



Performance Analysis using Intel® VTune™ Amplifier XE



Performance methodology

profiling and tuning

The Goal: minimize the time it takes your program / module / function to execute

- Identify Hotspots and focus on them
- It's just a few functions (20% of code does 80% of job)
- Optimize them (with compiler or hand optimizations)
- Check for hotspots again, and find new ones

How to optimize the Hotspots?

- Maximize CPU utilization and minimize elapsed time
- Ensure CPU is busy all the time
- All Cores busy – parallelism
- Busy with useful tasks
- Optimize tasks execution

The Software Optimization Process

- Back to methodology. Make sure:
 - System level optimization is done
 - Application is multithreaded or using 100% all available CPUs
 - See examples on the next slide
- Now identify top Hotspots and focus on them
- Determine if that 100% CPU usage is indeed effective
- If not, find out microarchitectural reasons of the inefficiency
- Optimize the software by eliminating bottlenecks



Identifying bottlenecks in the target application and eliminating them appropriately is the key to an efficient optimization

Performance profiling

Terminology

Elapsed Time

The total time your target application ran. Wall clock time at end of application
– Wall clock time at start of application

CPU Time

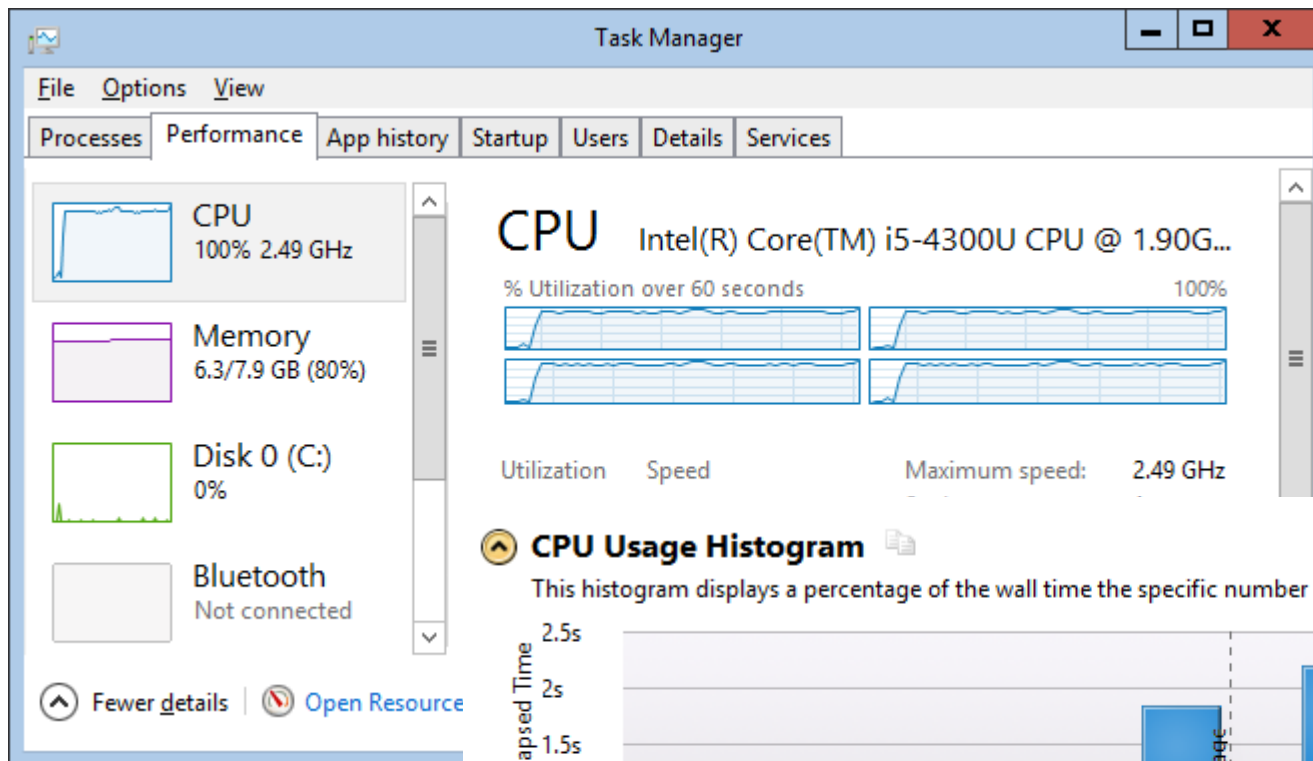
The amount of time a thread spends executing on a logical processor. For multiple threads, the CPU time of the threads is summed.

Wait Time

The amount of time that a given thread waited for some event to occur, such as: synchronization waits and I/O waits

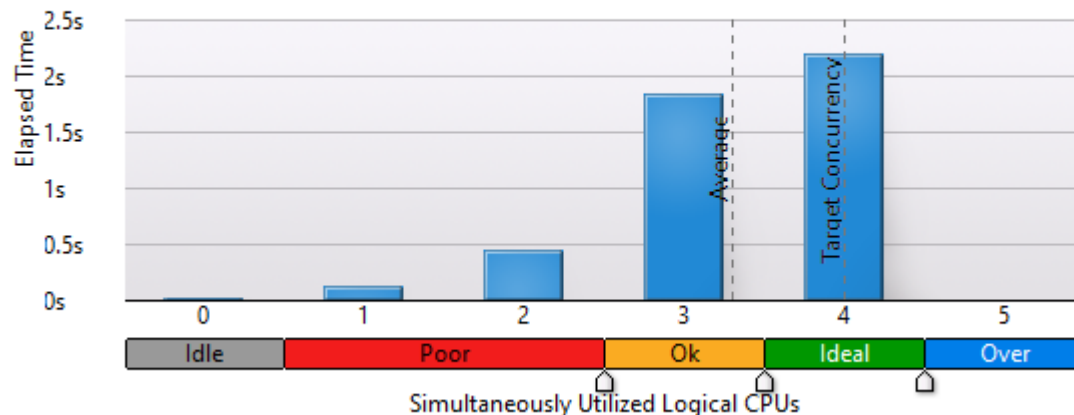
Examples of “100% CPU busy”

1. Multithreaded application



CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously.

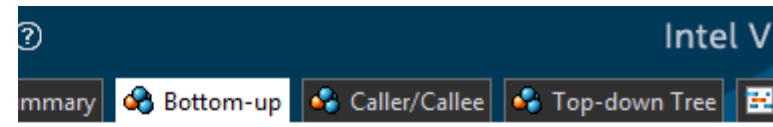
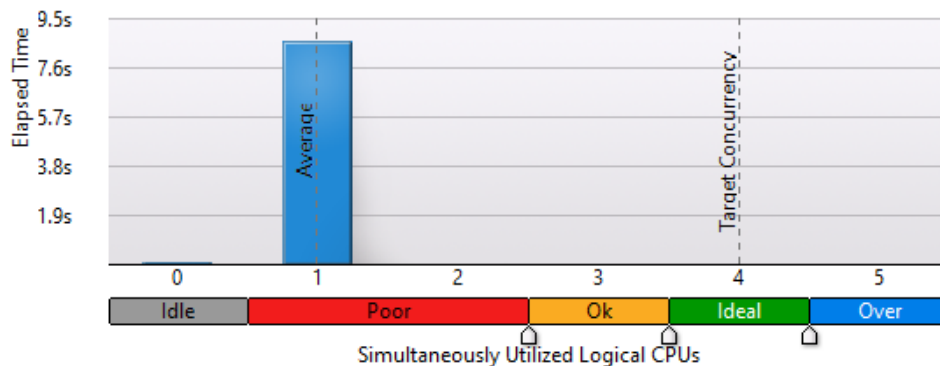


Examples of "100% CPU busy"

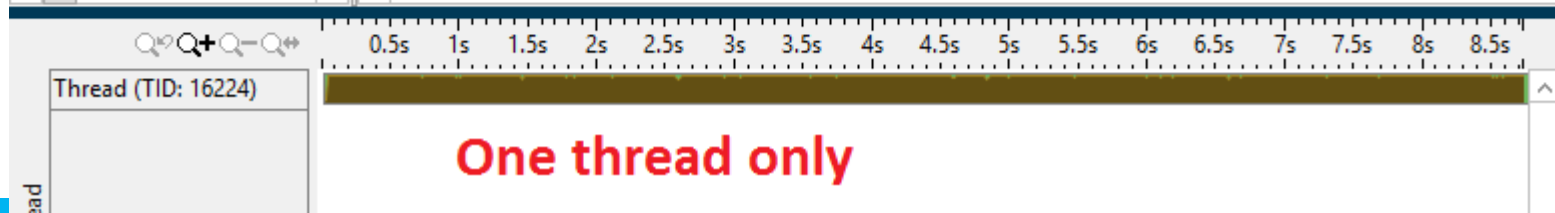
2. Single threaded application

⬆ CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously.



Function / Call Stack	CPU Time	Instructi... Retired	CPI Rate	CPU Frequency Ratio	Module	Function (Full)	Source File	Start Address
+ multiply0	8.452s	5,644,90 ...	4.107	1.100	matrix.exe	multiply0	multiply.c	0x4014d0
+ func@0x140153df0	0.004s	0		0.200	ntoskrnl.exe	func@0x140153df0		0x1401 ...
+ func@0x1400a5de0	0.002s	5,700,000	1.000	1.000	ntoskrnl.exe	func@0x1400a5de0		0x1400 ...
+ func@0x140152040	0.002s	0		1.667	ntoskrnl.exe	func@0x140152040		0x1401 ...
Selected 1 row(s):		8.452s	5,644,90 ...	4.107	1.100			



Agenda

Introduction to Intel® VTune™ Amplifier XE profiler

High-level Features

Types of Analysis

Hotspot analysis

- Basic Hotspots
- Advanced Hotspots
- Lab 1: Find the Performance Hotspot

Concurrency Analysis

- Lab 2: Analyzing Parallelism

Locks and Waits Analysis

- Lab 3: Identifying Parallelism issues

User and Synchronization API, Frame/Task Analysis

- Lab 4: Instrumenting user source code

Command Line Interface, Installation, Remote Collection

Conclusion

Intel® VTune™ Amplifier XE

Performance Profiler

Where is my application...

Spending Time?

Function - Call Stack ▾	CPU Time ▾
algorithm_2	3.560s
do_xform ←	3.560s
algorithm_1	1.412s
BaseThreadInitTh	0.000s

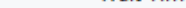
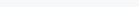
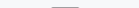
- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?

