. 100% utilization means that your application keeps all the logical CPU cores busy for the entire time that it runs.

Effective Physical Core Utilization

Metric Description

This metric represents how efficiently the application utilized the physical CPU cores available and helps evaluate the parallel efficiency of the application. It shows the percent of average utilization by all physical CPU cores on the system. Effective Physical Core Utilization contains only effective time and does not contain spin and overhead. An utilization of 100% means that all of the physical CPU cores were loaded by computations of the application.

Possible Issues

The metric value is low, which may signal a poor physical CPU cores utilization caused by:

- load imbalance
- threading runtime overhead
- contended synchronization
- thread/process underutilization
- incorrect affinity that utilizes logical cores instead of physical cores

Explore sub-metrics to estimate the efficiency of MPI and OpenMP parallelism or run the Locks and Waits analysis to identify parallel bottlenecks for other parallel runtimes.

Effective Time

Metric Description

Effective Time is CPU time spent in the user code. This metric does not include Spin and Overhead time.

Elapsed Time

Metric Description

Elapsed time is the wall time from the beginning to the end of collection.

Elapsed Time (Global)

Metric Description

Elapsed time is the wall time from the beginning to the end of collection.

Elapsed Time (Total)

Metric Description

Elapsed time is the wall time from the beginning to the end of collection.

Estimated BB Execution Count

Metric Description

Statistical estimation of the basic block execution count.

Estimated Ideal Time

Metric Description

Ideal Time is the estimated time for all parallel regions potentially load-balanced with zero OpenMP runtime overhead according to the formula: Total User CPU time in all regions/Number of OpenMP threads.

Execution Stalls

Metric Description

Execution stalls may signify that a machine is running at full capacity, with no computation resources wasted. Sometimes, however, long-latency operations can serialize while waiting for critical computation resources. This metric is the ratio of cycles with no micro-operations executed to all cycles.

Possible Issues

The percentage of cycles with no micro-operations executed is high. Look for long-latency operations at code regions with high execution stalls and try to use alternative methods or lower latency operations. For example, consider replacing 'div' operations with right-shifts, or try to reduce the latency of memory accesses.

False Sharing

Metric Description

This metric shows how often CPU was stalled on store operations to a shared cache line. It can be easily avoided by padding to make threads access different lines.

Far Branch

Metric Description

This metric indicates when a call/return is using a far pointer. A far call is often used to transfer from user code to privileged code.

Possible Issues

Transferring from user to privileged code may be too frequent. Consider reducing calls to system APIs.

Flags Merge Stalls

Metric Description

Shift cl operations require a potentially expensive flag merge. This metric estimates the performance penalty of that merge.

Possible Issues

A significant proportion of cycles were spent handling flags merge operations. Use the source view to discover the responsible instructions and try to avoid their use.

FPU Utilization

Metric Description

This metric represents how intensively your program uses the FPU. 100% means that the FPU is fully loaded and is retiring a vector instruction with full capacity every cycle of the application execution.

Possible Issues

The metric value is low. This can indicate poor FPU utilization because of non-vectorized floating point operations, or inefficient vectorization due to legacy vector instruction set or memory access pattern issues. Consider using vector analysis in Intel Advisor for data and tips to improve vectorization efficiency in your application.

% of Packed FP Instructions

Metric Description

This metric represents the percentage of all packed floating point instructions.

% of 128-bit Packed Floating Point Instructions

Metric Description

The metric represents % of 128-bit packed floating point instructions.

% of 256-bit Packed Floating Point Instructions

Metric Description

The metric represents % of 256-bit packed floating point instructions.

% of Packed SIMD Instructions

Metric Description

This metric represents the percentage of all packed floating point instructions.

% of Scalar FP Instructions

Metric Description

This metric represents the percentage of scalar floating point instructions.

% of Scalar SIMD Instructions

Metric Description

The metric represents the percentage of scalar SIMD instructions.

FP Arithmetic/Memory Read Instructions Ratio

Metric Description

This metric represents the ratio between arithmetic floating point instructions and memory read instructions. A value less than 0.5 indicates unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution.

FP Arithmetic/Memory Write Instructions Ratio

Metric Description

This metric represents the ratio between arithmetic floating point instructions and memory write instructions. A value less than 0.5 indicates unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution.

Loop Type

Metric Description

Displays a loop type (body, peel, reminder) based on the Intel Compiler optreport information.

SP FLOPs per Cycle

Metric Description

Number of single precision floating point operations (FLOPs) per clocktick. This metric shows the efficiency of both vector code generation and execution. Explore the list of generated issues on the metric to see the reasons behind the low FLOP numbers. The maximum number of FLOPs per cycle depends on your hardware platform. All double operations are converted to two single operations.

Vector Capacity Usage

Metric Description

This metric represents how the application code vectorization relates to the floating point computations. A value of 100% means that all floating point instructions are vectorized with the full vector capacity.

Vector Instruction Set

Metric Description

Displays the Vector Instruction Set used for arithmetic floating point computations and memory access operations.

Possible Issues

You are not using a modern vectorization instruction set. Consider recompiling your code using compiler options that allow using a modern vectorization instruction set. See the compiler User and Reference Guide for C++ or Fortran for more details.

Front-End Bandwidth

Metric Description

This metric represents a fraction of slots during which CPU was stalled due to front-end bandwidth issues, such as inefficiencies in the instruction decoders or code restrictions for caching in the DSB (decoded uOps cache). In such cases, the front-end typically delivers a non-optimal amount of uOps to the back-end.

Front-End Bandwidth DSB

Metric Description

This metric represents a fraction of cycles during which CPU was likely limited due to DSB (decoded uop cache) fetch pipeline. For example, inefficient utilization of the DSB cache structure or bank conflict when reading from it, are categorized here.

Front-End Bandwidth LSD

Metric Description

This metric represents a fraction of cycles during which CPU operation was limited by the LSD (Loop Stream Detector) unit. Typically, LSD provides good uOp supply. However, in some rare cases, optimal uOp delivery cannot be reached for small loops whose size (in terms of number of uOps) does not suit well the LSD structure.

Possible Issues

A significant number of CPU cycles is spent waiting for uOps for the LSD (Loop Stream Detector) unit. Typically, LSD provides good uOp support. However, in some rare cases, optimal uOp delivery cannot be reached for small loops whose size (in terms of number of uOps) does not suit well the LSD structure.

Front-End Bandwidth MITE

Metric Description

This metric represents a fraction of cycles during which CPU was stalled due to the MITE fetch pipeline issues, such as inefficiencies in the instruction decoders.

Front-End Bound

Metric Description

Front-End Bound metric represents a slots fraction where the processor's Front-End undersupplies its Back-End. Front-End denotes the first part of the processor core responsible for fetching operations that are executed later on by the Back-End part. Within the Front-End, a branch predictor predicts the next address to fetch, cache-lines are fetched from the memory subsystem, parsed into instructions, and lastly decoded into micro-ops (uOps). Front-End Bound metric denotes unutilized issue-slots when there is no Back-End stall (bubbles where Front-End delivered no uOps while Back-End could have accepted them). For example, stalls due to instruction-cache misses would be categorized as Front-End Bound.

Possible Issues

A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.

Tips

Make sure the code working size is not too large, the code layout does not require too many memory accesses per cycle to get enough instructions for filling four pipeline slots, or check for microcode assists.

Front-End Other

Metric Description

This metric accounts for those slots that were not delivered by the front-end and do not count as a common front-end stall.

Possible Issues

The front-end did not deliver a significant portion of pipeline slots that do not classify as a common front-end stall.

Branch Resteers

Metric Description

This metric represents cycles fraction the CPU was stalled due to Branch Resteers.

Possible Issues

A significant fraction of cycles was stalled due to Branch Resteers. Branch Resteers estimate the Front-End delay in fetching operations from corrected path, following all sorts of mispredicted branches. For example, branchy code with lots of mispredictions might get categorized as Branch Resteers. Note the value of this node may overlap its siblings.

DSB Switches

Metric Description

The Decoded Stream Buffer (DSB) cache stores uOps that have already been decoded. This helps to avoid several penalties of the legacy decode pipeline, called the MITE (Micro-instruction Translation Engine). However, when control flows out of the region cached in the DSB, the front-end incurs a penalty as uOp issue switches from the DSB to the MITE. The DSB Switches metric measures this penalty.

Possible Issues

A significant portion of cycles is spent switching from the DSB to the MITE. This may happen if a hot code region is too large to fit into the DSB.

Tips

Consider changing code layout (for example, via profile-guided optimization) to help your hot regions fit into the DSB.

See the "Optimization for Decoded ICache" section in the *Intel 64 and IA-32 Architectures Optimization Reference Manual* for more details.

ICache Misses

Metric Description

To introduce new uOps into the pipeline, the core must either fetch them from a decoded instruction cache, or fetch the instructions themselves from memory and then decode them. In the latter path, the requests to memory first go through the L1I (level 1 instruction) cache that caches the recent code working set. Front-end stalls can accrue when fetched instructions are not present in the L1I. Possible reasons are a large code working set or fragmentation between hot and cold code. In the latter case, when a hot instruction is fetched into the L1I, any cold code on its cache line is brought along with it. This may result in the eviction of other, hotter code.

Possible Issues

A significant proportion of instruction fetches are missing in the instruction cache.

Tips

1. Use profile-guided optimization to reduce the size of hot code regions.
2. Consider compiler options to reorder functions so that hot functions are located together.
3. If your application makes significant use of macros, try to reduce this by either converting the relevant macros to functions or using linker options to eliminate repeated code.
4. Consider the Os/O1 optimization level or the following subset of optimizations to decrease your code footprint:
 - Use inlining only when it decreases the footprint.
 - Disable loop unrolling.
 - Disable intrinsic inlining.

ITLB Overhead

Metric Description

In x86 architectures, mappings between virtual and physical memory are facilitated by a page table, which is kept in memory. To minimize references to this table, recently-used portions of the page table are cached in a hierarchy of 'translation look-aside buffers', or TLBs, which are consulted on every virtual address translation. As with data caches, the farther a request has to go to be satisfied, the worse the performance impact. This metric estimates the performance penalty of page walks induced on ITLB (instruction TLB) misses.

Possible Issues

A significant proportion of cycles is spent handling instruction TLB misses.

Tips

1. Use profile-guided optimization and IPO to reduce the size of hot code regions.
2. Consider compiler options to reorder functions so that hot functions are located together.
3. If your application makes significant use of macros, try to reduce this by either converting the relevant macros to functions or using linker options to eliminate repeated code.
4. For Windows targets, add function splitting.
5. Consider using large code pages.

Length Changing Prefixes

Metric Description

This metric represents a fraction of cycles during which CPU was stalled due to Length Changing Prefixes (LCPs). To avoid this issue, use proper compiler flags. Intel Compiler enables these flags by default.

Possible Issues

This metric represents a fraction of cycles during which CPU was stalled due to Length Changing Prefixes (LCPs).

Tips

To avoid this issue, use proper compiler flags. Intel Compiler enables these flags by default.

See the "Length-Changing Prefixes (LCP)" section in the Intel 64 and IA-32 Architectures Optimization Reference Manual.

MS Switches

Metric Description

This metric represents a fraction of cycles when the CPU was stalled due to switches of uop delivery to the Microcode Sequencer (MS). Commonly used instructions are optimized for delivery by the DSB or MITE pipelines. Certain operations cannot be handled natively by the execution pipeline, and must be performed by microcode (small programs injected into the execution stream). Switching to the MS too often can negatively impact performance. The MS is designated to deliver long uOp flows required by CISC instructions like CPUID, or uncommon conditions like Floating Point Assists when dealing with Denormals.

Possible Issues

A significant fraction of cycles was stalled due to switches of uOp delivery to the Microcode Sequencer (MS). Commonly used instructions are optimized for delivery by the DSB or MITE pipelines. Certain operations cannot be handled natively by the execution pipeline, and must be performed by microcode (small programs injected into the execution stream). Switching to the MS too often can negatively impact performance. The MS is designated to deliver long uOp flows required by CISC instructions like CPUID, or uncommon conditions like Floating Point Assists when dealing with Denormals. Note that this metric value may be highlighted due to Microcode Sequencer issue.

Front-End Latency

Metric Description

This metric represents a fraction of slots during which CPU was stalled due to front-end latency issues, such as instruction-cache misses, ITLB misses or fetch stalls after a branch misprediction. In such cases, the front-end delivers no uOps.

General Retirement

Metric Description

This metric represents a fraction of slots during which CPU was retiring uOps not originated from the Microcode Sequencer. This correlates with the total number of instructions executed by the program. A uOps-per-Instruction ratio of 1 is expected. While this is the most desirable of the top 4 categories, high values may still indicate areas for improvement. If possible focus on techniques that reduce instruction count or result in more efficient instructions generation such as vectorization.

Hardware Event Count

Hardware Event Sample Count

ICache Line Fetch

Metric Description

This metric estimates a fraction of cycles lost due to the instruction cacheline fetching.

Possible Issues

A significant number of CPU cycles lost due to the instruction cacheline fetching.

Ideal Time

Metric Description

Ideal Time is the estimated time for all parallel regions potentially load-balanced with zero OpenMP runtime overhead according to the formula: Total User CPU time in all regions/Number of OpenMP threads.

Imbalance or Serial Spinning

Metric Description

Imbalance or Serial Spin time is wall time when working threads are spinning on a synchronization barrier consuming CPU resources. High metric value on parallel regions can be caused by load imbalance or inefficient concurrency of all working threads. To address load imbalance, consider applying dynamic work scheduling. High metric value on serial execution (Serial - outside any region) can indicate that serial application time is significant and limiting efficient processor utilization. Explore options for parallelization, algorithm or microarchitecture tuning of the serial part of the application.

Possible Issues

CPU time spent waiting on an OpenMP barrier inside of a parallel region can be a result of load imbalance. Where relevant, try dynamic work scheduling to reduce the imbalance. High metric value on serial execution (Serial - outside any region) may signal significant serial application time that is limiting efficient processor utilization. Explore options for parallelization, algorithm or microarchitecture tuning of the serial part of the application.

Inactive Sync Wait Count

Metric Description

Inactive Sync Wait Count is the number of context switches a thread experiences when it is excluded from execution by the OS scheduler due to synchronization. Excessive number of thread context switches may negatively impact application performance. Apply optimization techniques to reduce synchronization contention and eliminate the problem.

Inactive Sync Wait Time

Metric Description

Inactive Sync Wait Time is the time when a thread remains inactive and excluded from execution by the OS scheduler due to synchronization. Significant Inactive Sync Wait Time on the critical path of an application execution, combined with a poor CPU Utilization, negatively impacts application parallelism. Explore wait stacks to identify contended synchronization objects and apply optimization techniques to reduce the contention.

Possible Issues

Average wait time per synchronization context switch is low that can signal high contended synchronization between threads or inefficient use of system API

Inactive Time

Metric Description

The time while threads were preempted by the system and remained inactive.

Inactive Wait Count

Metric Description

Inactive Wait Count is the number of context switches a thread experiences when it is excluded from execution by the OS scheduler due to either synchronization or preemption. Excessive number of thread context switches may negatively impact application performance. Reduce synchronization contention to minimize synchronization context switches, or eliminate thread oversubscription to minimize thread preemption.

Inactive Wait Time

Metric Description

Inactive Wait Time is the time when a thread remains inactive and excluded from execution by the OS scheduler due to either synchronization or preemption. Significant Inactive Wait Time on the critical path of an application execution, combined with a poor CPU Utilization, negatively impacts application parallelism. Explore wait stacks to identify contended synchronization objects and apply optimization techniques to reduce the contention.

Inactive Wait Time with poor CPU Utilization

Metric Description

Inactive Wait Time is the time when a thread remains inactive and excluded from execution by the OS scheduler due to either synchronization or preemption. Significant Inactive Wait Time on the critical path of an application execution, combined with a poor CPU Utilization, negatively impacts application parallelism. Explore wait stacks to identify contended synchronization objects and apply optimization techniques to reduce the contention.

Incoming Bandwidth Bound

Metric Description

This metric represents a percentage of elapsed time the system spent with a high incoming bandwidth utilization of the Intel Omni-Path Fabric. Note that the metric is calculated towards theoretical maximum networking bandwidth and does not take into account dynamic network conditions such as link oversubscription that can reduce the theoretical maximum.

Possible Issues

High incoming network bandwidth utilization was detected. This may lead to increased communication time. You may use Intel Trace Analyzer and Collector for communication pattern analysis.

Incoming Packet Rate Bound

Metric Description

This metric represents a percentage of elapsed time the system spent with a high incoming packet rate of the Intel Omni-Path Fabric. Explore the Packet Rate Histogram to scale the issue.

Possible Issues

High incoming network packet rate was detected. This may lead to increased communication time. You may use Intel Trace Analyzer and Collector for communication pattern analysis.

Instruction Starvation

Metric Description

A large code working set size or a high degree of branch misprediction can induce instruction delivery stalls at the front-end, such as misses in the L1I. Such stalls are called Instruction Starvation. This metric is the ratio of cycles generated when no instruction was issued by the front-end to all cycles.

Possible Issues

A significant number of CPU cycles is spent waiting for code to be delivered due to L1I misses or other problems. Look for ways to reduce the code working set, branch misprediction, and the use of virtual functions.

Interrupt Time

I/O Wait Time

Metric Description

This metric represents a portion of time when threads reside in I/O wait state while there are idle cores on the system

IPC

Metric Description

Instructions Retired per Cycle, or IPC shows average number of retired instructions per cycle. Modern superscalar processors issue up to four instructions per cycle, suggesting a theoretical best IPC of 4. But various effects (long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end) tend to pull the observed IPC down. A IPC of 1 is generally considered acceptable for HPC applications but different application domains will have very different expected values. Nonetheless, IPC is an excellent metric for judging an overall potential for application performance tuning.

Possible Issues

The IPC may be too low. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing low IPC.

L1 Bound

Metric Description

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.

Possible Issues

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

4K Aliasing

Metric Description

This metric estimates how often memory load accesses were aliased by preceding stores (in the program order) with a 4K address offset. Possible false match may incur a few cycles to re-issue a load. However, a short re-issue duration is often hidden by the out-of-order core and HW optimizations. Hence, you may safely ignore a high value of this metric unless it propagates up into parent nodes of the hierarchy (for example, to L1_Bound).

Possible Issues

A significant proportion of cycles is spent dealing with false 4k aliasing between loads and stores.

Tips

Use the source/assembly view to identify the aliasing loads and stores, and then adjust your data layout so that the loads and stores no longer alias. See the *Intel 64 and IA-32 Architectures Optimization Reference Manual* for more details.

DTLB Overhead

Metric Description

In x86 architectures, mappings between virtual and physical memory are facilitated by a page table, which is kept in memory. To minimize references to this table, recently-used portions of the page table are cached in a hierarchy of 'translation look-aside buffers', or TLBs, which are consulted on every virtual address translation. As with data caches, the farther a request has to go to be satisfied, the worse the performance impact. This metric estimates the performance penalty paid for missing the first-level data TLB (DTLB) that includes hitting in the second-level data TLB (STLB) as well as performing a hardware page walk on an STLB miss.

Possible Issues

A significant proportion of cycles is being spent handling first-level data TLB misses.

Tips

1. As with ordinary data caching, focus on improving data locality and reducing the working-set size to minimize the DTLB overhead.
2. Consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page.
3. Try using larger page sizes for large amounts of frequently-used data.

FB Full

Metric Description

This metric does a rough estimation of how often L1D Fill Buffer unavailability limited additional L1D miss memory access requests to proceed. The higher the metric value, the deeper the memory hierarchy level the misses are satisfied from. Often it hints on approaching bandwidth limits (to L2 cache, L3 cache or external memory).

Possible Issues

This metric does a rough estimation of how often L1D Fill Buffer unavailability limited additional L1D miss memory access requests to proceed. The higher the metric value, the deeper the memory hierarchy level the misses are satisfied from. Often it hints on approaching bandwidth limits (to L2 cache, L3 cache or external memory). Avoid adding software prefetches if indeed memory BW limited.

Loads Blocked by Store Forwarding

Metric Description

To streamline memory operations in the pipeline, a load can avoid waiting for memory if a prior store, still in flight, is writing the data that the load wants to read (a 'store forwarding' process). However, in some cases, generally when the prior store is writing a smaller region than the load is reading, the load is blocked for a significant time pending the store forward. This metric measures the performance penalty of such blocked loads.

Possible Issues

Loads are blocked during store forwarding for a significant proportion of cycles.

Tips

Use source/assembly view to identify the blocked loads, then identify the problematically-forwarded stores, which will typically be within the ten dynamic instructions prior to the load. If the forwarding store is smaller than the load, change the store to be the same size as the load.

Lock Latency

Metric Description

This metric represents cycles fraction the CPU spent handling cache misses due to lock operations. Due to the microarchitecture handling of locks, they are classified as L1 Bound regardless of what memory source satisfied them.

Possible Issues

A significant fraction of CPU cycles spent handling cache misses due to lock operations. Due to the microarchitecture handling of locks, they are classified as L1 Bound regardless of what memory source satisfied them. Note that this metric value may be highlighted due to Store Latency issue.

Split Loads

Metric Description

Throughout the memory hierarchy, data moves at cache line granularity - 64 bytes per line. Although this is much larger than many common data types, such as integer, float, or double, unaligned values of these or other types may span two cache lines. Recent Intel architectures have significantly improved the performance of such 'split loads' by introducing split registers to handle these cases, but split loads can still be problematic, especially if many split loads in a row consume all available split registers.

Possible Issues

A significant proportion of cycles is spent handling split loads.

Tips

Consider aligning your data to the 64-byte cache line granularity. See the *Intel 64 and IA-32 Architectures Optimization Reference Manual* for more details.

L1 Hit Rate

Metric Description

The L1 cache is the first, and shortest-latency, level in the memory hierarchy. This metric provides the ratio of demand load requests that hit the L1 cache to the total number of demand load requests.

L1D Replacement Percentage

Metric Description

When a cache line is brought into the L1 cache, another line must be evicted to make room for it. When lines in active use are evicted, a performance problem may arise from continually rotating data back into the cache. This metric measures the percentage of all replacements due to each row. For example, if the grouping is set to 'Function', this metric shows the percentage of all replacements due to each function, summing up to 100%.

Possible Issues

This row is responsible for a majority of all L1 cache replacements. Some replacements are unavoidable, and a high level of replacements may not indicate a problem. Consider this metric only when looking for the source of a significant number of L1 cache misses for a particular grouping. If these replacements are marked as a problem, try rearranging data structures (for example, moving infrequently-used data away from more-frequently-used data so that unused data is not taking up cache space) or re-ordering operations (to get as much use as possible out of data before it is evicted).

L1D Replacements

Metric Description

Replacements into the L1D

L1I Stall Cycles

Metric Description

In a shared-memory machine, instructions and data are stored in the same memory address space. However, for performance, they are cached separately. Large code working set, branch misprediction, including one caused by excessive use of virtual functions, can induce misses into L1I and so cause instruction starvation that badly influence application performance.

Possible Issues

A significant number of CPU cycles is spent waiting for code to arrive into L1I. Review application code for the patterns causing instruction starvation and rearrange the code.

L2 Bound

Metric Description

This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance.

L2 Hit Bound

Metric Description

The L2 is the last and longest-latency level in the memory hierarchy before the main memory (DRAM) or MCDRAM. While L2 hits are serviced much more quickly than hits in DRAM or MCDRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data. The L2 Hit Bound metric shows a ratio of cycles spent handling L2 hits to all cycles. The cycles spent handling L2 hits are calculated as $L2\text{ CACHE HIT COST} * L2\text{ CACHE HIT COUNT}$ where L2 CACHE HIT COST is a constant measured as typical L2 access latency in cycles.

Possible Issues

A significant proportion of cycles is being spent on data fetches that miss the L1 but hit the L2. This metric includes coherence penalties for shared data.

Tips

1. If contested accesses or data sharing are indicated as likely issues, address them first. Otherwise, consider the performance tuning applicable to an L2-missing workload: reduce the data working set size, improve data access locality, consider blocking or partitioning your working set so that it fits into the L1, or better exploit hardware prefetchers.

2. Consider using software prefetchers, but note that they can interfere with normal loads, potentially increasing latency, as well as increase pressure on the memory system.

L2 Hit Rate

Metric Description

The L2 is the last and longest-latency level in the memory hierarchy before DRAM or MCDRAM. While L2 hits are serviced much more quickly than hits in DRAM or MCDRAM, they can still incur a significant performance penalty. This metric provides a ratio of the demand load requests that hit the L2 to the total number of the demand load requests serviced by the L2. This metric does not include instruction fetches.