Line		MEM_LOAD... LLC_MISS
475	float rx, ry, rz =	
476	float param1 = (AP	30,000
477	float param2 = (AP	
478	bool neg = (rz < 0	

- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?

Wait Time▼				Wait Count				
	Idle		Poor		Ok		Ideal	
176.504s								18,277
84.681s								5,499
84.612s								5,489

- See locks by wait time
- Red/Green for CPU utilization during wait

- Windows & Linux
- Low overhead
- No special recompiles

Advanced Profiling For Scalable Multicore Performance

Intel® VTune™ Amplifier XE

Tune Applications for Scalable Multicore Performance

Fast, Accurate Performance Profiles

- Hotspot (Statistical call tree)
- Call counts (Statistical)
- Hardware-Event Sampling

Thread Profiling

- Visualize thread interactions on timeline
- Balance workloads

Easy set-up

- Pre-defined performance profiles
- Use a normal production build

Find Answers Fast

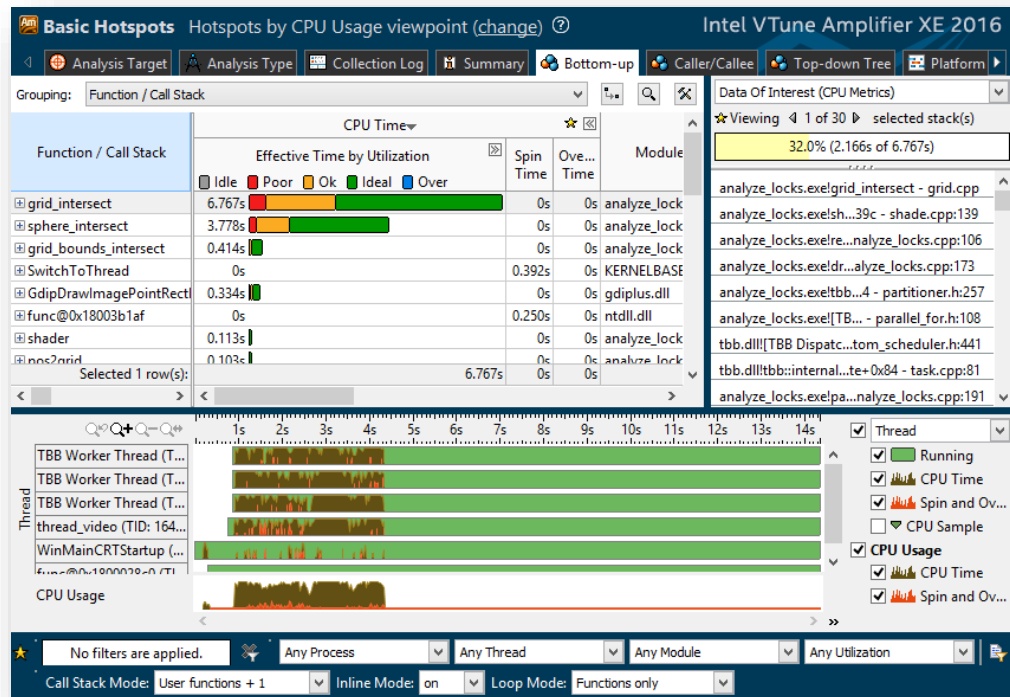
- Filter extraneous data
- View results on the source / assembly

Compatible

- Microsoft, GCC, Intel compilers
- C/C++, Fortran, Assembly, .NET, Java
- Latest Intel® processors
and compatible processors¹

Windows or Linux

- Visual Studio Integration (Windows)
- Standalone user i/f and command line
- 32 and 64-bit



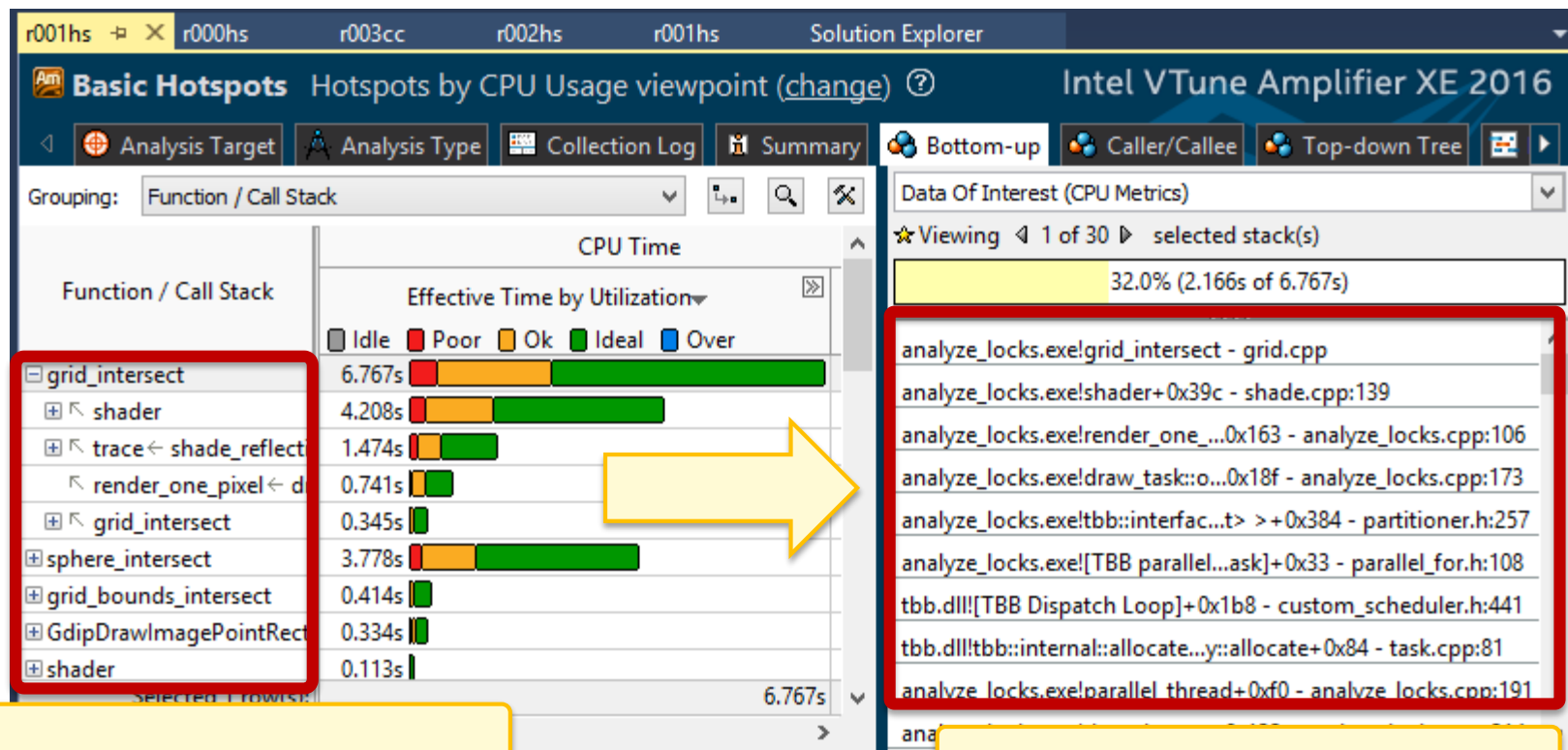
¹ IA32 and Intel® 64 architectures.
Many features work with compatible processors.
Event based sampling requires a genuine Intel® Processor.

A set of instruments to identify performance problems

Quick Overview

Intel® VTune™ Amplifier XE

Identify hotspots



Hottest Functions

Hottest Call Stack

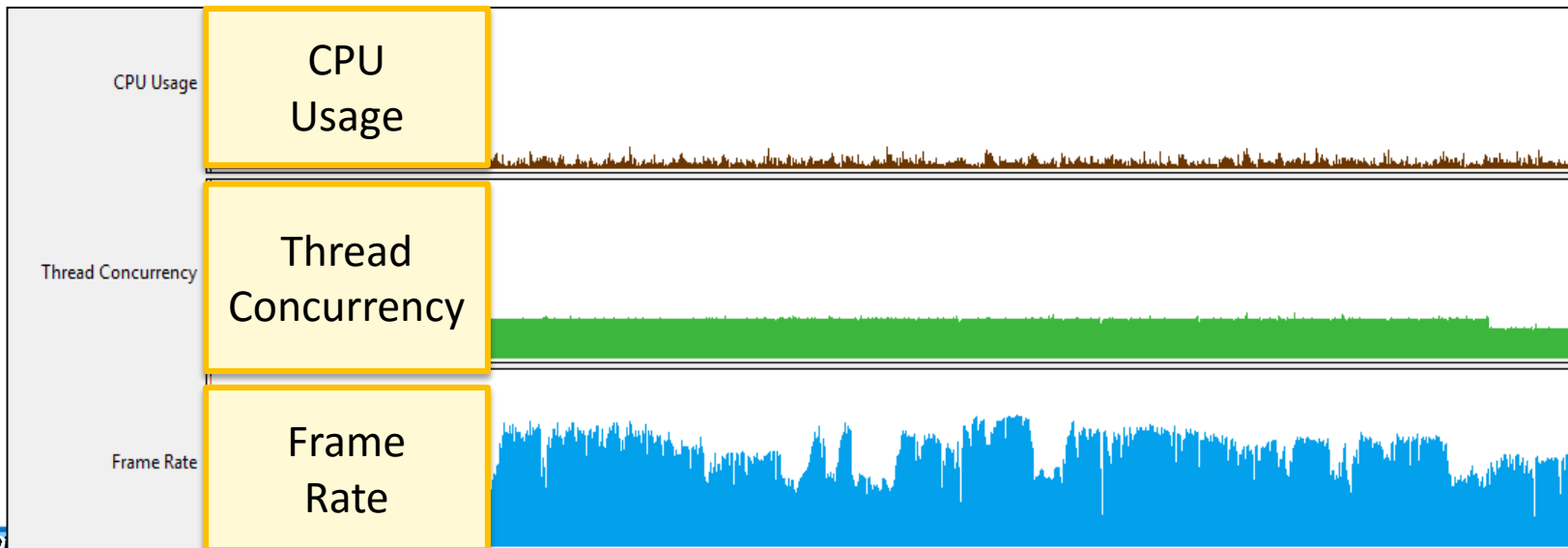
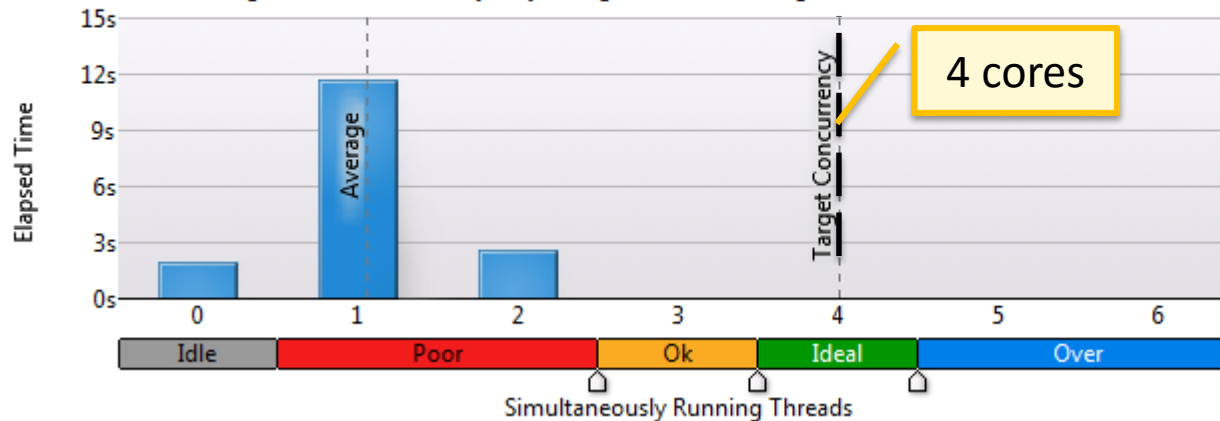
Quickly identify what is important