Possible Issues

The L2 is the last and longest-latency level in the memory hierarchy before DRAM or MCDRAM. While L2 hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric provides the ratio of demand load requests that hit the L2 to the total number of the demand load requests serviced by the L2. This metric does not include instruction fetches.

L2 HW Prefetcher Allocations

Metric Description

The number of L2 allocations caused by HW Prefetcher.

L2 Input Requests

Metric Description

A total number of L2 allocations. This metric accounts for both demand loads and HW prefetcher requests.

L2 Miss Bound

Metric Description

The L2 is the last and longest-latency level in the memory hierarchy before the main memory (DRAM) or MCDRAM. Any memory requests missing here must be serviced by local or remote DRAM or MCDRAM, with significant latency. The L2 Miss Bound metric shows a ratio of cycles spent handling L2 misses to all cycles. The cycles spent handling L2 misses are calculated as $L2\text{ CACHE MISS COST} * L2\text{ CACHE MISS COUNT}$ where L2 CACHE MISS COST is a constant measured as typical DRAM access latency in cycles.

Possible Issues

A high number of CPU cycles is being spent waiting for L2 load misses to be serviced.

Tips

1. Reduce the data working set size, improve data access locality, blocking and consuming data in chunks that fit into the L2, or better exploit hardware prefetchers.
2. Consider using software prefetchers but note that they can increase latency by interfering with normal loads, as well as increase pressure on the memory system.

L2 Miss Count

Metric Description

The L2 is the last and longest-latency level in the memory hierarchy before the main memory (DRAM) or MCDRAM. Any memory requests missing here must be serviced by local or remote DRAM or MCDRAM, with significant latency. The L2 Miss Count metric shows the total number of demand loads that missed the L2. Misses due to the HW prefetcher are not included.

L2 Replacement Percentage

Metric Description

When a cache line is brought into the L2 cache, another line must be evicted to make room for it. When lines in active use are evicted, a performance problem may arise from continually rotating data back into the cache. This metric measures the percentage of all replacements due to each row. For example, if the grouping is set to 'Function', this metric shows the percentage of all replacements due to each function, summing up to 100%.

Possible Issues

This row is responsible for a majority of all L2 cache replacements. Some replacements are unavoidable, and a high level of replacements may not indicate a problem. Consider this metric only when looking for the source of a significant number of L2 cache misses for a particular grouping. If these replacements are marked as a problem, try rearranging data structures (for example, moving infrequently-used data away from more-frequently-used data so that unused data is not taking up cache space) or re-ordering operations (to get as much use as possible out of data before it is evicted).

L2 Replacements

Metric Description

Replacements into the L2

L3 Bound

Metric Description

This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.

Contested Accesses

Metric Description

Contested accesses occur when data written by one thread is read by another thread on a different core. Examples of contested accesses include synchronizations such as locks, true data sharing such as modified locked variables, and false sharing. This metric is a ratio of cycles generated while the caching system was handling contested accesses to all cycles.

Possible Issues

There is a high number of contested accesses to cachelines modified by another core. Consider either using techniques suggested for other long latency load events (for example, LLC Miss) or reducing the contested accesses. To reduce contested accesses, first identify the cause. If it is synchronization, try increasing synchronization granularity. If it is true data sharing, consider data privatization and reduction. If it is false data sharing, restructure the data to place contested variables in distinct cachelines. This may increase the working set due to padding, but false sharing can always be avoided.

Data Sharing

Metric Description

Data shared by multiple threads (even just read shared) may cause increased access latency due to cache coherency. This metric measures the impact of that coherency. Excessive data sharing can drastically harm multithreaded performance. This metric is defined by the ratio of cycles while the caching system is handling shared data to all cycles. It does not measure waits due to contention on a variable, which is measured by the analysis.

Possible Issues

Significant data sharing by different cores is detected.

Tips

1. Examine the Contested Accesses metric to determine whether the major component of data sharing is due to contested accesses or simple read sharing. Read sharing is a lower priority than Contested Accesses or issues such as LLC Misses and Remote Accesses.
2. If simple read sharing is a performance bottleneck, consider changing data layout across threads or rearranging computation. However, this type of tuning may not be straightforward and could bring more serious performance issues back.

L3 Latency

Metric Description

This metric shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited). Avoiding private cache misses (i.e. L2 misses/L3 hits) will improve the latency, reduce contention with sibling physical cores and increase performance. Note the value of this node may overlap with its siblings.

LLC Hit

Metric Description

The LLC (last-level cache) is the last, and longest-latency, level in the memory hierarchy before main memory (DRAM). While LLC hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data.

Possible Issues

A significant proportion of cycles is being spent on data fetches that miss in the L2 but hit in the LLC. This metric includes coherence penalties for shared data.

Tips

1. If contested accesses or data sharing are indicated as likely issues, address them first. Otherwise, consider the performance tuning applicable to an LLC-missing workload: reduce the data working set size, improve data access locality, consider blocking or partitioning your working set so that it fits into the low-level cache, or better exploit hardware prefetchers.
2. Consider using software prefetchers, but note that they can interfere with normal loads, potentially increasing latency, as well as increase pressure on the memory system.

SQ Full

Metric Description

This metric measures fraction of cycles where the Super Queue (SQ) was full taking into account all request-types and both hardware SMT threads. The Super Queue is used for requests to access the L2 cache or to go out to the Uncore.

LLC Load Misses Serviced By Remote DRAM

Metric Description

In NUMA (non-uniform memory architecture) machines, memory requests missing in LLC may be serviced either by local or remote DRAM. Memory requests to remote DRAM incur much greater latencies than those to local DRAM. It is recommended to keep as much frequently accessed data local as possible. This metric is defined by the ratio of cycles when LLC load misses are serviced by remote DRAM to all cycles.

Possible Issues

A significant amount of time is spent servicing memory requests from remote DRAM. Wherever possible, try to consistently use data on the same core, or at least the same package, as it was allocated on.

LLC Miss Count

Metric Description

The LLC (last-level cache) is the last, and longest-latency, level in the memory hierarchy before main memory (DRAM). Any memory requests missing here must be serviced by local or remote DRAM, with significant latency. The LLC Miss Count metric shows total number of demand loads which missed LLC. Misses due to HW prefetcher are not included.

LLC Replacement Percentage

Metric Description

When a cache line is brought into the last-level cache, another line must be evicted to make room for it. When lines in active use are evicted, a performance problem may arise from continually rotating data back into the cache. This metric measures the percentage of all replacements due to each row. For example, if the grouping is set to 'Function', this metric shows the percentage of all replacements due to each function, summing up to 100%.

Possible Issues

This row is responsible for a majority of all last-level cache replacements. Some replacements are unavoidable, and a high level of replacements may not indicate a problem. Consider this metric only when looking for the source of a significant number of last-level cache misses for a particular grouping. If these replacements are marked as a problem, try rearranging data structures (for example, moving infrequently-used data away from more-frequently-used data so that unused data is not taking up cache space) or re-ordering operations (to get as much use as possible out of data before it is evicted).

LLC Replacements

Metric Description

Replacements into the LLC

Local DRAM Access Count

Metric Description

This metric shows the total number of LLC misses serviced by the local memory. Misses due to HW prefetcher are not included.

Logical Core Utilization

Metric Description

This metric represents how efficiently the application utilized the CPUs available and helps evaluate the parallel efficiency of the application. It shows the percent of average CPU utilization by all logical CPUs on the system.

Loop Entry Count

Metric Description

Statistical estimation of the number of times the loop was entered from the outside. Values of this metric are not aggregated per call stack filter mode.

(Info) LSD Coverage

Metric Description

This metric shows a fraction of uOps delivered by the LSD (Loop Stream Detector or Loop Cache).

Machine Clears

Metric Description

Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric measures three such events: memory ordering violations, self-modifying code, and certain loads to illegal address ranges. Machine Clears metric represents slots fraction the CPU has wasted due to Machine Clears. These slots are either wasted by uOps fetched prior to the clear, or stalls the out-of-order portion of the machine needs to recover its state after the clear.

Possible Issues

A significant portion of execution time is spent handling machine clears.

Tips

Examine the MACHINE_CLEAR events to determine the specific cause. See the "Memory Disambiguation" section in the *Intel 64 and IA-32 Architectures Optimization Reference Manual* for more details.

Max DRAM Single-Package Bandwidth

Metric Description

Maximum DRAM bandwidth for single package measured by running a micro-benchmark before the collection starts. If the system has already been actively loaded at the moment of collection start (for example, with the attach mode), the value may be less accurate.

Max DRAM System Bandwidth

Metric Description

Maximum DRAM bandwidth measured for the whole system (across all packages) by running a micro-benchmark before the collection starts. If the system has already been actively loaded at the moment of collection start (for example, with the attach mode), the value may be less accurate.

MCDRAM Bandwidth Bound

Metric Description

This metric represents percentage of elapsed time the system spent with high MCDRAM bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue.

Possible Issues

The system spent a significant percentage of elapsed time with high MCDRAM bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue. Consider improving data locality and L1/L2 cache reuse.

MCDRAM Cache Bandwidth Bound

Metric Description

This metric represents percentage of elapsed time the system spent with high MCDRAM Cache bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue.

Possible Issues

The system spent a significant percentage of elapsed time with high MCDRAM Cache bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue. Consider improving data locality and L1/L2 cache reuse.

MCDRAM Flat Bandwidth Bound

Metric Description

This metric represents percentage of elapsed time the system spent with high MCDRAM Flat bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue.

Possible Issues

The system spent a significant percentage of elapsed time with high MCDRAM Flat bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue. Consider improving data locality and/or merging compute-intensive code with bandwidth-intensive code.

Memory Bandwidth

Metric Description

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Possible Issues

A significant fraction of cycles were stalled due to approaching bandwidth limits of the main memory (DRAM).

Tips

Improve data accesses to reduce cacheline transfers from/to memory using these possible techniques:

- Consume all bytes of each cacheline before it is evicted (for example, reorder structure elements and split non-hot ones).
- Merge compute-limited and bandwidth-limited loops.
- Use NUMA optimizations on a multi-socket system.

NOTE

Software prefetches do not help a bandwidth-limited application.

Memory Bound

Metric Description

This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline.

Possible Issues

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use [Memory Access](#) analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

DRAM Bound

Metric Description

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

DRAM Bandwidth Bound

Metric Description

This metric represents percentage of elapsed time the system spent with high DRAM bandwidth utilization. Since this metric relies on the accurate peak system DRAM bandwidth measurement, explore the Bandwidth Utilization Histogram and make sure the Low/Medium/High utilization thresholds are correct for your system. You can manually adjust them, if required.

Possible Issues

The system spent much time heavily utilizing DRAM bandwidth. Improve data accesses to reduce cacheline transfers from/to memory using these possible techniques: 1) consume all bytes of each cacheline before it is evicted (for example, reorder structure elements and split non-hot ones); 2) merge compute-limited and bandwidth-limited loops; 3) use NUMA optimizations on a multi-socket system. Note: software prefetches do not help a bandwidth-limited application. Run Memory Access analysis to identify data structures to be allocated in High Bandwidth Memory (HBM), if available.

UPI Utilization Bound

Metric Description

This metric represents percentage of elapsed time the system spent with high UPI utilization. Explore the Bandwidth Utilization Histogram and make sure the Low/Medium/High utilization thresholds are correct for your system. You can manually adjust them, if required.

NOTE

The UPI Utilization metric replaced QPI Utilization starting with systems based on Intel® microarchitecture code name Skylake.

Possible Issues

The system spent much time heavily utilizing UPI bandwidth. Improve data accesses using NUMA optimizations on a multi-socket system.

Memory Latency

Metric Description

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Possible Issues

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM).

Tips

Improve data accesses or interleave them with compute using such possible techniques as data layout restructuring or software prefetches (through the compiler).

Local DRAM

Metric Description

This metric shows how often CPU was stalled on loads from local memory. Caching will improve the latency and increase performance.

Possible Issues

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote Cache

Metric Description

This metric shows how often CPU was stalled on loads from remote cache in other sockets. This is caused often due to non-optimal NUMA allocations.

Possible Issues

The number of CPU stalls on loads from the remote cache exceeds the threshold. This is often caused by non-optimal NUMA memory allocations.

Remote DRAM

Metric Description

This metric shows how often CPU was stalled on loads from remote memory. This is caused often due to non-optimal NUMA allocations.

Possible Issues

The number of CPU stalls on loads from the remote memory exceeds the threshold. This is often caused by non-optimal NUMA memory allocations.

NUMA: % of Remote Accesses

Metric Description

In Non-uniform Memory Architecture (NUMA) machines, memory requests without LLC may be serviced by local or remote DRAM. Memory requests to remote DRAM incur much greater latencies than requests to local DRAM. This metric shows the percentage of remote accesses. As far as possible, keep this metric low and frequently accessed data local. This metric does not take into account memory accesses serviced by remote cache.

Possible Issues

A significant amount of DRAM loads were serviced from remote DRAM. Wherever possible, try to consistently use data on the same core, or at least the same package, as it was allocated on.

Memory Efficiency

Metric Description

This metric represents how efficiently the memory subsystem was used by the application. It shows the percent of cycles where the pipeline was not stalled due to demand load or store instructions. The metric is based on the Memory Bound measurement.

Microarchitecture Usage

Metric Description

Microarchitecture Usage metric is a key indicator that helps estimate (in %) how effectively your code runs on the current microarchitecture. Microarchitecture Usage can be impacted by long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end.

Possible Issues

Your code efficiency on this platform is too low.

Possible cause: memory stalls, instruction starvation, branch misprediction or long latency instructions.

Tips

Run Microarchitecture Exploration analysis to identify the cause of the low microarchitecture usage efficiency.

Microcode Sequencer

Metric Description

This metric represents a fraction of slots during which CPU was retiring uOps fetched by the Microcode Sequencer (MS) ROM. The MS is used for CISC instructions not fully decoded by the default decoders (like repeat move strings), or by microcode assists used to address some modes of operation (like in Floating-Point assists).

Possible Issues

A significant fraction of cycles was spent retiring uOps fetched by the Microcode Sequencer.

Tips

1. Make sure the /arch compiler flags are correct.
2. Check the child Assists metric and, if it is highlighted as an issue, follow the provided recommendations.

Note that this metric value may be highlighted due to MS Switches issue.

Mispredicts Resteers

Metric Description

This metric measures the fraction of cycles the CPU was stalled due to Branch Resteers as a result of Branch Misprediction at execution stage.

Possible Issues

A significant fraction of cycles could be stalled due to Branch Resteers as a result of Branch Misprediction at execution stage.

MO Machine Clear Overhead

Metric Description

Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric estimates the overhead of machine clears due to Memory Ordering. The memory ordering (MO) machine clear happens when a snoop request from another processor matches a source for a data operation in the pipeline. In this situation the pipeline is cleared before the loads and stores in progress are retired. Then the pipeline is restarted from the previous retired instruction, ensuring that memory ordering of loads and stores can be preserved, both within one core and across cores. Memory ordering issues cause a severe penalty in all processors based on Intel architecture.

Possible Issues

A significant portion of execution time is spent clearing the machine to handle memory ordering requirements. To avoid this, reorder your load and store instructions, particularly loads and stores of data that is shared, or reduce sharing requirements.

MPI Imbalance

Metric Description

MPI Imbalance shows the CPU time spent by ranks spinning in waits on communication operations, normalized by the number of ranks. High metric value can be caused by application workload imbalance between ranks, nonoptimal communication schema or settings of MPI library. Explore details on communication inefficiencies with Intel Trace Analyzer and Collector.

MPI Rank on the Critical Path

Metric Description

The section contains metrics for the rank with minimum MPI Busy Wait time that is on the critical path of application execution on this node. Consider exploring CPU utilization efficiency for this rank.

MS Entry

Metric Description

This metric estimates a fraction of cycles lost due to the Microcode Sequencer entry.

Possible Issues

A significant number of CPU cycles lost due to the Microcode Sequencer entry.

MUX Reliability

Metric Description

This metric estimates reliability of HW event-related metrics. Since the number of collected HW events exceeds the number of counters, Intel® VTune™ Profiler uses event multiplexing (MUX) to share HW counters and collect different subsets of events over time. This may affect the precision of collected event data. The ideal value for this metric is 1. If the value is less than 0.7, the collected data may be not reliable.

Possible Issues

Precision of collected HW event data is not enough. Metrics data may be unreliable. Consider increasing your application execution time, using the multiple runs mode instead of event multiplexing, or creating a custom analysis with a limited subset of HW events. If you are using a driverless collection, consider reducing the value of `/sys/bys/event_source/devices/cpu/perf_event_mux_interval_ms` file.

NOTE

A high value for this metric does not guarantee an accuracy of the hardware-based metrics. However, a low value definitely puts the metrics in question and you should re-run the analysis using the **Allow multiple runs** option or increase the execution time to improve the accuracy.

OpenMP* Analysis. Collection Time

Metric Description

Collection Time is wall time from the beginning to the end of collection, excluding Paused Time.

OpenMP Region Time

Metric Description

OpenMP Region Time is a duration of all the lexical region instances.

Other

Metric Description

This metric represents a non-floating-point (FP) uop fraction the CPU has executed. If your application has no FP operations, this is likely to be the biggest fraction.

Outgoing Bandwidth Bound

Metric Description

This metric represents a percentage of elapsed time the system spent with a high outgoing bandwidth utilization of the Intel Omni-Path Fabric. Note that the metric is calculated towards theoretical maximum networking bandwidth and does not take into account dynamic network conditions such as link oversubscription that can reduce the theoretical maximum.

Possible Issues

High outgoing network bandwidth utilization was detected. This may lead to increased communication time. You may use Intel Trace Analyzer and Collector for communication pattern analysis.

Outgoing Packet Rate Bound

Metric Description

This metric represents a percentage of elapsed time the system spent with high Intel Omni-Path Fabric outgoing packet rate. Explore the Packet Rate Histogram to scale the issue.

Possible Issues

High outgoing network packet rate was detected. This may lead to increased communication time. You may use Intel Trace Analyzer and Collector for communication pattern analysis.

Overhead Time

Metric Description

Overhead time is CPU time spent on the overhead of known synchronization and threading libraries, such as system synchronization APIs, Intel® oneAPI Threading Building Blocks(oneTBB), and OpenMP.

Possible Issues

A significant portion of CPU time is spent in synchronization or threading overhead. Consider increasing task granularity or the scope of data synchronization.

Page Walk

Metric Description

In x86 architectures, mappings between virtual and physical memory are facilitated by a page table that is kept in memory. To minimize references to this table, recently-used portions of the page table are cached in a hierarchy of 'translation look-aside buffers', or TLBs, which are consulted on every virtual address translation. As with data caches, the farther a request has to go to be satisfied, the worse the performance impact is. This metric estimates the performance penalty paid for missing the first-level TLB that includes hitting in the second-level data TLB (STLB) as well as performing a hardware page walk on an STLB miss.

Possible Issues

Page Walks have a large performance penalty because they involve accessing the contents of multiple memory locations to calculate the physical address. Since this metric includes the cycles handling both instruction and data TLB misses, look at ITLB Overhead and DTLB Overhead and follow the instructions to improve performance. Also examine PAGE_WALKS.D_SIDE_CYCLES and PAGE_WALKS.I_SIDE_CYCLES events in the source/assembly view for further breakdown. Account for skid.

Parallel Region Time

Metric Description

Parallel Region Time is the total duration for all instances of all lexical parallel regions. Percent value is based on Collection Time.

Paused Time

Metric Description

Paused time is the amount of Elapsed time during which the analysis was paused using either the GUI, CLI commands, or user API.

Persistent Memory Bound

Metric Description

This metric estimates how frequently the CPU was stalled on accesses to external Intel Optane DC Persistent Memory by loads. This metric is defined on machines with Intel Optane DC Persistent Memory App Direct Mode.

Pipeline Slots

Metric Description

A pipeline slot represents hardware resources needed to process one uOp.

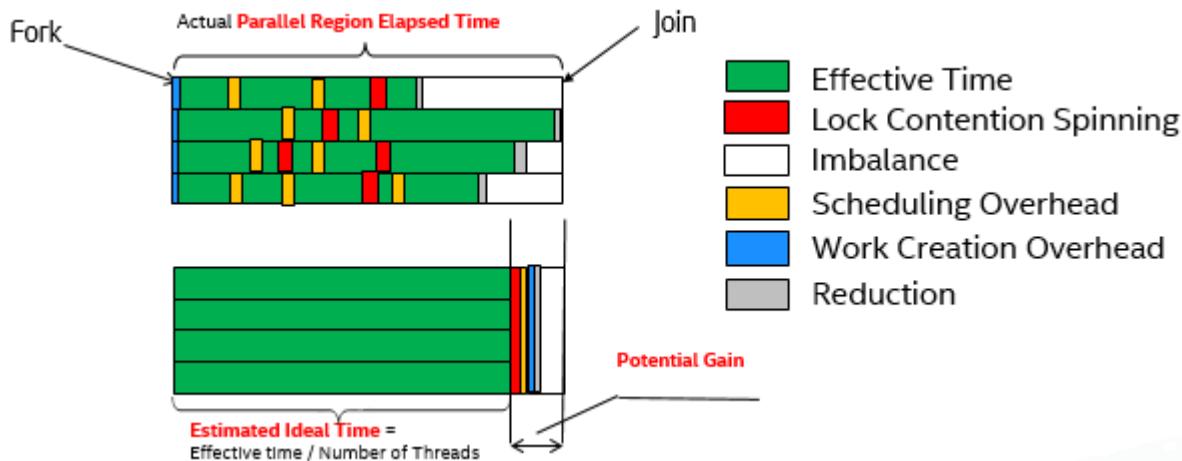
The [Top-Down Characterization](#) assumes that for each CPU core, on each clock cycle, there are several pipeline slots available. This number is called *Pipeline Width*.

OpenMP* Potential Gain

Metric Description

Potential Gain shows the maximum time that could be saved if the OpenMP region is optimized to have no load imbalance assuming no runtime overhead (Parallel Region Time minus Region Ideal Time). If the Potential Gain is large, make sure the workload for this region is enough and the loop schedule is optimal.

VTune Profiler uses the following methodology to calculate the Potential Gain metric that is a sum of inefficiencies normalized by the number of threads:



Possible Issues

The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is optimal.

Imbalance

Metric Description

OpenMP Potential Gain Imbalance shows maximum elapsed time that could be saved if the OpenMP construct is optimized to have no imbalance. It is calculated as summary of CPU time by all OpenMP threads spinning on a barrier divided by the number of OpenMP threads.

Possible Issues

Significant time spent waiting on an OpenMP barrier inside of a parallel region can be a result of load imbalance. Consider using dynamic work scheduling to reduce the imbalance, where possible.

Lock Contention

Metric Description

OpenMP Potential Gain Lock Contention shows elapsed time cost of OpenMP locks and ordered synchronization. High metric value may signal inefficient parallelization with highly contended synchronization objects. To avoid intensive synchronization, consider using reduction, atomic operations or thread local variables where possible. This metric is based on CPU sampling and does not include passive waits.

Possible Issues

When synchronization objects are used inside a parallel region, threads can spend CPU time waiting on a lock release, contending with other threads for a shared resource. Where possible, reduce synchronization by using reduction or atomic operations, or minimize the amount of code executed inside the critical section.

Pre-Decode Wrong

Metric Description

This metric estimates a fraction of cycles lost due to the decoder predicting wrong instruction length.

Possible Issues

A significant number of CPU cycles lost due to the decoder predicting wrong instruction length.

Remote Cache Access Count

Metric Description

This metric shows the total number of LLC misses serviced by the remote cache in other sockets. Misses due to HW prefetcher are not included.

Remote DRAM Access Count

Metric Description

This metric shows the total number of LLC misses serviced by the remote memory. Misses due to HW prefetcher are not included.

Remote / Local DRAM Ratio

Metric Description

In NUMA (non-uniform memory architecture) machines, memory requests missing LLC may be serviced either by local or remote DRAM. Memory requests to remote DRAM incur much greater latencies than those to local DRAM. It is recommended to keep as much frequently accessed data local as possible. This metric is defined by the ratio of remote DRAM loads to local DRAM loads.

Possible Issues

A significant amount of DRAM loads were serviced from remote DRAM. Wherever possible, try to consistently use data on the same core, or at least the same package, as it was allocated on.

Retire Stalls

Metric Description

This metric is defined as a ratio of the number of cycles when no micro-operations are retired to all cycles. In the absence of performance issues, long latency operations, and dependency chains, retire stalls are insignificant. Otherwise, retire stalls result in a performance penalty.