Intel® VTune™ Amplifier XE

Get a quick snapshot

Thread Concurrency Histogram

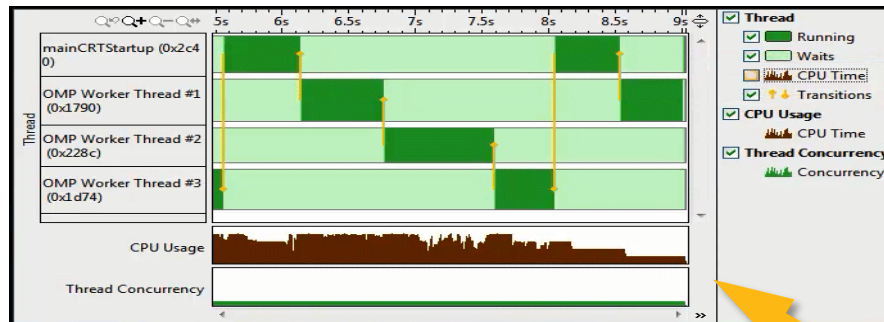
This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.



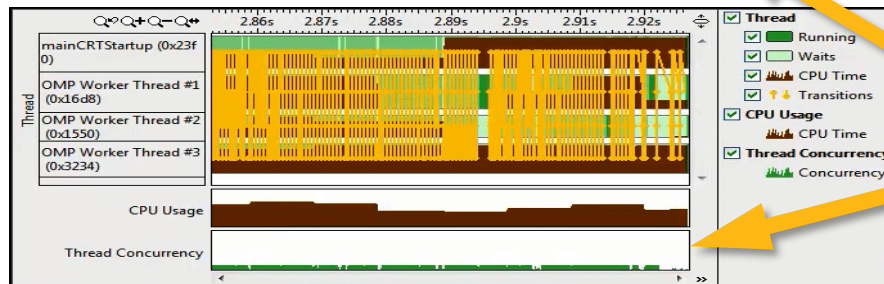
Intel® VTune™ Amplifier XE

Look for Common Patterns

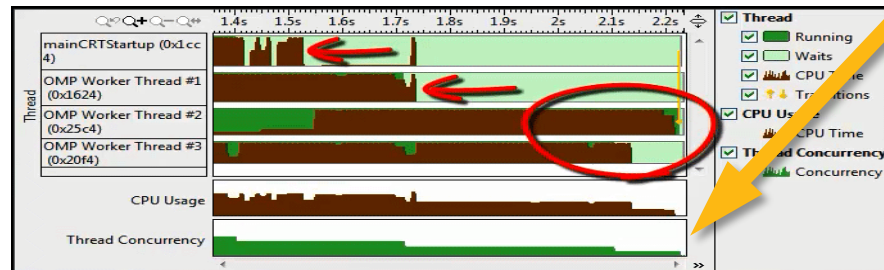
Coarse Grain
Locks



High Lock
Contention



Load
Imbalance



Low
Concurrency

Intel® VTune™ Amplifier XE

Find Answers Fast

Adjust Data Grouping

Function - Call Stack
Module - Function - Call Stack
Source File - Function - Call Stack
Thread - Function - Call Stack
... (Partial list shown)

Double Click Function
to View Source

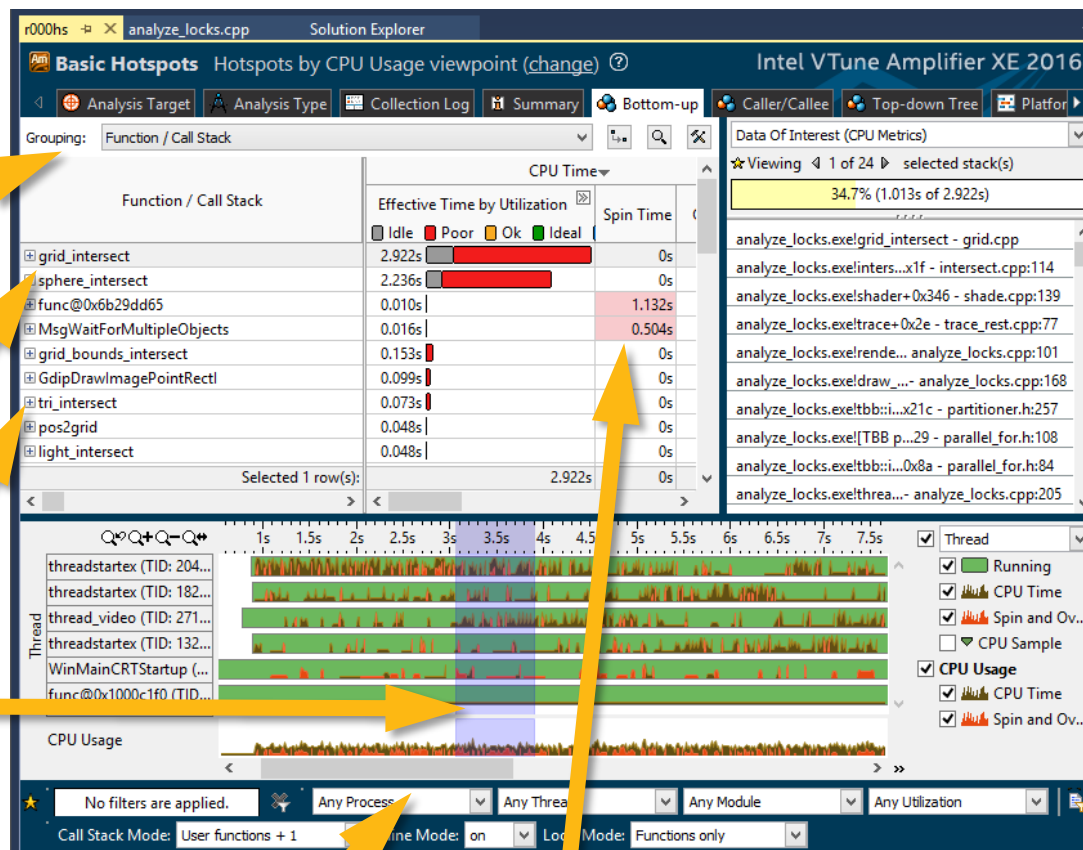
Click [+] for Call Stack

Filter
by Timeline Selection
(or by Grid Selection)

Zoom In And Filter On Selection
Filter In by Selection
Remove All Filters

Filter by Process
& Other Controls

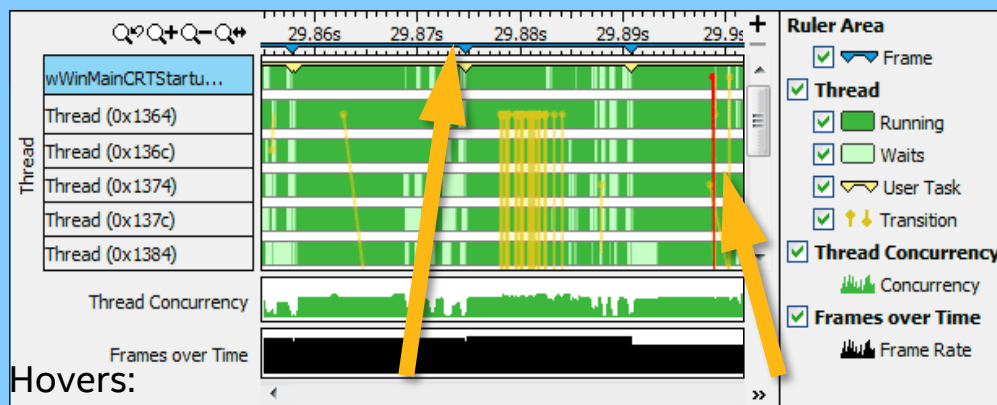
Tuning Opportunities Shown
in Pink. Hover for Tips



Intel® VTune™ Amplifier XE

Timeline Visualizes Thread Behavior

🔑 Transitions Locks & Waits



Frame

Frame
Start: 29.858s Duration: 0.017s
Frame: 72
Frame Domain: Smoke::Framework::execute()
Frame Type: Good
Frame Rate: 59.8242179

🔑 Transition

Transition
wWinMainCRTStartup (0x12d4) to Thread (0x138c) (29.899s to 29.899s)
Sync Object: TBB Scheduler
Object Creation File: taskmanagertbb.cpp
Object Creation Line: 318

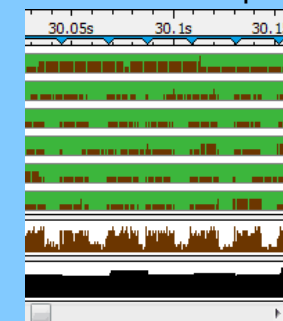
🏠 CPU Time Basic Hotspots



User Task

User Task
Start: 29.958s Duration: 0.018s
Task Type: Smoke::Framework::execute()::Other
Task End Call Stack: Framework::Execute
CPU Time
94.233472%

Advanced Hotspots



Optional: Use API to mark frames and user tasks

Frame User Task

Optional: Add a mark during collection

Mark Timeline

Intel® VTune™ Amplifier XE

See Profile Data On Source / Asm

View Source / Asm or both

CPU Time

Right click for instruction reference manual

Quick Asm navigation:
Select source to highlight Asm

The screenshot displays the Intel VTune Amplifier XE 2016 interface. The main window is titled 'Basic Hotspots' and shows 'Hotspots by CPU Usage viewpoint (change)'. The interface is divided into several panes. On the left, the 'Source' pane shows a list of source code lines with associated CPU time percentages. A blue box with an arrow points to the 'Source' tab, indicating that selecting a source line highlights the corresponding assembly instructions. In the center, the 'Assembly' pane shows the disassembled instructions for the selected source line. A blue box with an arrow points to the 'Assembly' tab, indicating that selecting an assembly instruction highlights the corresponding source code line. On the right, the 'Data Of Interest (CPU Metrics)' pane shows a list of selected stack frames, including 'analyze_locks.exe...ersect - grid.cpp' and 'analyze_locks.exe...shade.cpp:139'. A blue box with an arrow points to the 'Data Of Interest' pane, indicating that right-clicking on an instruction in the assembly pane opens the instruction reference manual. At the bottom, a blue box with an arrow points to the 'Click jump to scroll Asm' button, indicating that clicking this button jumps to the selected assembly instruction and scrolls the assembly pane to it.