Possible Issues

A high number of retire stalls is detected. This may result from branch misprediction, instruction starvation, long latency operations, and other issues. Use this metric to find where you have stalled instructions. Once you have located the problem, analyze metrics such as LLC Miss, Execution Stalls, Remote Accesses, Data Sharing, and Contested Accesses, or look for long-latency instructions like divisions and string operations to understand the cause.

Retiring

Metric Description

Retiring metric represents a Pipeline Slots fraction utilized by useful work, meaning the issued uOps that eventually get retired. Ideally, all Pipeline Slots would be attributed to the Retiring category. Retiring of 100% would indicate the maximum possible number of uOps retired per cycle has been achieved. Maximizing Retiring typically increases the Instruction-Per-Cycle metric. Note that a high Retiring value does not necessarily mean no more room for performance improvement. For example, Microcode assists are categorized under Retiring. They hurt performance and can often be avoided.

Possible Issues

A high fraction of pipeline slots was utilized by useful work.

Tips

While the goal is to make this metric value as big as possible, a high Retiring value for non-vectorized code could prompt you to consider code vectorization. Vectorization enables doing more computations without significantly increasing the number of instructions, thus improving the performance. Note that this metric value may be highlighted due to Microcode Sequencer (MS) issue, so the performance can be improved by avoiding using the MS.

Self Time and Total Time

Self Time

Self time is the time spent in a particular program unit. For example, Self time for a source line shows the time the application spent at this particular source line. Self time can help you understand the impact that a function has on the program. Investigating the impact of single functions is also known as bottom-up analysis.

For example, in a single-threaded program with negligible Wait time, the Self time for the function `foo()` is 10% of the program CPU time. If you optimize `foo()` so that it is twice as fast, the Elapsed time for the program improves by 5%.

The impact of Self time on the Elapsed time of a parallel application depends on the utilization of different threads. Reducing the time that a given function runs to zero may have no impact on the Elapsed time of the application.

Total Time

Total time is the accumulated time that a program unit incurs. For functions, Total time includes Self time of the function itself and Self time of all functions that were called from that function. Total time enables a high-level understanding of how time is spent in the application. Investigating the impact of functions together with their callees is also known as top-down analysis.

Serial CPU Time

Metric Description

Serial CPU Time is the *CPU* time (compare with [Serial Time outside parallel regions](#)) spent by the application outside any OpenMP region in the master thread during collection. It directly impacts application Collection Time and scaling. High values signal a performance problem to be solved via code parallelization or algorithm tuning.

MPI Busy Wait Time

Metric Description

MPI Busy Wait Time is CPU time when MPI runtime library is spinning on waits in communication operations. High metric value can be caused by load imbalance between ranks, active communications or nonoptimal settings of MPI library. Explore details on communication inefficiencies with Intel® Trace Analyzer and Collector.

Possible Issues

CPU time spent on waits for MPI communication operations is significant and can negatively impact the application performance and scalability. This can be caused by load imbalance between ranks, active communications or non-optimal settings of MPI library. Explore details on communication inefficiencies with Intel Trace Analyzer and Collector.

Other

Metric Description

This metric shows unclassified [Serial CPU Time](#).

Serial Time (outside parallel regions)

Metric Description

Serial Time is the time spent by the application outside any OpenMP region in the master thread during collection. It directly impacts application Collection Time and scaling. High values signal a performance problem to be solved via code parallelization or algorithm tuning.

Possible Issues

Serial Time of your application is high. It directly impacts application Elapsed Time and scalability. Explore options for parallelization, algorithm or microarchitecture tuning of the serial part of the application.

SIMD Assists

Metric Description

SIMD assists are invoked when an EMMS instruction is executed after MMX technology code has changed the MMX state in the floating point stack. The EMMS instruction clears the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions, which can incur a performance penalty when intermixing MMX and

X87 instructions. The SIMD assists are required in the streaming SIMD Extensions (SSE) instructions with denormal input when the DAZ (Denormals Are Zeros) flag is off or underflow result when the FTZ (Flush To Zero) flag is off.

Possible Issues

A significant portion of execution time is spent in SIMD assists. Consider enabling the DAZ (Denormals Are Zero) and/or FTZ (Flush To Zero) options in your compiler to flush denormals to zero. This option may improve performance if the denormal values are not critical in your application. Also note that the DAZ and FTZ modes are not compatible with the IEEE Standard 754.

SIMD Compute-to-L1 Access Ratio

Metric Description

This metric provides the ratio of SIMD compute instructions to the total number of memory loads, each of which will first access the L1 cache. On this platform, it is important that this ratio is large to ensure efficient usage of compute resources.

SIMD Compute-to-L2 Access Ratio

Metric Description

This metric provides the ratio of SIMD compute instructions to the total number of memory loads that hit the L2 cache. On this platform, it is important that this ratio is large to ensure efficient usage of compute resources.

SIMD Instructions per Cycle

Metric Description

This metric represents how intensively your program uses the FPU. 100% means that the FPU is fully loaded and is retiring a vector instruction with full capacity every cycle of the application execution.

Slow LEA Stalls

Metric Description

Some Load Effective Address (LEA) instructions (like three-operand LEA instructions) have increased latency and reduced dispatch port choices. The Slow LEA Stalls metric estimates the performance penalty of such slow LEAs.

Possible Issues

A significant proportion of cycles were spent handling slow LEA operations. Use the source view to discover the responsible instructions and try to avoid their use.

SMC Machine Clear

Metric Description

Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric measures only self-modifying code (SMC) events. This event counts the number of times a program writes to a code section that is shared with another processor or itself as a data page, causing the entire pipeline and the trace caches to be cleared. Self-modifying code causes a severe penalty in all processors based on Intel architecture.

Possible Issues

A significant portion of execution time is spent handling machine clears incurred by self-modifying code event. Dynamically-modified code (for example, target fix-ups) is likely to suffer from performance degradation due to SMC. To avoid this, introduce indirect branches and use data tables on data pages (not code pages) with register-indirect calls.

SP FLOPs per Cycle

Metric Description

Number of single precision floating point operations (FLOPs) per clocktick. This metric shows the efficiency of both vector code generation and execution. Explore the list of generated issues on the metric to see the reasons behind the low FLOP numbers. The maximum number of FLOPs per cycle depends on your hardware platform. All double operations are converted to two single operations.

SP GFLOPS

Metric Description

Number of single precision giga-floating point operations calculated per second. All double operations are converted to two single operations.

Spin Time

Metric Description

Spin time is Wait Time during which the CPU is busy. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting. Some Spin Time may be preferable to the alternative of increased thread context switches. Too much Spin Time, however, can reflect lost opportunity for productive work.

Possible Issues

A significant portion of CPU time is spent waiting. Use this metric to discover which synchronizations are spinning. Consider adjusting spin wait parameters, changing the lock implementation (for example, by backing off then descheduling), or adjusting the synchronization granularity.

Communication (MPI)

Metric Description

MPI Busy Wait Time is CPU time when MPI runtime library is spinning on waits in communication operations. High metric value can be caused by load imbalance between ranks, active communications or nonoptimal settings of MPI library. Explore details on communication inefficiencies with Intel Trace Analyzer and Collector.

Possible Issues

CPU time spent on waits for MPI communication operations is significant and can negatively impact the application performance and scalability. This can be caused by load imbalance between ranks, active communications or non-optimal settings of MPI library. Explore details on communication inefficiencies with Intel Trace Analyzer and Collector.

Imbalance or Serial Spinning

Metric Description

Imbalance or Serial Spinning time is CPU time when working threads are spinning on a synchronization barrier consuming CPU resources. This can be caused by load imbalance, insufficient concurrency for all working threads or waits on a barrier in the case of serialized execution.

Possible Issues

The threading runtime function related to time spent on imbalance or serial spinning consumed a significant amount of CPU time. This can be caused by a load imbalance, insufficient concurrency for all working threads, or busy waits of worker threads while serial code is executed. If there is an imbalance, apply dynamic work scheduling or reduce the size of work chunks or tasks. If there is insufficient concurrency, consider collapsing the outer and inner loops. If there is a wait for completion of serial code, explore options for parallelization with Intel Advisor, algorithm, or microarchitecture tuning of the application's serial code

with VTune Profiler Hotspots or Microarchitecture Exploration analysis respectively. For OpenMP* applications, use the Per-Barrier OpenMP Potential Gain metric set in the HPC Performance Characterization analysis to discover the reason for high imbalance or serial spin time.

Lock Contention

Metric Description

Lock Contention time is CPU time when working threads are spinning on a lock consuming CPU resources. High metric value may signal inefficient parallelization with highly contended synchronization objects. To avoid intensive synchronization, consider using reduction, atomic operations or thread local variables where possible.

Possible Issues

When synchronization objects are used inside a parallel region, threads can spend CPU time waiting on a lock release, contending with other threads for a shared resource. Where possible, reduce synchronization by using reduction or atomic operations, or minimize the amount of code executed inside the critical section.

Other (Spin)

Metric Description

This metric shows unclassified Spin time spent in a threading runtime library.

Spin and Overhead Time

Overhead Time

Overhead time is the time the system takes to deliver a shared resource from a releasing owner to an acquiring owner. Ideally, the Overhead time should be close to zero because it means the resource is not being wasted through idleness. However, not all CPU time in a parallel application may be spent on doing real payload work. In cases when a parallel runtime (for example, Intel® Threading Building Blocks, Intel® Cilk™, OpenMP*) is used inefficiently, a significant portion of time may be spent inside the parallel runtime wasting CPU time at high concurrency levels. For example, if you increase the number of threads performing some fixed load of work in parallel, each thread gets less work and the overhead, as a relative measure, will get larger. It is a basic application of Amdahl's Law.

To detect this wasted CPU time, Intel® VTune™ Profiler analyzes the call stack at the point of interest and computes the Overhead time performance metric. VTune Profiler classifies the stack layers into user, system, and overhead layers and attributes the CPU time spent in system functions called by overhead functions to the overhead functions.

Spin Time

Spin time is the Wait time during which the CPU is busy. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting. Some Spin time may be preferable to the alternative of increased thread context switches. Too much Spin time, however, can reflect lost opportunity for productive work.

Overhead and Spin Time

VTune Profiler provides the combined **Overhead and Spin Time** metric in the grid and Timeline view of the Hotspots by CPU Utilization, Hotspots by Thread Concurrency, and Hotspots [viewpoints](#). This metric represents the sum of the Overhead and Spin time values calculated as **CPU Time where Call Site Type is Overhead + CPU Time where Call Site Type is Synchronization**. To view the Overhead and Spin time values separately, expand the column by clicking the



symbol.

NOTE

VTune Profiler ignores the Overhead and Spin time when calculating the CPU Utilization metric.

Possible Issues

A significant portion of CPU time is spent in synchronization or threading overhead. Consider increasing task granularity or the scope of data synchronization.

Atomics**Metric Description**

Atomics time is CPU time that a runtime library spends on atomic operations.

Possible Issues

CPU time spent on atomic operations is significant. Consider using reduction operations where possible.

Creation**Metric Description**

Creation time is CPU time that a runtime library spends on organizing parallel work.

Possible Issues

CPU time spent on parallel work arrangement can be a result of too fine-grain parallelism. Try parallelizing outer loops, rather than inner loops, to reduce the work arrangement overhead.

Other (Overhead)**Metric Description**

This metric shows unclassified Overhead time spent in a threading runtime library.

Reduction**Metric Description**

Reduction time is CPU time that a runtime library spends on loop or region reduction operations.

Possible Issues

A significant portion of CPU time is spent on doing reduction.

Scheduling**Metric Description**

Scheduling time is CPU time that a runtime library spends on work assignment for threads. If the time is significant, consider using coarse-grain work chunking.

Possible Issues

Dynamic scheduling with small work chunks can cause increased overhead due to threads frequently returning to the scheduler for more work. Try increasing the chunk size to reduce this overhead.

Tasking**Metric Description**

Tasking time is CPU time that a runtime library spends on allocating and completing tasks.

Split Stores

Metric Description

Throughout the memory hierarchy, data moves at cache line granularity - 64 bytes per line. Although this is much larger than many common data types, such as integer, float, or double, unaligned values of these or other types may span two cache lines. Recent Intel architectures have significantly improved the performance of such 'split stores' by introducing split registers to handle these cases. But split stores can still be problematic, especially if they consume split registers which could be servicing other split loads.

Possible Issues

A significant portion of cycles is spent handling split stores.

Tips

Consider aligning your data to the 64-byte cache line granularity.

Note that this metric value may be highlighted due to Port 4 issue.

Store Bound

Metric Description

This metric shows how often CPU was stalled on store operations. Even though memory store accesses do not typically stall out-of-order CPUs there are few cases where stores can lead to actual stalls.

Possible Issues

CPU was stalled on store operations for a significant fraction of cycles.

Tips

Consider False Sharing analysis as your next step.

Store Latency

Metric Description

This metric represents cycles fraction the CPU spent handling long-latency store misses (missing 2nd level cache).

Possible Issues

This metric represents a fraction of cycles the CPU spent handling long-latency store misses (missing the 2nd level cache). Consider avoiding/reducing unnecessary (or easily loadable/computable) memory store. Note that this metric value may be highlighted due to a Lock Latency issue.

Task Time

Metric Description

Total amount of time spent within a task.

Thread Concurrency

Thread Oversubscription

Metric Description

Thread Oversubscription indicates time spent in the code with the number of simultaneously working threads more than the number of available logical cores on the system.

Possible Issues

Significant amount of time application spent in thread oversubscription. This can negatively impact parallel performance because of thread preemption and context switch cost.

Total Iteration Count

Metric Description

Statistical estimation of the total loop iteration count. Values of this metric are not aggregated per call stack filter mode.

[uOps]

Metric Description

uOp, or micro-op, is a low-level hardware operation. The CPU Front-End is responsible for fetching the program code represented in architectural instructions and decoding them into one or more uOps.

VPU Utilization

Metric Description

This metric measures the fraction of micro-ops that performed packed vector operations of any vector length and any mask. VPU utilization metric can be used in conjunction with the compiler's vectorization report to assess VPU utilization and to understand the compiler's judgement about the code. Note that this metric does not account for loads and stores and does not take into consideration vector length as well as masking. Includes integer packed SIMD.

Possible Issues

This metric measures the fraction of micro-ops that performed packed vector operations of any vector length and any mask. VPU utilization metric can be in conjunction with the compiler's vectorization report to assess VPU utilization and to understand the compiler's judgement about the code. Note that this metric does not account for loads and stores and does not take into consideration vector length as well as masking. This metric includes integer packed SIMD.

Wait Count

Metric Description

Wait Count measures the number of times software threads wait due to APIs that block or cause synchronization.

Wait Rate

Metric Description

Average Wait time (in milliseconds) per synchronization context switch. Low metric value may signal an increased contention between threads and inefficient use of system API.

Possible Issues

The average Wait time is too low. This could be caused by small timeouts, high contention between threads, or excessive calls to system synchronization functions. Explore the call stack, the timeline, and the source code to identify what is causing low wait time per synchronization context switch.

Wait Time

Metric Description

Wait Time occurs when software threads are waiting due to APIs that block or cause synchronization. Wait Time is per-thread, therefore the total Wait Time can exceed the application Elapsed Time.

GPU Metrics Reference

NOTE Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see [GPU Architecture Terminology for Intel® Xe Graphics](#).

Intel® VTune™ Profiler collects and analyzes the following groups of GPU metrics for Intel® HD Graphics and Intel® Iris® Graphics:

- **Overview** metrics:

- Memory Read Bandwidth
- Memory Write Bandwidth
- ALU0 Active
- ALU0 Instructions
- ALU1 Active
- ALU1 Instructions
- ALU2 Active
- ALU2 Instructions
- ALU0 and ALU1 Active
- ALU0 and ALU2 Active

The fifth and subsequent generations of the Intel® Core™ processor family code named Broadwell include these metrics:

- L3 Shader Bandwidth
- Shared Local Memory Read Bandwidth
- Shared Local Memory Write Bandwidth

- **Global Memory Accesses** metrics:

- Shared Local Memory Read Bandwidth
- Shared Local Memory Write Bandwidth
- Render/GPGPU Command Streamer Loaded
- GPU EU Array Usage

The fifth and subsequent generations of the Intel® Core™ processor family include these metrics:

- EU Threads Occupancy
- EU Send Pipeline Active
- L3 Shader Bandwidth

The first and subsequent generations of the Intel® Arc™ GPUs (codenamed Alchemist) include these metrics:

- L3 Read Bandwidth
- L3 Write Bandwidth
- Stack-to-stack Incoming Bandwidth
- Stack-to-stack Outgoing Bandwidth
- Host-to-GPU Memory Read Bandwidth
- Host-to-GPU Memory Write Bandwidth
- System Memory Read Bandwidth
- System Memory Write Bandwidth
- The **Full Compute** group of metrics combines metrics from the **Overview** and **Global Memory Accesses** groups. Use this information to explore the reasons why the GPU execution units were waiting using the same data view.
- **Render Basic** (preview) metrics:
 - Samples Killed in PS, pixels
 - Samples Written
 - Samples Blended

- PS EU Active %
- PS EU Stall %
- VS EU Active
- VS EU Stall

All groups also include the following metrics which track EU activity:

- EU Array Active
- EU Array Stalled
- EU Array Idle
- EU Threads Occupancy
- Computing Threads Started
- GPU Core Frequency

NOTE

To analyze Intel® HD Graphics and Intel® Iris® Graphics hardware events, make sure to [set up your system for GPU analysis](#)

See Also

[Running GPU Analysis from Command Line](#)

[GPU Architecture Terminology for Intel® Xe Graphics](#)

ALU0 Active**Metric Description**

The normalized sum of all cycles on all cores when the XVE ALU0 pipeline was actively processing .

See Also

[Reference for Performance Metrics](#)

ALU0 Instructions**Metric Description**

The number of floating point instructions executed in the XVE ALU0 pipeline.

See Also

[Reference for Performance Metrics](#)

ALU1 Active**Metric Description**

The normalized sum of all cycles on all cores when the XVE ALU1 pipeline was actively processing.

See Also

[Reference for Performance Metrics](#)

ALU1 Instructions

Metric Description

The number of floating point instructions executed in the XVE ALU1 pipeline.

See Also

[Reference for Performance Metrics](#)

ALU2 Active

Metric Description

The normalized sum of all cycles on all cores when the XVE ALU2 pipeline was active.

See Also

[Reference for Performance Metrics](#)

ALU2 Instructions

Metric Description

The number of instructions executed in the XVE ALU2 pipeline.

See Also

[Reference for Performance Metrics](#)

ALU0 and ALU1 Active

Metric Description

The percentage of GPU time when the ALU0 pipeline (which performs floating point instructions) and the ALU1 pipeline (which performs control, send, math, and integer instructions) were both utilized.

See Also

[Reference for Performance Metrics](#)

ALU0 and ALU2 Active

Metric Description

The percentage of GPU time when both ALU0 and ALU2 pipelines were utilized.

See Also

[Reference for Performance Metrics](#)

Average Time

Metric Description

Average amount of time spent in the task.

See Also

[Reference for Performance Metrics](#)

Computing Threads Started

Metric Description

Number of threads started across all EUs for compute work.

Possible Issues

High thread issue rate lowers GPU usage efficiency due to thread creation overhead even for lightweight GPU threads. To improve performance, change the kernel code to increase the load in a working item, adjust global working size, and so decrease the number of GPU threads.

See Also

[Reference for Performance Metrics](#)

Computing Threads Started, Threads/sec

Metric Description

Number of threads started across all EUs for compute work per second.

See Also

[Reference for Performance Metrics](#)

CPU Time

Metric Description

CPU Time is time during which the CPU is actively executing your application.

See Also

[Reference for Performance Metrics](#)

EU 2 FPU Pipelines Active

Metric Description

The normalized sum of all cycles on all cores when both EU FPU pipelines were actively processing

See Also

[Reference for Performance Metrics](#)

EU Array Active

Metric Description

The normalized sum of all cycles on all cores spent actively executing instructions.

See Also

[Reference for Performance Metrics](#)

EU Array Idle

Metric Description

The normalized sum of all cycles on all cores when no threads were scheduled on a core.

Possible Issues

A significant portion of GPU time is spent idle. That is usually caused by imbalance or thread scheduling problems.

See Also

[Reference for Performance Metrics](#)

EU Array Stalled/Idle

Metric Description

The average time the EUs were stalled or idle.

Possible Issues

The time when the EUs were stalled or idle is high, which has a negative impact on compute-bound applications.

See Also

[Reference for Performance Metrics](#)

EU Array Stalled

Metric Description

The normalized sum of all cycles on all cores spent stalled. At least one thread is loaded, but the core is stalled for some reason.

Possible Issues

A significant portion of GPU time is spent in stalls. For compute bound code it indicates that the performance might be limited by memory or sampler accesses.

See Also

[Reference for Performance Metrics](#)

EU IPC Rate

Metric Description

The average rate of instructions per cycle (IPC) calculated for 2 FPU pipelines

See Also

[Reference for Performance Metrics](#)

EU Send pipeline active

Metric Description

The normalized sum of all cycles on all cores when EU send pipeline was actively processing

See Also

[Reference for Performance Metrics](#)

EU Threads Occupancy

Metric Description

The normalized sum of all cycles on all cores and thread slots when a slot has a thread scheduled.

See Also

[Reference for Performance Metrics](#)

Host to GPU Memory Read Bandwidth

Metric Description

This metric counts the number of host reads to the GPU local (HBM) memory downstream.

See Also

[Reference for Performance Metrics](#)

Host-to-GPU Memory Write Bandwidth

Metric Description

This metric counts the number of host writes to the GPU local (HBM) memory downstream.

See Also

[Reference for Performance Metrics](#)

Global

Metric Description

Total working size of a computing task.

See Also

[Reference for Performance Metrics](#)

GPU EU Array Usage

Metric Description

The normalized sum of all cycles on all cores with at least one thread loaded.

See Also

[Reference for Performance Metrics](#)

GPU L3 Bound

Metric Description

This metric shows how often the GPU was idle or stalled on the L3 cache.

Possible Issues

L3 bandwidth was high when EUs were stalled or idle. Consider improving cache reuse.

See Also

[Reference for Performance Metrics](#)

GPU L3 Miss Ratio

Metric Description

Read and write miss ratio in GPU L3 cache. This doesn't count code lookups.

See Also

[Reference for Performance Metrics](#)

GPU L3 Misses

Metric Description

Read and write misses in GPU L3 cache.

See Also

[Reference for Performance Metrics](#)

GPU L3 Misses, Misses/sec

Metric Description

Read and write misses in GPU L3 cache. This doesn't count code lookups.

See Also

[Reference for Performance Metrics](#)

GPU Memory Read Bandwidth, GB/sec

Metric Description

GPU memory read bandwidth between the GPU, chip uncore (LLC) and main memory. This metric counts all memory accesses that miss the internal GPU L3 cache or bypass it and are serviced either from uncore or main memory.

See Also

[Reference for Performance Metrics](#)

GPU Memory Texture Read Bandwidth, GB/sec

Metric Description

Sampler unit misses in sampler cache.

See Also

[Reference for Performance Metrics](#)

GPU Memory Write Bandwidth, GB/sec

Metric Description

GPU write bandwidth between the GPU, chip uncore (LLC) and main memory. This metric counts all memory accesses that miss the internal GPU L3 cache or bypass it and are serviced either from uncore or main memory.

See Also

[Reference for Performance Metrics](#)

GPU Texel Quads Count, Count/sec

Metric Description

Number of texels returned from the sampler.

See Also

[Reference for Performance Metrics](#)

GPU Utilization

Metric Description

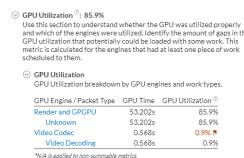
The percentage of time when GPU engine was utilized.

VTune Profiler collects high level information about the **GPU Utilization** metric when you run the [GPU Offload](#) and [GPU Compute/Media Hotspots](#) analyses. This information is available in the [GPU Offload](#) viewpoint. To see more detailed metric information, [rebuild the Linux kernel to enable i915 ftrace events](#).

Use the **Summary**, **Platform**, and **Graphics** window to explore the GPU utilization at the application and computing task level.

GPU Utilization in the Summary Window

If your system satisfies [configuration requirements for GPU analysis](#) (i915 ftrace event collection is supported), VTune Profiler displays detailed **GPU Utilization** analysis data across all engines that had at least one DMA packet executed. By default, the VTune Profiler flags the GPU utilization less than 80% as a performance issue. In the example below, 85.9% of the application elapsed time was utilized by GPU engines.



Depending on the target platform used for GPU analysis, the **GPU Utilization** section in the Summary window shows the time (in seconds) used by GPU engines. Note that GPU engines may work in parallel and the total time taken by GPU engines does not necessarily equal the application Elapsed time.