So. Li.	Source	CPU Time: Total	Effective Time	Spin Time	Address	Sour... Line	Assembly	CPU Time	Effective Tim...
579	cur = g->cells[voxindex];	1.6%	0.0%	0.0%	0x1400112d0	581	Block 38:		
580	while (cur != NULL) {	0.1%	0.0%	0.0%	0x1400112d4	581	mov rax, qword ptr [rdi+0x8]	2.2%	
581	if (ry->mbox[cur->obj-]	25.1%	0.0%	0.0%	0x1400112d7	581	mov edx, dword ptr [rbx+0x10]	8.1%	
582	ry->mbox[cur->obj->i	8.5%	0.0%	0.0%	0x1400112d9	581	mov ecx, dword ptr [rax]	0.2%	
583	cur->obj->methods->i	6.1%	0.0%	0.0%	0x1400112dd	581	mov rax, qword ptr [rbx+0x18]	14.1%	
584	}				0x1400112de	581	cmp dword ptr [rax+rcx*4], edx	0.6%	
585	cur = cur->next;	5.8%	0.0%	0.0%	0x1400112e0	581	jz 0x1400112f2 <Block 40>		
586	}				0x1400112e2	582	Block 39:		
587	curvox.z += step.z;	0.1%	0.0%	0.0%	0x1400112e2	582	mov dword ptr [rax+rcx*4], edx	8.5%	
588	if (ry->maxdist < tmax.z	0.3%	0.0%	0.0%	0x1400112e5	583	mov rcx, qword ptr [rdi+0x8]	1.6%	
589	break;				0x1400112e9	583	mov rdx, rbx	0.3%	
590	voxindex += step.z*g->xs	0.1%	0.0%	0.0%	0x1400112ec	583	mov rax, qword ptr [rcx+0x10]		
					0x1400112f0	583	call qword ptr [rax]	4.3%	
					0x1400112f2	585	Block 40:		
					0x1400112f2	585	mov rdi, qword ptr [rdi+0x8]		

Quickly scroll to hot spots. Scroll Bar
"Heat Map" is an overview of hot spots

High-level Features

Intel® VTune™ Amplifier XE

Feature Highlights

Basic Hot Spot Analysis (Statistical Call Graph)

- Locates the time consuming regions of your application
- Provides associated call-stacks that let you know how you got to these time consuming regions
- Call-tree built using these call stacks

Advanced Hotspot and architecture analysis

- Based on Hardware Event-based Sampling (EBS)
- Pre-defined tuning experiments

Thread Profiling

- Visualize thread activity and lock transitions in the timeline
- Provides lock profiling capability
- Shows CPU/Core utilization and concurrency information

GPU Compute Performance Analysis

- Collect GPU data for tuning OpenCL applications. Correlate GPU and CPU activities

Intel® VTune™ Amplifier XE

Feature Highlights

Attach to running processes

- Hotspot and Concurrency analysis modes can attach to running processes

System wide data collection

- EBS modes allows system wide data collection and the tool provides the ability to filter this data

GUI

- Standalone GUI available on Windows* and Linux
- Microsoft* Visual Studio integration

Command Line

- Comprehensive support for regression analysis and remote collection

Platform & application support

- Windows* and Linux (Android, Tizen, Yocto – in the ISS)
- Microsoft* .NET/C# applications
- Java* and mixed applications
- Fortran applications

Intel® VTune™ Amplifier XE

Feature Highlights

Event multiplexing

- Gather more information with each profiling run

Timeline correlation of thread and event data

- Populates thread active time with event data collected for that thread
- Ability to filter regions on the timeline

Advanced Source / Assembler View

- See event data graphed on the source / assembler
- View and analyze assembly as basic blocks
- Review the quality of vectorization in the assembly code display of your hot spot

Provides pre-defined tuning experiments

- Predefined profiles for quick analysis configuration
- A user profile can be created on a basis of a predefined profile

User API

- Rich set of user API for collection control, events highlighting, code instrumentation, and visualization enhancing.

Data Collectors and Analysis Types

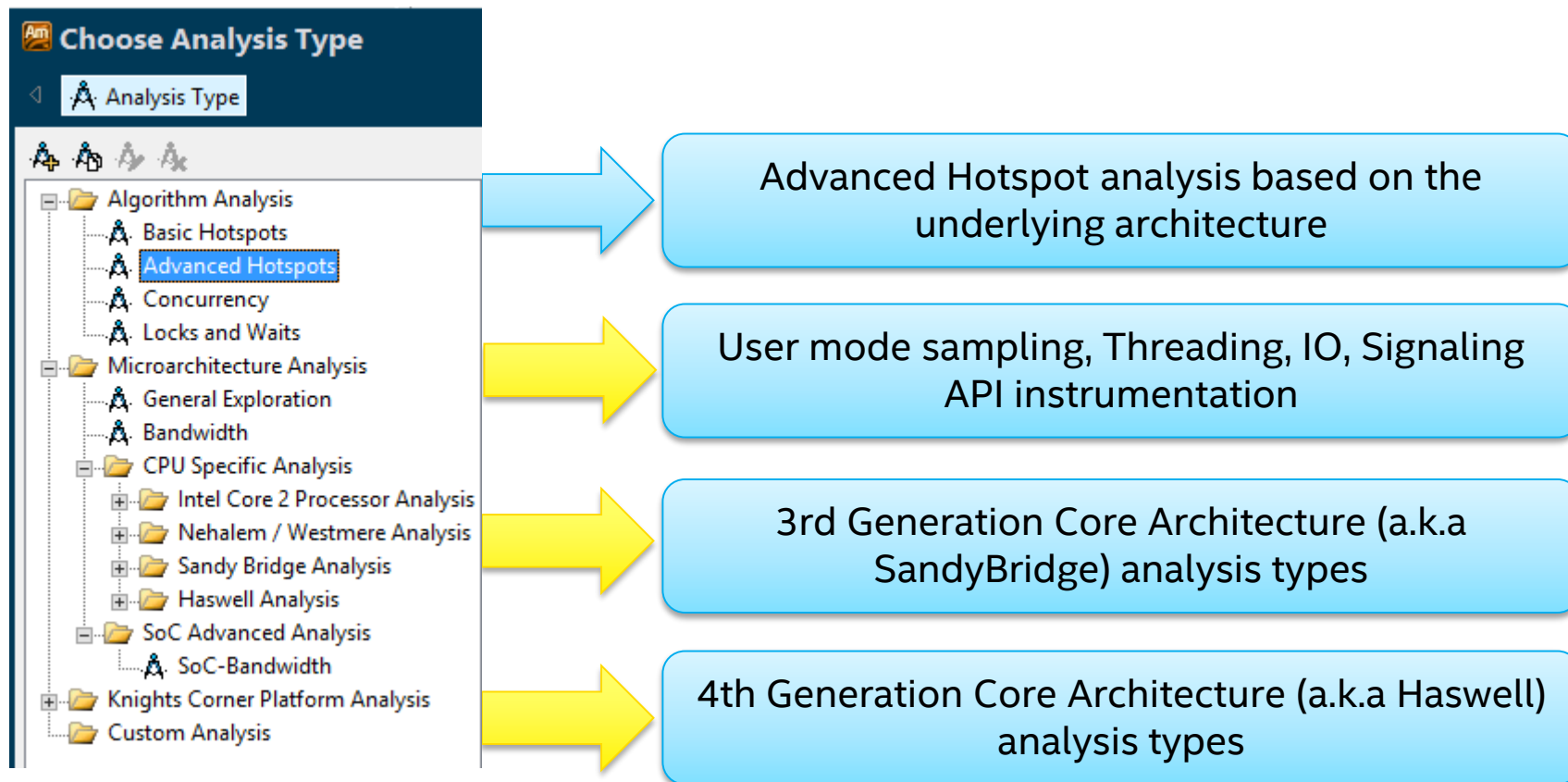
Intel® VTune™ Amplifier XE

Analysis Types (based on technology)

Software Collector Any x86 processor, any virtual, no driver	Hardware Collector Higher res., lower overhead, system wide
Basic Hotspots Which functions use the most time?	Advanced Hotspots Which functions use the most time? Where to inline? – Statistical call counts
Concurrency Tune parallelism. Colors show number of cores used.	General Exploration Where is the biggest opportunity? Cache misses? Branch mispredictions?
Locks and Waits Tune the #1 cause of slow threaded performance – waiting with idle cores.	Advanced Analysis Dig deep to tune bandwidth, cache misses, access contention, etc.