You may correlate GPU Time data with the Elapsed Time metric. The GPU Time value shows a share of the Elapsed time used by a particular GPU engine. If the GPU Time takes a significant portion of the Elapsed Time, it clearly indicates that the application is GPU-bound.

If your system does not support i915 ftrace event collection, all the GPU Utilization statistics will be calculated based on the hardware events and attributed to the **Render** and **GPGPU** engine.

GPU Utilization in the Platform Window

Explore *overall* GPU utilization per GPU engine at each moment of time. By default, the **Platform** window displays GPU Utilization and software queues per GPU engine. Hover over an object executed on the GPU (in yellow) to view a short summary on GPU utilization, where *GPU Utilization* is the time when a GPU engine was executing a workload. You can explore the top GPU Utilization band in the chart to estimate the percentage of GPU engine utilization (yellow areas vs. white spaces) and options to submit additional work to the hardware.

To view and analyze GPU software queues, select an object (packet) in the queue and the VTune Profiler highlights the corresponding software queue bounds:

Full software queue prevents packet submissions and causes waits on a CPU side in the user-mode driver until there is space in the queue. To check whether such a stall decreases your performance, you may decrease a workload on the hardware and switch to the **Graphics** window to see if there are less waits on the CPU in threads that spawn packets. Another option could be to additionally load the queue by tasks and see whether the queue length increases.

Possible Issues

GPU utilization is low. Consider offloading more work to the GPU to increase overall application performance.

See Also

[GPU Application Analysis on Intel® HD Graphics and Intel® Iris® Graphics](#)

[Reference for Performance Metrics](#)

Instance Count

Metric Description

Total number of times a task is run.

See Also

[Reference for Performance Metrics](#)

L3 Read Bandwidth

Metric Description

Total number of bytes read from L3 cache to XVE array.

See Also

[Reference for Performance Metrics](#)

L3 Write Bandwidth

Metric Description

Total number of bytes written to L3 cache by XVE array.

See Also

[Reference for Performance Metrics](#)

L3 Sampler Bandwidth, GB/sec

Metric Description

Total number of bytes transferred between Samplers and L3 caches.

See Also

[Reference for Performance Metrics](#)

L3 Shader Bandwidth, GB/sec

Metric Description

Total number of bytes transferred directly between EUs and L3 caches.

See Also

[Reference for Performance Metrics](#)

LLC Miss Rate due GPU Lookups

Metric Description

The Last Level Uncore cache (LLC) miss rate across all look-ups done from the GPU.

See Also

[Reference for Performance Metrics](#)

LLC Miss Ratio due GPU Lookups

Metric Description

The Last Level Uncore cache (LLC) miss count across all lookups done from the GPU.

See Also

[Reference for Performance Metrics](#)

Local

Metric Description

Local space size of a computing task. For example, for an OpenCL kernel, it is a working group size.

See Also

[Reference for Performance Metrics](#)

Maximum GPU Utilization

Metric Description

Maximum GPU usage across engines that had at least one packet on them.

See Also

[Reference for Performance Metrics](#)

Occupancy

Metric Description

The normalized sum of all cycles on all core and thread slots when a slot has a thread scheduled.

Possible Issues

Low value of the occupancy metric may be caused by inefficient work scheduling. Make sure work items are neither too small nor too large.

See Also

[Reference for Performance Metrics](#)

PS EU Active %

The metric **PS EU Active %** represents the percentage of overall GPU time that the EUs were actively executing Pixel Shader instructions.

This metric is important if pixel shading seems to be the bottleneck for selected rendering calls.

Possible Issues

- If **PS EU Active %** is 50%, it means that half of the overall GPU time was spent actively executing Pixel Shader instructions.
- If **PS EU Active %** is 0%, it means that no Pixel Shader was associated with the selected draw calls, or that the amount of time actively executing Pixel Shader instructions was negligible.

To improve performance:

- If **PS EU Active %** accounts for most of the EU active time, then to improve performance you may need to simplify the pixel shader.
- If **PS EU Active %** is larger than you would expect and you are encountering slow rendering times, you should examine the pixel shader code for potential reasons why these stalls may be occurring.

See Also

[GPU Rendering Analysis \(Preview\)](#)

PS EU Stall %

Metric Description

The metric **PS EU Stall %** represents the percentage of overall GPU time that the EUs were stalled in Pixel Shader instructions. This metric is important if pixel shading seems to be the bottleneck for selected rendering calls.

NOTE

This metric does not show total amount of stalled time in the pixel shader, but only the fraction of time when pixel shader stalls caused the entire EU to stall. The entire EU stalls when all of its threads are stalled.

Possible Issues

- If **PS EU Stall %** is 50%, it means that half of the overall GPU time was spent stalled on Pixel Shader instructions.
- If **PS EU Stall %** is 0% it means that no Pixel Shader was associated with selected rendering calls or Pixel Shader threads were not causing EUs stalls.

To improve performance:

- If **PS EU Stall %** accounts for most the EU active time, then to improve performance you may need to simplify the pixel shader.
- If **PS EU Stall %** is larger than you expect and you are encountering slow rendering times, you need to concentrate on pixel shader code to find reasons for these stalls.

See Also

[GPU Rendering Analysis \(Preview\)](#)

Ratio to Max Bandwidth, %

Metric Description

Ratio of the bandwidth on this link to its theoretical peak.

See Also

[Reference for Performance Metrics](#)

Ratio to Max Bandwidth, %

Metric Description

Ratio of the write bandwidth on this link to its write theoretical peak.

See Also

[Reference for Performance Metrics](#)

Ratio to Max Bandwidth, %

Metric Description

Ratio of the read bandwidth on this link to its read theoretical peak.

See Also

[Reference for Performance Metrics](#)

Render/GPGPU Command Streamer Loaded

Metric Description

The normalized sum of all cycles where commands exist on the GPU Render/GPGPU ring.

See Also

[Reference for Performance Metrics](#)

Samples Blended

Metric Description

The **Samples Blended** metric represents the total number of blended samples or pixels written to all render targets.

See Also

[GPU Rendering Analysis \(Preview\)](#)

Samples Killed in PS, pixels

Metric Description

The **Samples Killed in PS, pixels** metric represents the total number of samples or pixels dropped in pixel shaders.

See Also

[GPU Rendering Analysis \(Preview\)](#)

Samples Written

Metric Description

The **Samples Written** metric represents the number of pixels/samples written to render targets.

The graphics driver 9.17.10 introduces a new notion of deferred clears. For the sake of optimization, the driver decides whether to defer the actual rendering of clear calls in case subsequent clear and draw calls make it unnecessary. As a result, when clear calls are deferred, the Intel® VTune™ Profiler shows their GPU Duration and Samples Written as zero. If later it turns out that a clear call needs to be drawn, the work associated with that clear call gets included in the duration of the erg that was being drawn when this clear call was deferred, not necessarily a clear call. This means that in the VTune Profiler metrics associated with a clear call accurately reflect the real work associated with that erg.

See Also

[GPU Rendering Analysis \(Preview\)](#)

Sampler Busy

Metric Description

The normalized sum of all cycles on all cores when the Sampler was busy while EUs were stalled or idle.

Possible Issues

Sampler was overutilized when EUs were stalled or idle. Consider reducing the image-related operations.

See Also

[Reference for Performance Metrics](#)

Sampler Is Bottleneck

Metric Description

Sampler stalls EUs due to the full input fifo queue, and starves the output fifo, so EUs need to wait to submit requests to sampler.

Possible Issues

Significant amount of sampler accesses might cause stalls. Consider decreasing the use of the sampler or access it with a better locality.

See Also

[Reference for Performance Metrics](#)

Shared Local Memory Read Bandwidth, GB/sec

Metric Description

Untyped memory reads from Shared Local Memory.

See Also

[Reference for Performance Metrics](#)

Shared Local Memory Write Bandwidth, GB/sec

Metric Description

Untyped memory writes to Shared Local Memory.

See Also

[Reference for Performance Metrics](#)

SIMD Width

Metric Description

The number of working items processed by a GPU thread.

See Also

[Reference for Performance Metrics](#)

Stack-to-stack Incoming Bandwidth

Metric Description

Incoming bandwidth for the stack-to-stack link. This metric counts all incoming writes and outgoing reads return.

See Also

[Reference for Performance Metrics](#)

Stack-to-stack Outgoing Bandwidth

Metric Description

Outgoing bandwidth for the stack-to-stack link. This metric counts all outgoing writes and incoming reads return.

See Also

[Reference for Performance Metrics](#)

System Memory Read Bandwidth

Metric Description

System memory read bandwidth originated from GPU. This metric counts the number of system memory reads (upstream).

See Also

[Reference for Performance Metrics](#)

System Memory Write Bandwidth

Metric Description

System memory write bandwidth originated from the GPU. This metric counts the number of system memory writes (upstream).

See Also

[Reference for Performance Metrics](#)

Size

Metric Description

Amount of memory processed on a GPU.

See Also

[Reference for Performance Metrics](#)

Total, GB/sec

Metric Description

Average bandwidth of data transfer between a CPU and a GPU. In some cases (for example, `clEnqueueMapBuffer`), there may be transfers generating high bandwidth values because memory is not copied but shared via L3 cache.

See Also

[Reference for Performance Metrics](#)

Total Time

Metric Description

Total amount of time spent within a task.

See Also

[Reference for Performance Metrics](#)

Typed Memory Read Bandwidth, GB/sec

Metric Description

Bandwidth of memory read from typed buffers. Note that reads from images (for example created with `clCreateImage`) are counted by sampler accesses and Texture Read metrics.

See Also

[Reference for Performance Metrics](#)

Typed Memory Write Bandwidth, GB/sec

Metric Description

Bandwidth of memory written to typed buffers (for example created with `clCreateImage`).

See Also

[Reference for Performance Metrics](#)

Typed Reads Coalescence

Metric Description

Transaction Coalescence is a ratio of the used bytes to all bytes requested by the transaction. The lower the coalescence, the bigger part of the bandwidth is wasted. It originates from the GPU Data Port function that dynamically merges scattered memory operations into fewer operations over non-duplicated 64-byte cacheline requests. For example, if a 16-wide SIMD operation consecutively reads integer array elements with a stride of 2, the coalescence of such a transaction is 50%, because half of the bytes in the requested cacheline is not used.

See Also

[Reference for Performance Metrics](#)

Typed Writes Coalescence

Metric Description

Transaction Coalescence is a ratio of the used bytes to all bytes requested by the transaction. The lower the coalescence, the bigger part of the bandwidth is wasted. It originates from the GPU Data Port function that dynamically merges scattered memory operations into fewer operations over non-duplicated 64-byte cacheline requests. For example, if a 16-wide SIMD operation consecutively reads integer array elements with a stride of 2, the coalescence of such a transaction is 50%, because half of the bytes in the requested cacheline is not used.

See Also

[Reference for Performance Metrics](#)

Untyped Memory Read Bandwidth, GB/sec

Metric Description

Bandwidth of memory read from untyped buffers (for example created with `clCreateBuffer`).

See Also

[Reference for Performance Metrics](#)

Untyped Memory Write Bandwidth, GB/sec

Metric Description

Bandwidth of memory written to untyped buffers (for example created with `clCreateBuffer`).

See Also

[Reference for Performance Metrics](#)

Untyped Reads Coalescence

Metric Description

Transaction Coalescence is a ratio of the used bytes to all bytes requested by the transaction. The lower the coalescence, the bigger part of the bandwidth is wasted. It originates from the GPU Data Port function that dynamically merges scattered memory operations into fewer operations over non-duplicated 64-byte

cacheline requests. For example, if a 16-wide SIMD operation consecutively reads integer array elements with a stride of 2, the coalescence of such a transaction is 50%, because half of the bytes in the requested cacheline is not used.

See Also

[Reference for Performance Metrics](#)

Untyped Writes Coalescence

Metric Description

Transaction Coalescence is a ratio of the used bytes to all bytes requested by the transaction. The lower the coalescence, the bigger part of the bandwidth is wasted. It originates from the GPU Data Port function that dynamically merges scattered memory operations into fewer operations over non-duplicated 64-byte cacheline requests. For example, if a 16-wide SIMD operation consecutively reads integer array elements with a stride of 2, the coalescence of such a transaction is 50%, because half of the bytes in the requested cacheline is not used.

See Also

[Reference for Performance Metrics](#)

VS EU Active

Metric Description

The **VS EU Active** metric represents the percentage of overall GPU time that the execution units (EUs) were actively executing Vertex Shader instructions. This metric is important if vertex processing seems to be a bottleneck for selected rendering calls.

Possible Issues

- If **VS EU Active** is 50%, half of the overall GPU time was spent actively executing Vertex Shader instructions.
- If **VS EU Active** is 0%, no Vertex Shader was associated with the selected draw calls, or the amount of time actively executing Vertex Shader instructions was negligible.

To improve performance:

- If **VS EU Active** accounts for most of the EU active time, then to improve performance you should simplify the vertex shader or simplify and optimize the geometry of your primitives.
- If **VS EU Active** is significant, you should examine your vertex shader code to find the reasons that might be causing stalls.

See Also

[GPU Rendering Analysis \(Preview\)](#)

VS EU Stall

Metric Description

The **VS EU Stall** metric represents the percentage of overall GPU time that the execution units (EUs) were stalled in Vertex Shader instructions. This metric is important if vertex processing seems to be the bottleneck for selected rendering calls.

NOTE

This metric does not include the total amount of time stalled in the vertex shader, but only the fraction of the time when vertex shader stalls were causing the entire EU to stall. The entire EU stalls when all of its threads are stalled.

Possible Issues

- If **VS EU Stall** is 50%, it means that half of the overall GPU time was spent stalled on Vertex Shader instructions.
- If **VS EU Stall** is 0%, it means that no Vertex Shader was associated with selected rendering calls or Vertex Shader threads were not causing EUs stalls.

To improve performance:

- If **VS EU Stall** accounts for most of the EU active time, then to improve performance you might need to simplify the vertex shader or simplify and optimize geometry.
- If **VS EU Stall** is significant, you need to concentrate on vertex shader code to find the reasons that are causing stalls.

See Also

[GPU Rendering Analysis \(Preview\)](#)

OpenCL™ Kernel Analysis Metrics Reference**Computing Task Total Time****Metric Description**

Total amount of time spent within a *computing task* (OpenCL™ kernel).

See Also

[Interpreting GPU OpenCL Application Analysis Data](#)

Instance Count**Metric Description**

Total number of times a *computing task* (OpenCL™ kernel) is run.

See Also

[Interpreting GPU OpenCL Application Analysis Data](#)

SIMD Width**Metric Description**

The number of working items processed by a GPU thread.

See Also

[Interpreting GPU OpenCL Application Analysis Data](#)

SIMD Utilization

Metric Description

The ratio of active SIMD lanes to the width of the SIMD instructions.

See Also

[Reference for Performance Metrics](#)

Work Size

Metric Description

Global Work Size is a total workspace size of a *computing task* (OpenCL™ kernel). *Local Work Size* is a local working group size of a computing task.

See Also

[Interpreting GPU OpenCL Application Analysis Data](#)

Energy Analysis Metrics Reference

Available Core Time

Metric Description

Total execution time over all cores.

See Also

[Reference for Performance Metrics](#)

C-State

C-State residencies are collected from hardware and/or the operating system (OS).

For systems that collect OS C-State residencies, CPU C-states are core power states requested by the Operating System Directed Power Management (OSPM) infrastructure that define the degree to which the processor is "idle".

For systems that collect hardware C-State residencies, CPU C-States are obtained by reading the processor's MSRs which count the actual time spent in each C-State.

C-States range from C0 to Cn. C0 indicates an active state. All other C-states (C1-Cn) represent idle sleep states where the processor clock is inactive (cannot execute instructions) and different parts of the processor are powered down. As the C-States get deeper, the exit latency duration becomes longer (the time to transition to C0) and the power savings becomes greater.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Energy Analysis](#)

[Interpreting Energy Analysis Data](#)

D0ix States

D0ix-states represent power states ranging from D0i0 to D0i3, where D0i0 is fully powered on and D0i3 is primarily powered off.

The SoC is organized into a north and south complex where the compute intensive components (for example, video decode, image processing, and others) are located in the north complex. The south complex contains I/O, audio, system management, and other components. SoC components should be in the D0i3 state when not in use.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Interpreting Energy Analysis Data](#)

DRAM Self Refresh

DRAM Self Refresh residency represents the percentage of time the system's DRAM was doing self-refresh during the collection period. The system's DRAM will enter a low power self-refresh mode when it is not being actively utilized.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Energy Analysis with Intel VTune Profiler](#)

[Interpreting Energy Analysis Data](#)

[Window: Bandwidth](#)

Energy Consumed (mJ)

This column shows the energy consumed per component (package, CPU, GPU) during the collection period (in millijoules).

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Energy Analysis](#)

[Interpreting Energy Analysis Data](#)

Idle Wake-ups

Number of times a thread caused the system to wake up from idleness to begin executing the thread.

This metric is available in the Hardware Events [viewpoint](#) if you enabled the **Collect stacks** option during the hardware event-based sampling analysis configuration.

See Also

[Hardware Event-based Sampling Collection with Stacks](#)

P-State

CPU P-states represent voltage-frequency control states defined as performance states in the industry standard Advanced Configuration and Power Interface (ACPI) specification (see <http://www.acpi.info> for more details).

In voltage-frequency control, the voltage and clocks that drive circuits are increased or decreased in response to a workload. The operating system requests specific P-states based on the current workload. The processor may accept or reject the request and set the P-state based on its own state.

P-states columns represent the processor's supported frequencies and the time spent in each frequency during the collection period.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Interpreting Energy Analysis Data](#)

[Energy Analysis Metrics](#)

S0ix States

S0ix-states represent the residency in the Intel® SoC idle standby power states. The S0ix states shut off part of the SoC when they are not in use. The S0ix states are triggered when specific conditions within the SoC have been achieved, for example: certain components are in low power states. The SoC consumes the least amount of power in the deepest (for example, S0i3) state.

On Linux*, Android*, and Chrome* OS, ACPI-SState represent the system's residency in the ACPI Suspend-To-RAM (S3). In the Suspend-To-RAM state, the Linux kernel powers down many of the systems' components while maintaining the system's state in its main memory. The system consumes the least amount of power possible while in the Suspend-To-RAM state. Note that any wakelock will prevent the system from entering the Suspend-To-RAM state.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Energy Analysis](#)

[Interpreting Energy Analysis Data](#)

[Window: Wakelocks](#)

Temperature

Temperature columns show the number of samples collected in each temperature reading (C°), for each device.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Energy Analysis](#) To analyze the power consumption of your Android*, Windows*, or Linux* platform, run the Intel® SoC Watch collector and view the results using Intel VTune Profiler.

Timer Resolution

The default timer resolution on Windows* is 15.6 ms – a timer interrupt 64 times a second. While in connected standby, the resolution will be changed by the operating system to 30 seconds. When programs increase the timer frequency (decrease the timer resolution), they increase power consumption of the platform.

The Timer Resolution shows the time spent in each resolution interval during the collection period.

NOTE

This metric is collected as part of [energy analysis](#). Collecting energy analysis data with Intel® SoC Watch is available for target Android*, Windows*, or Linux* devices. Import and viewing of the Intel SoC Watch results is supported with any version of the VTune Profiler.

See Also

[Energy Analysis](#)

[Interpreting Energy Analysis Data](#)

[Window: Timer Resolution](#)

Total Time in C0 State

Metric Description

Total time spent in the active C0 state over all cores.

See Also

[Reference for Performance Metrics](#)

Total Time in Non-C0 States

Metric Description

Total time in sleep states C1-Cx over all cores.

See Also

[Reference for Performance Metrics](#)

Total Time in S0 State

Metric Description

Total time spent in the active S0i0 state.

See Also

Reference for Performance Metrics

Total Wake-up Count

Total number of CPU wake-ups over all cores.

This metric is available in the Platform Power Analysis [viewpoint](#).

See Also

Interpreting Energy Analysis Data

Wake-ups

Metric Description

Percentage of core wake-ups over all cores.

See Also

Reference for Performance Metrics

Wake-ups/sec per Core

Metric Description

Rate of wake-ups.

See Also

Reference for Performance Metrics

Intel Processor Events Reference

Intel® VTune™ Profiler provides a set of hardware event-based analysis types that help you estimate how effectively your application uses hardware resources. These analysis types monitor hardware events supported by your system's *Performance Monitoring Unit (PMU)*. The PMU is hardware built inside a processor to measure its performance parameters such as instruction cycles, cache hits, cache misses, branch misses and many others.

NOTE

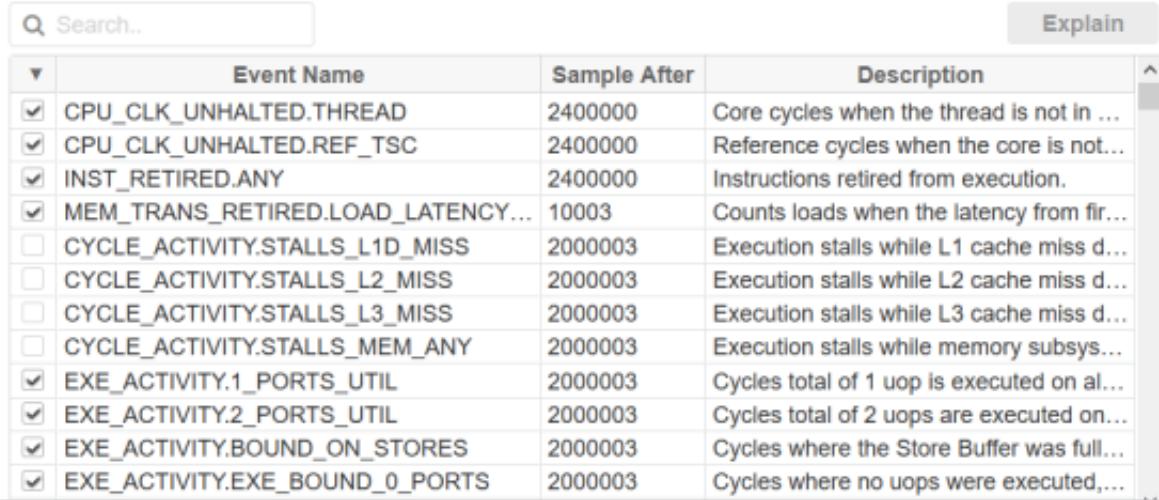
For more information on Intel® 64 and IA-32 architectures, explore *Intel Software Developer Manuals* available at <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.

For details on hardware events supported by your system's PMU, use any of the following options:

- When [adding new events](#) to your custom configuration, select an event in the table and explore its short description, or click the **Explain** button to open the *Intel Processor Events Reference* for more details:

Events configured for CPU: Intel(R) Processor code named Skylake ULT

NOTE: For analysis purposes, Intel VTune Amplifier 2018 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the target configuration window.



The screenshot shows a table titled "Events configured for CPU: Intel(R) Processor code named Skylake ULT". The table has columns for "Event Name", "Sample After", and "Description". There are 15 rows, each with a checkbox in the first column. Most checkboxes are checked, except for the first one which is unchecked. The "Description" column contains brief explanations for each event, such as "Core cycles when the thread is not in ...", "Reference cycles when the core is not...", and "Instructions retired from execution".

	Event Name	Sample After	Description
<input type="checkbox"/>	CPU_CLK_UNHALTED.THREAD	2400000	Core cycles when the thread is not in ...
<input checked="" type="checkbox"/>	CPU_CLK_UNHALTED.REF_TSC	2400000	Reference cycles when the core is not...
<input checked="" type="checkbox"/>	INST_RETIRED.ANY	2400000	Instructions retired from execution.
<input checked="" type="checkbox"/>	MEM_TRANS_RETIRED.LOAD_LATENCY...	10003	Counts loads when the latency from fir...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_L1D_MISS	2000003	Execution stalls while L1 cache miss d...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_L2_MISS	2000003	Execution stalls while L2 cache miss d...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_L3_MISS	2000003	Execution stalls while L3 cache miss d...
<input type="checkbox"/>	CYCLE_ACTIVITY.STALLS_MEM_ANY	2000003	Execution stalls while memory subsys...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.1_PORTS_UTIL	2000003	Cycles total of 1 uop is executed on al...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.2_PORTS_UTIL	2000003	Cycles total of 2 uops are executed on...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.BOUND_ON_STORES	2000003	Cycles where the Store Buffer was full...
<input checked="" type="checkbox"/>	EXE_ACTIVITY.EXE_BOUND_0_PORTS	2000003	Cycles where no uops were executed,...

- For a full list of processor events and descriptions, explore the web-based [Intel Processor Events Reference](#).

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described 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.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.