Intel® VTune™ Amplifier XE

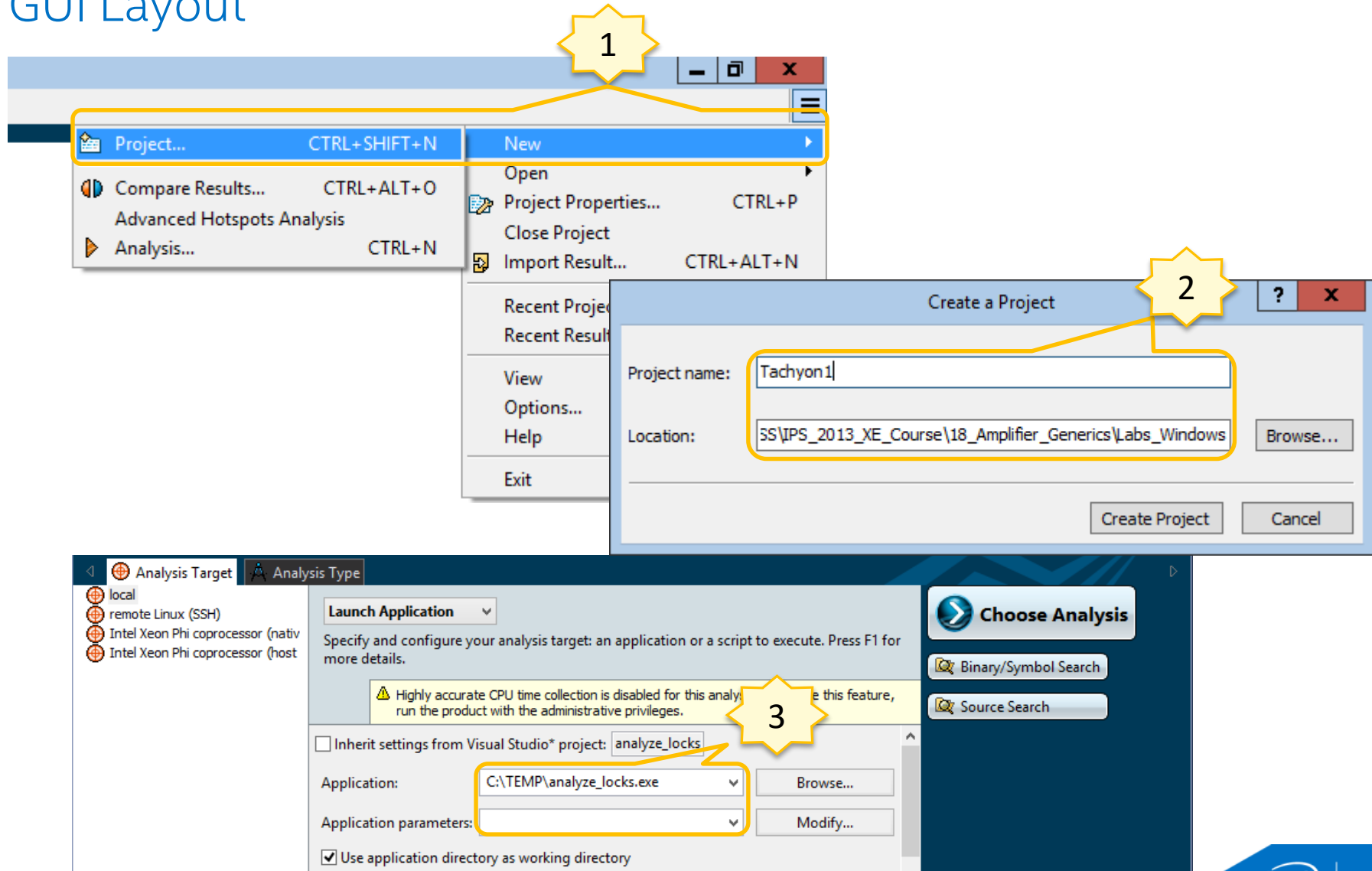
Pre-defined Analysis Types



GUI Layout

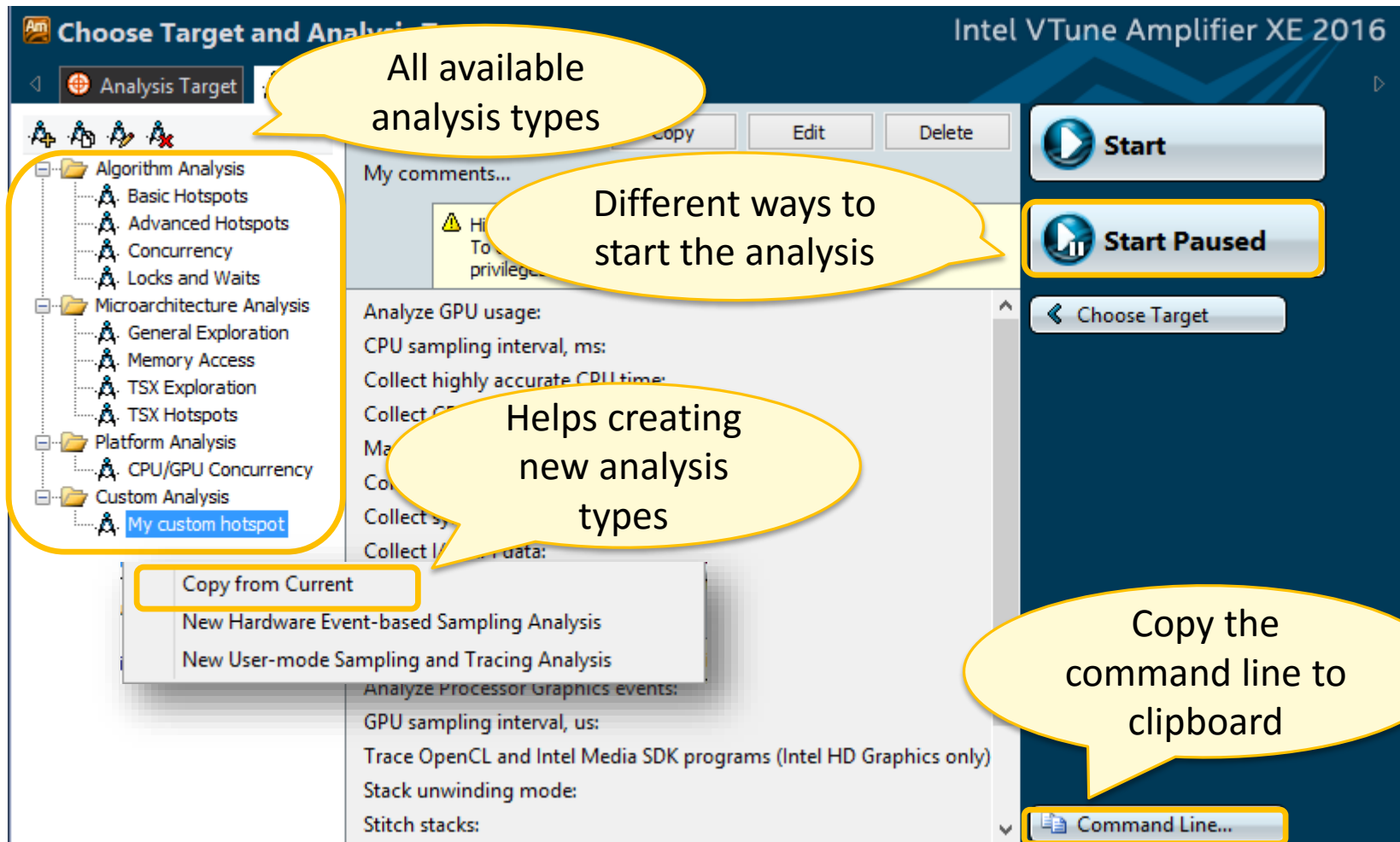
Creating a Project

GUI Layout



Selecting type of data collection

GUI Layout



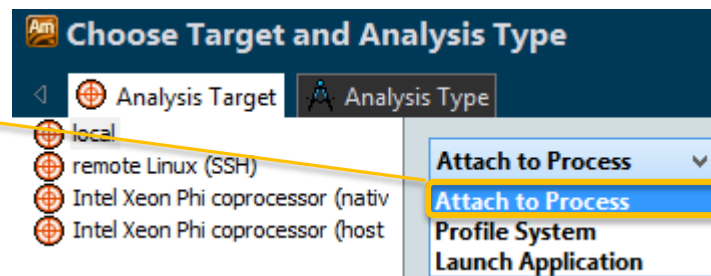
Profile a Running Application

No need to stop and re-launch the app when profiling

Two Techniques:

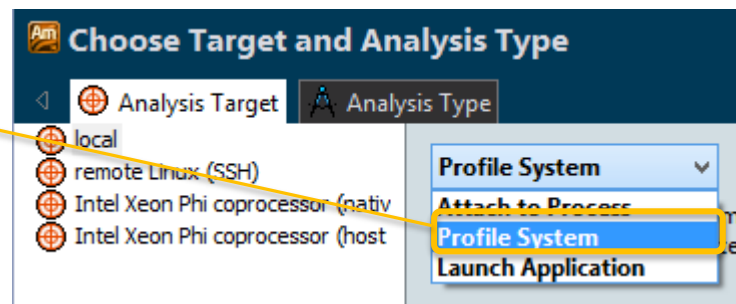
Attach to Process:

- Any type of analysis



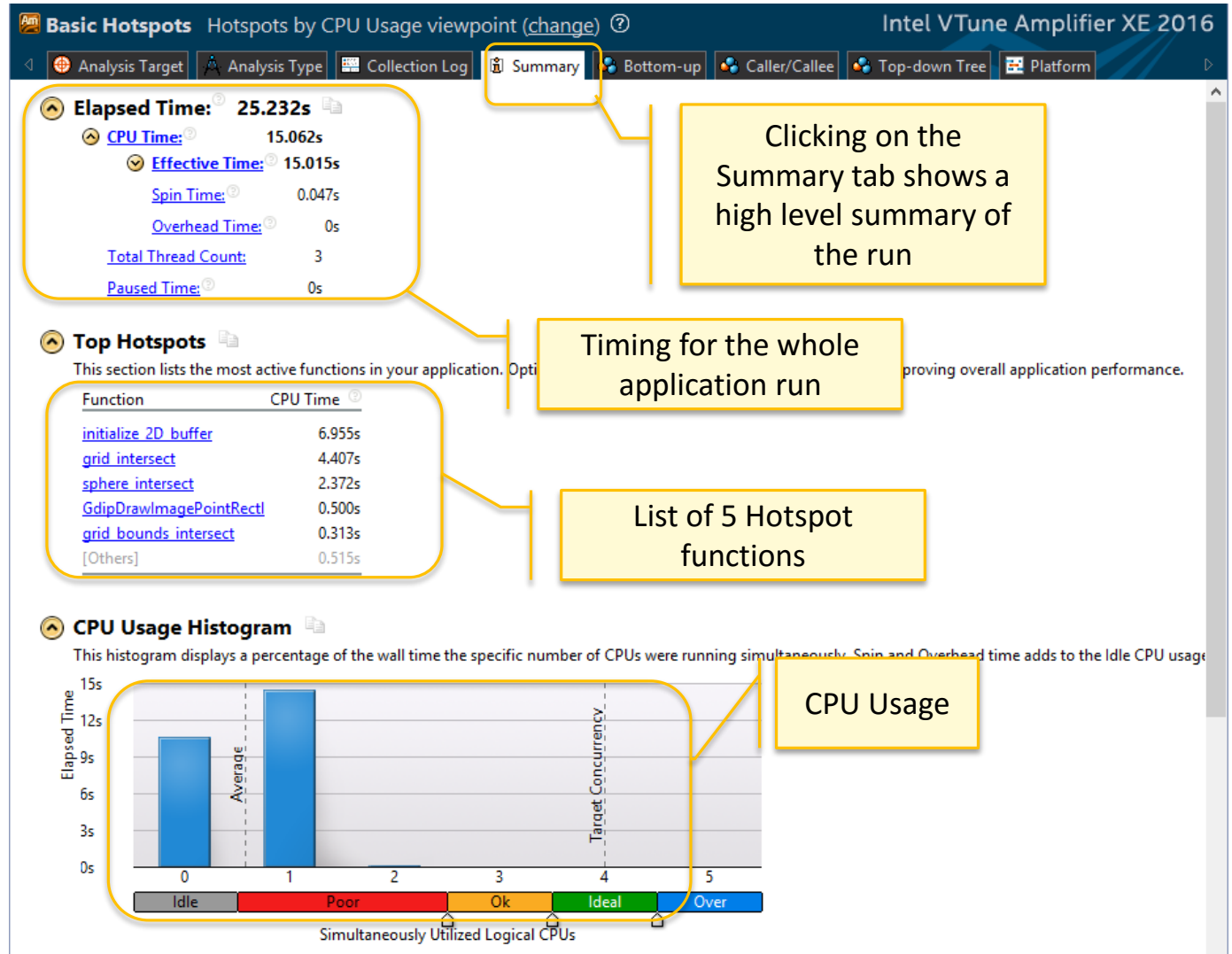
Profile System:

- Advanced Hotspots & Custom EBS
- Optional: Filter by process after collection

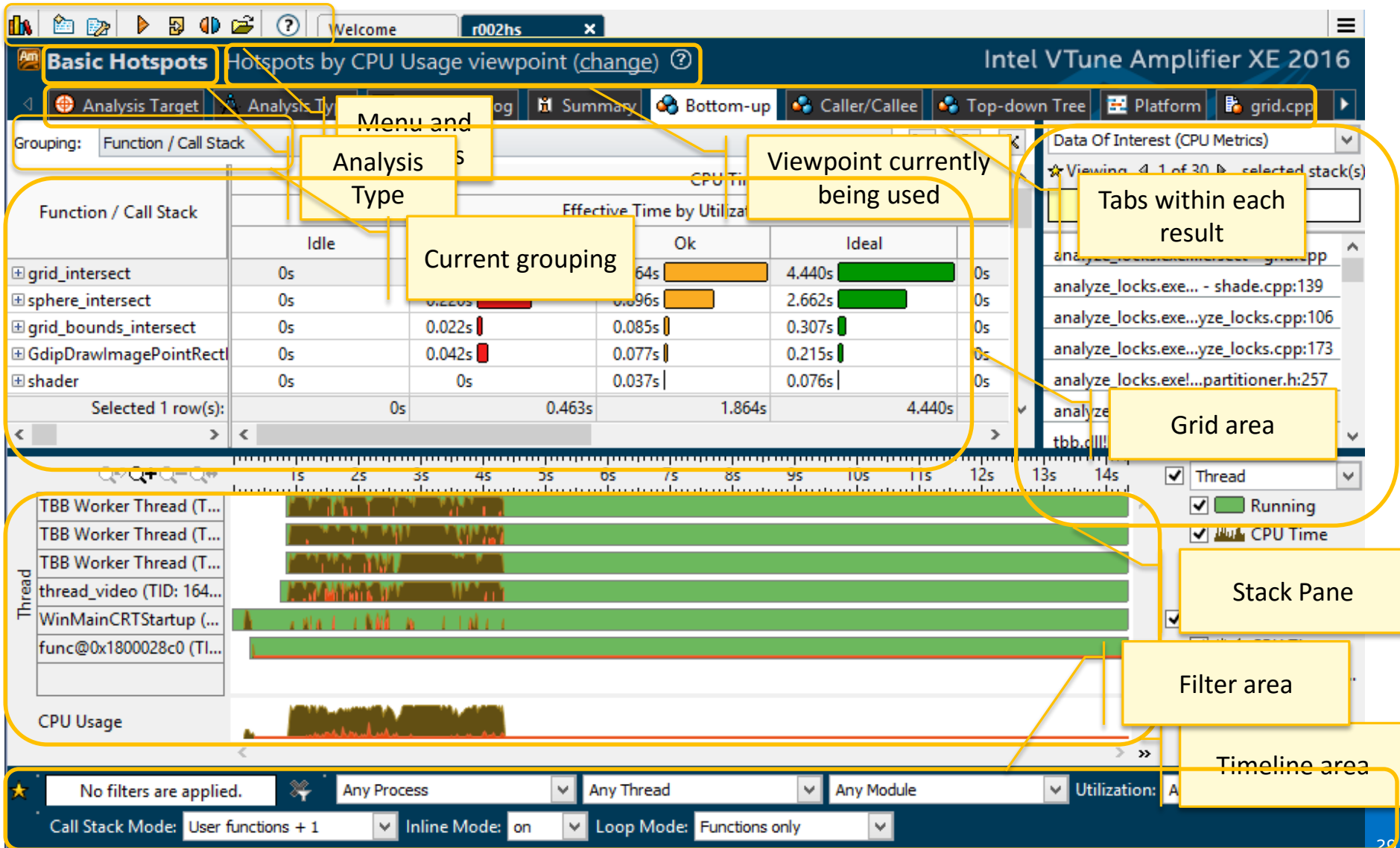


Summary View

GUI Layout



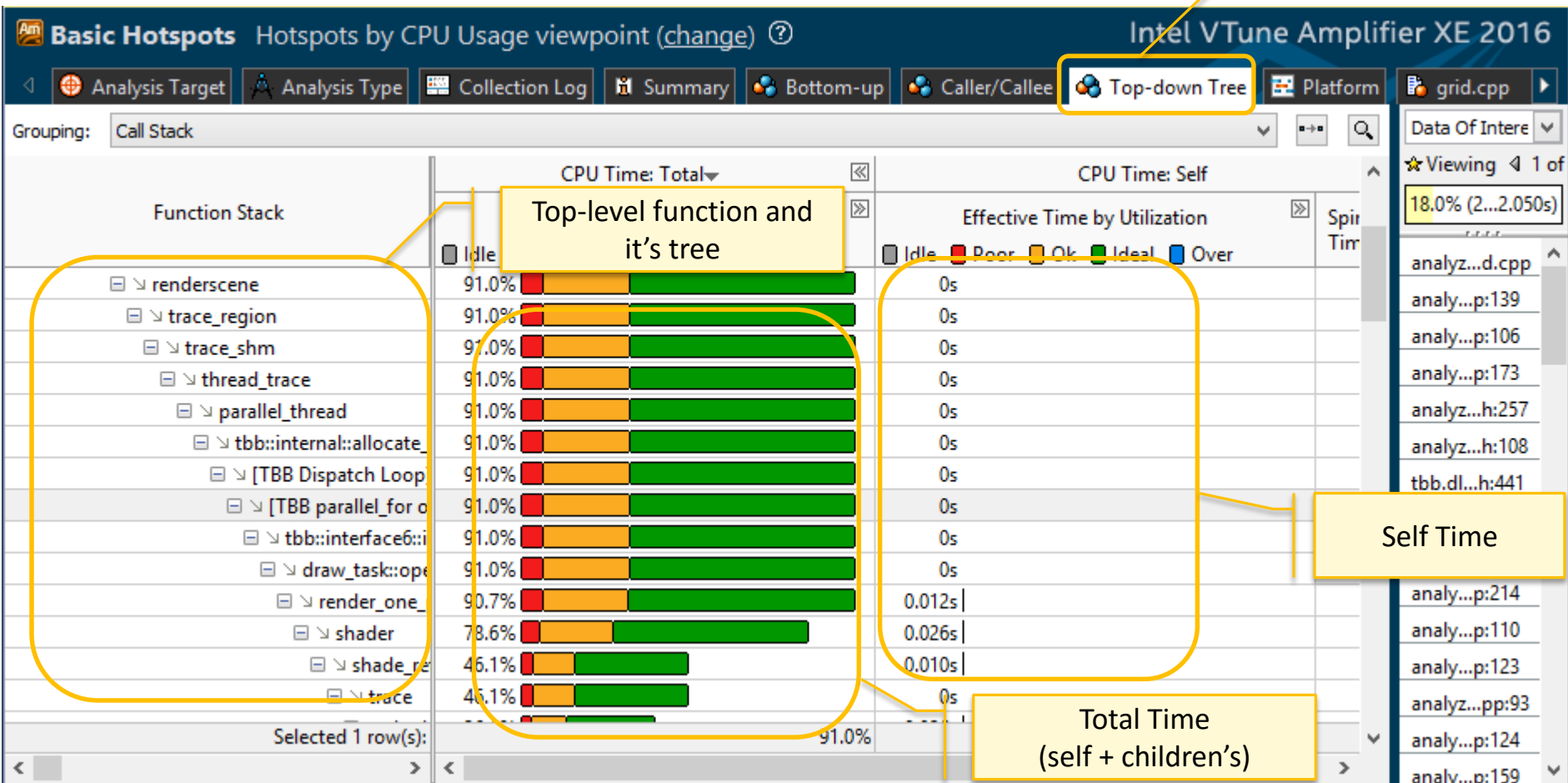
Bottom-Up View GUI Layout



Top-Down View

GUI Layout

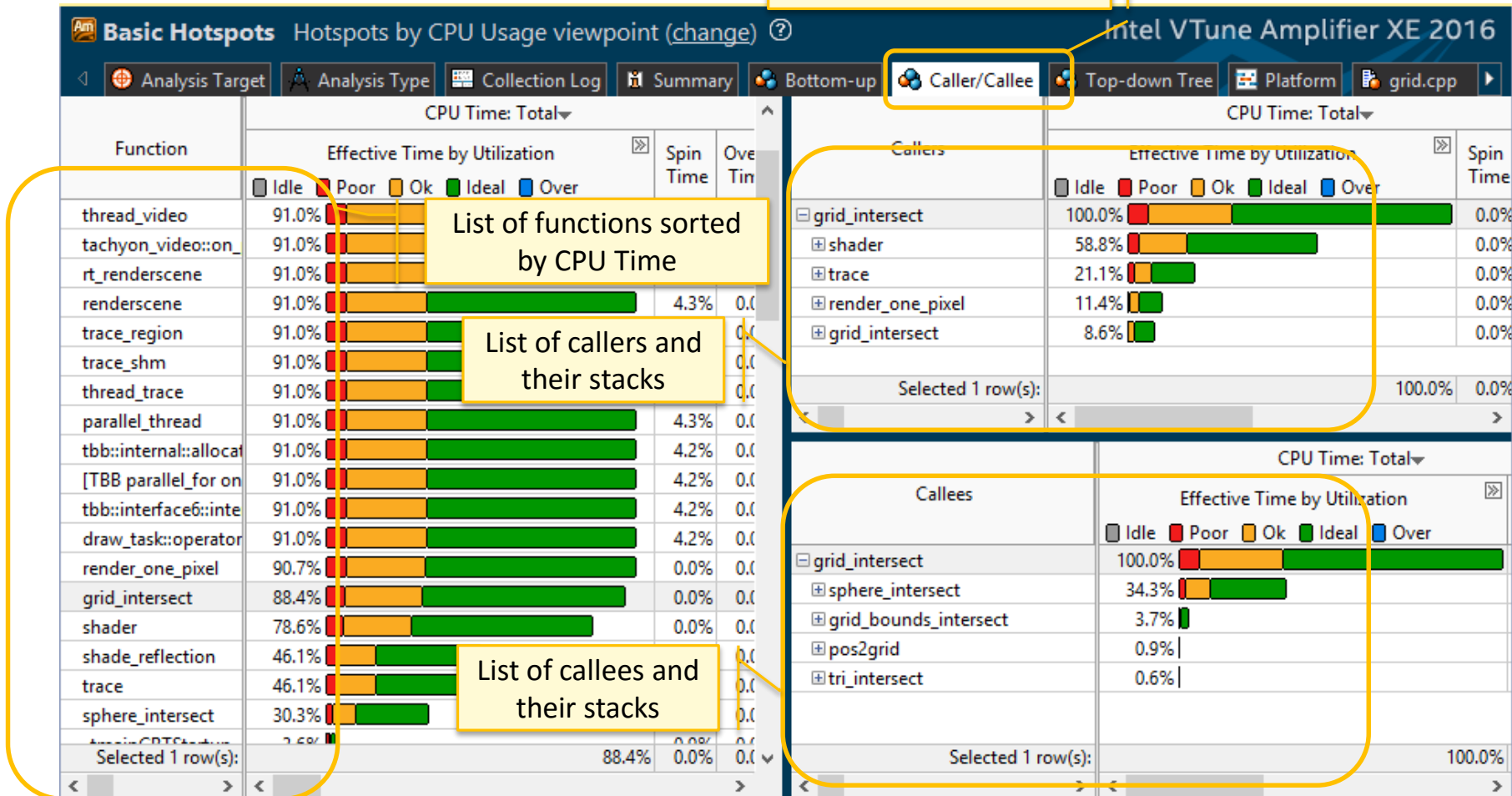
Clicking on the Top-Down Tree tab changes stack representation in the Grid



Caller/Callee View

GUI Layout

Select a function in the Bottom-Up and find the caller/callee



Adding User Marks to the Timeline

GUI Controls

The screenshot displays the Intel VTune GUI interface. On the left, a control panel contains buttons for 'Start', 'Start Paused', 'Project Properties', 'Resume', 'Pause', 'Stop', 'Cancel', and 'Mark Timeline'. The 'Start Paused' button is highlighted with a yellow box and an annotation: 'Start application without data collection'. The 'Resume' button is also highlighted with a yellow box and an annotation: 'Resume data collection when needed'. The 'Mark Timeline' button is highlighted with a yellow box and an annotation: 'Click "Mark Timeline" during collection'. Below the control panel, a timeline view shows a horizontal axis with time markers from 5s to 40.538s. A yellow vertical bar is positioned at approximately 40.538s, with an annotation: 'Observe the mark on the Time Line'. The timeline view also shows a list of threads on the left, including 'wWinMainCRTStartu...', 'Thread (0x1598)', and several 'TBB Worker Thread ...' entries. A yellow box highlights the 'Mark Timeline' button and the yellow vertical bar on the timeline, with an annotation: 'Observe paused region on the Time Line'. The bottom of the timeline view shows 'CPU Usage' and 'Thread Concurrency' graphs.

Start application without data collection

Resume data collection when needed

Click "Mark Timeline" during collection

Observe the mark on the Time Line

Observe paused region on the Time Line

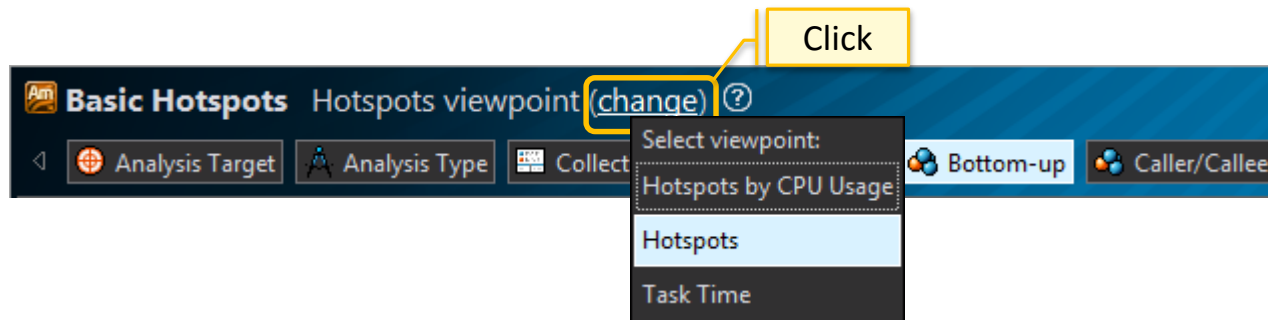
Key Result Analysis and GUI Concepts

Result Analysis

GUI Concepts

Viewpoints

- It is a pre-defined view that determines what needs to be displayed in the grid and timeline for a given analysis type
- An analysis type may support more than one view points
- To change viewpoints, select a viewpoint by clicking on

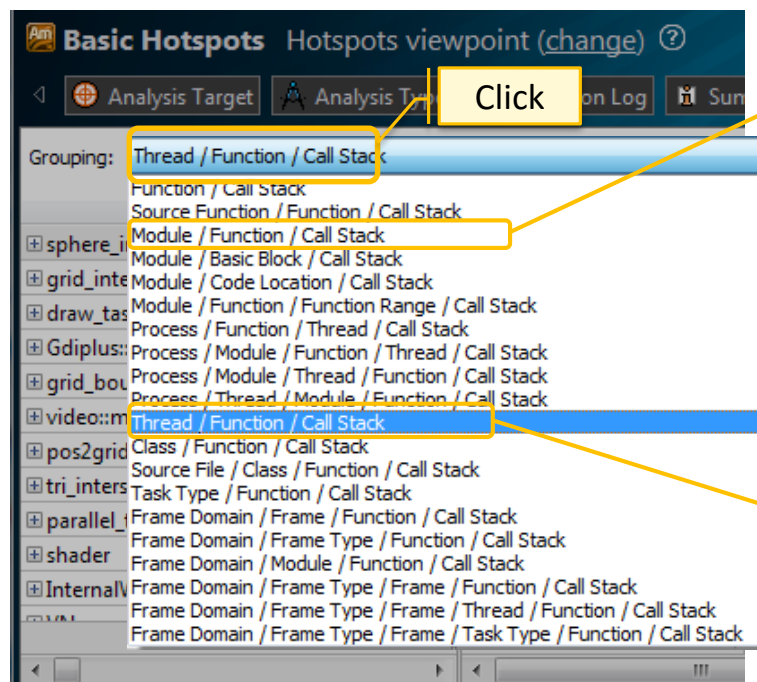


Result Analysis

GUI Concepts

Groupings

- Each analysis type has many viewpoints
- Each viewpoint has pre-defined groupings
- Allows you to analyze the data in different hierarchies and granularities



Grouping: Module / Function / Call Stack

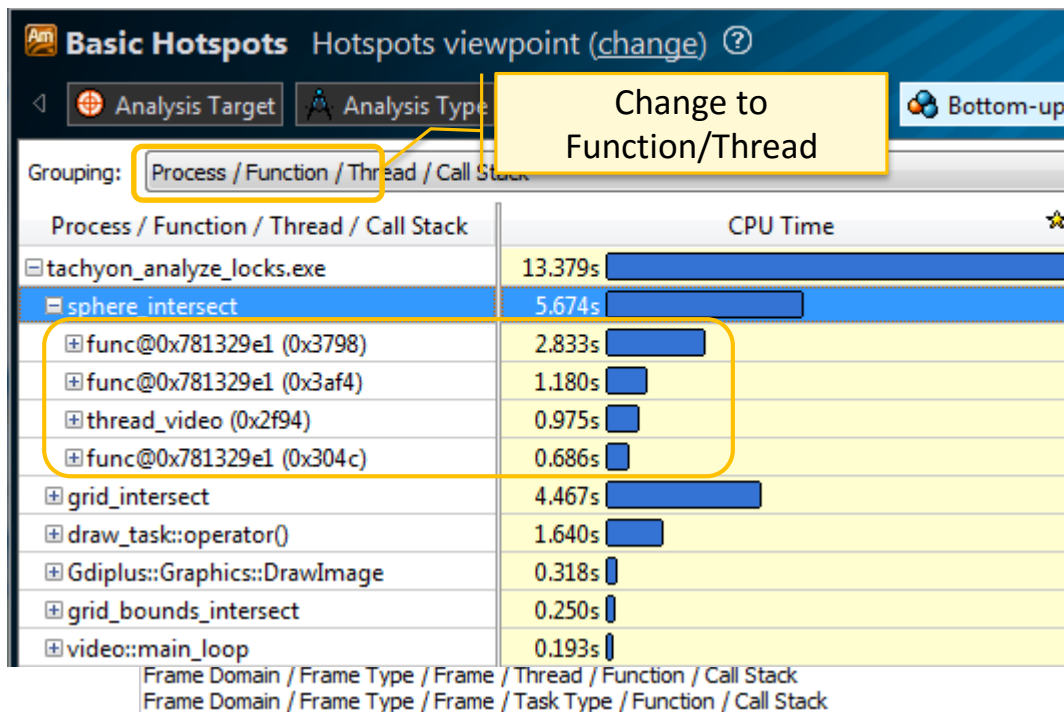
Module / Function / Call Stack	CPU Time
tachyon_analyze_locks.exe	13.367s
sphere_intersect	5.674s
grid_intersect	5.674s
grid_intersect	4.467s
intersect_objects	4.053s
grid_intersect	0.414s
draw_task::operator()	1.640s

Grouping: Thread / Function / Call Stack

Thread / Function / Call Stack	CPU Time
func@0x781329e1 (0x3798)	5.983s
func@0x781329e1 (0x3af4)	2.598s
thread_video (0x2f94)	2.384s
sphere_intersect	0.975s
grid_intersect	0.975s
intersect_objects	0.957s
shader	0.625s
trace	0.333s

Viewpoints and Groupings

For example, pre-defined groupings can be used to determine load imbalance



Key Concepts

Results Comparison

VTune™ Amplifier XE allows comparison of two similar runs

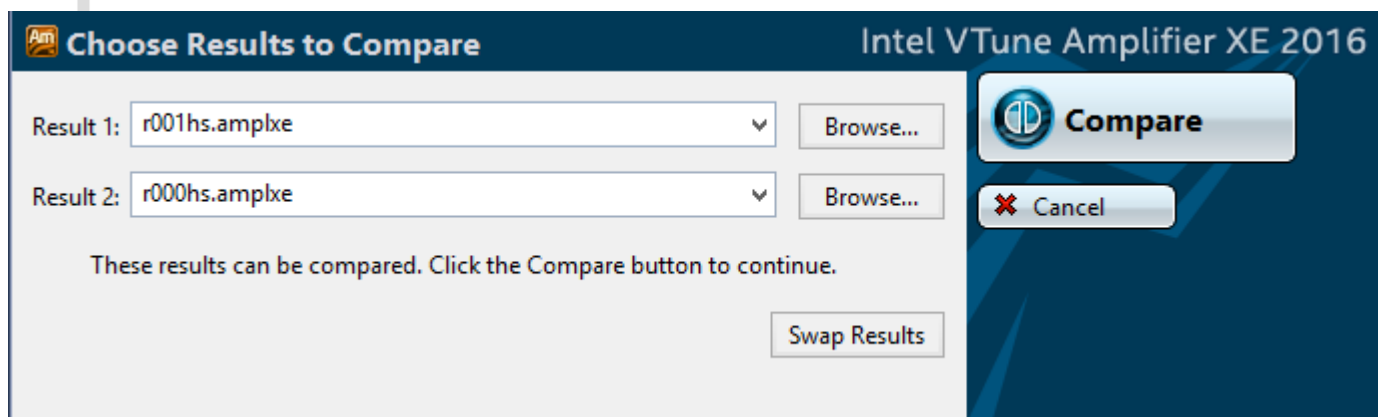
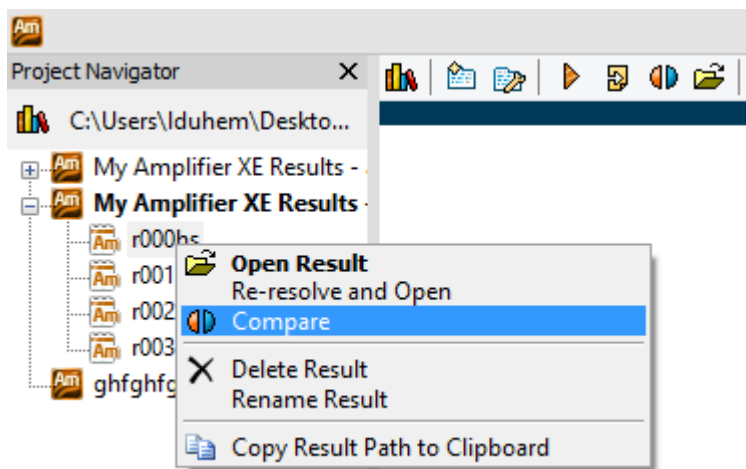
Extremely useful for:

- Benchmarking
- Regression analysis
- Testing

During performance optimization work source code may change

- Binary recompiled: compare based on source function
- Inside a function: compare based on functions level
- Functions changed: group by source files and compare
- Source files changed: compare by modules

Results Comparison

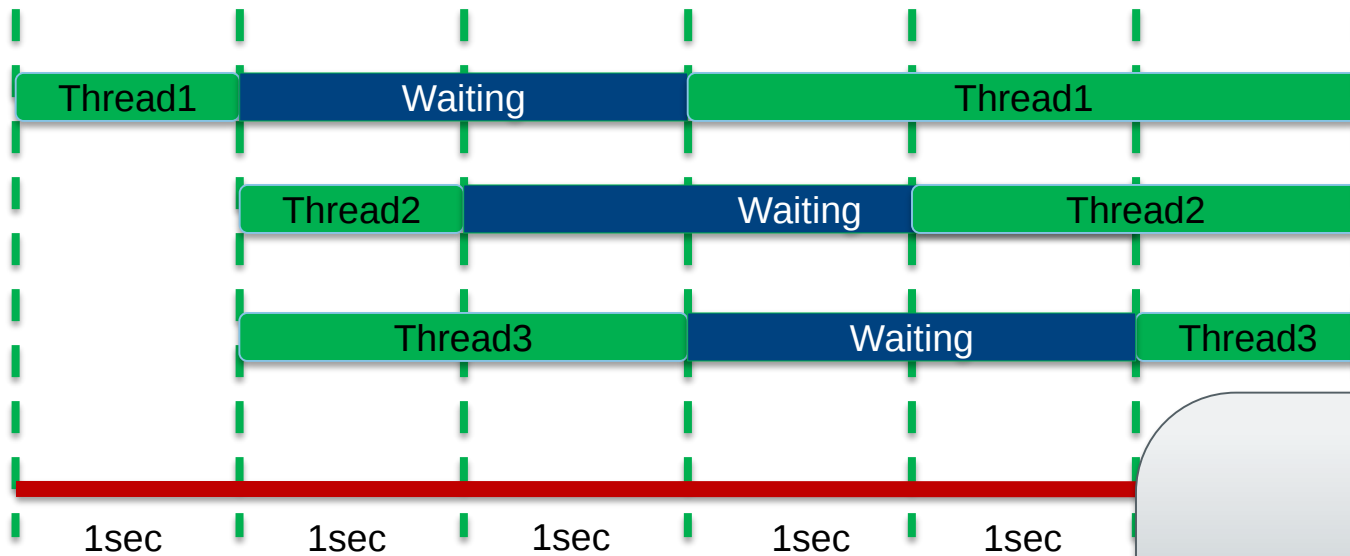


Analysis Types Revisited

Lab Activities

Performance profiling

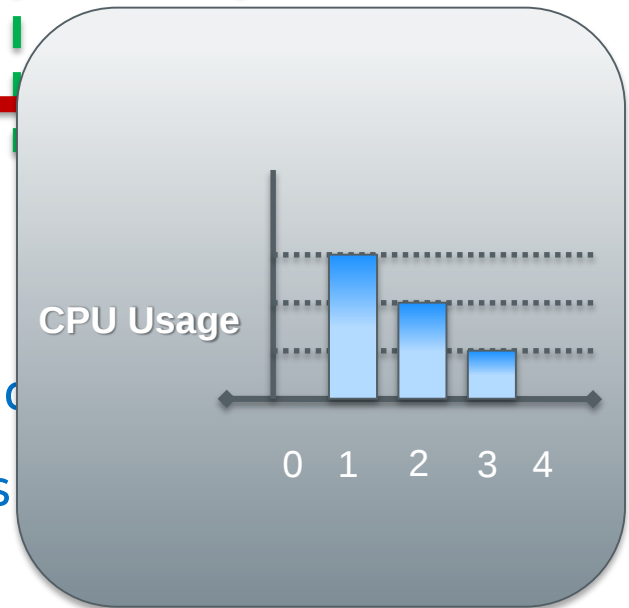
CPU Usage



Elapsed Time: 6 seconds

CPU Time: T1 (4s) + T2 (3s) + T3 (3s) = 10 seconds

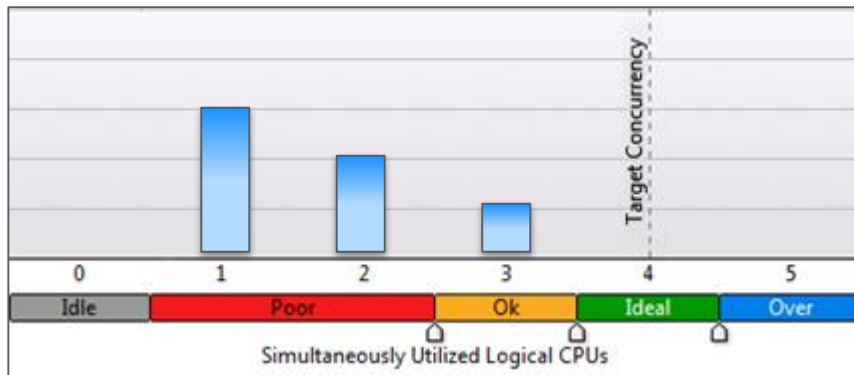
Wait Time: T1(2s) + T2(2s) + T3 (2s) = 6 seconds



CPU Usage

How it's presented by VTune Amplifier

Summary View: CPU Usage Histogram



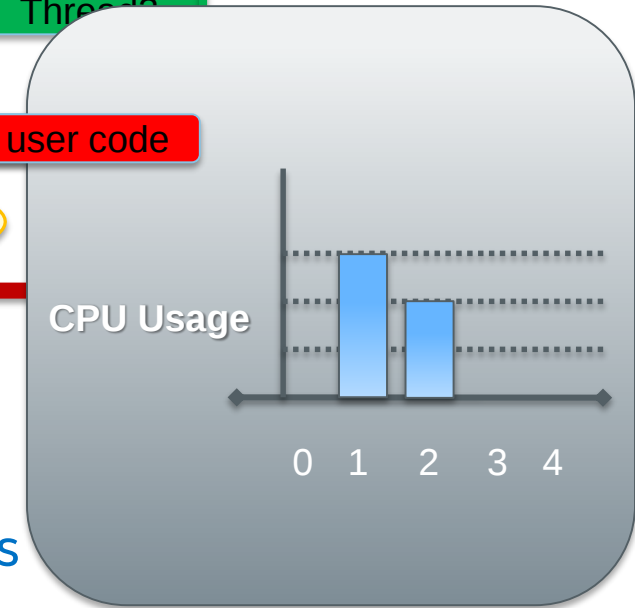
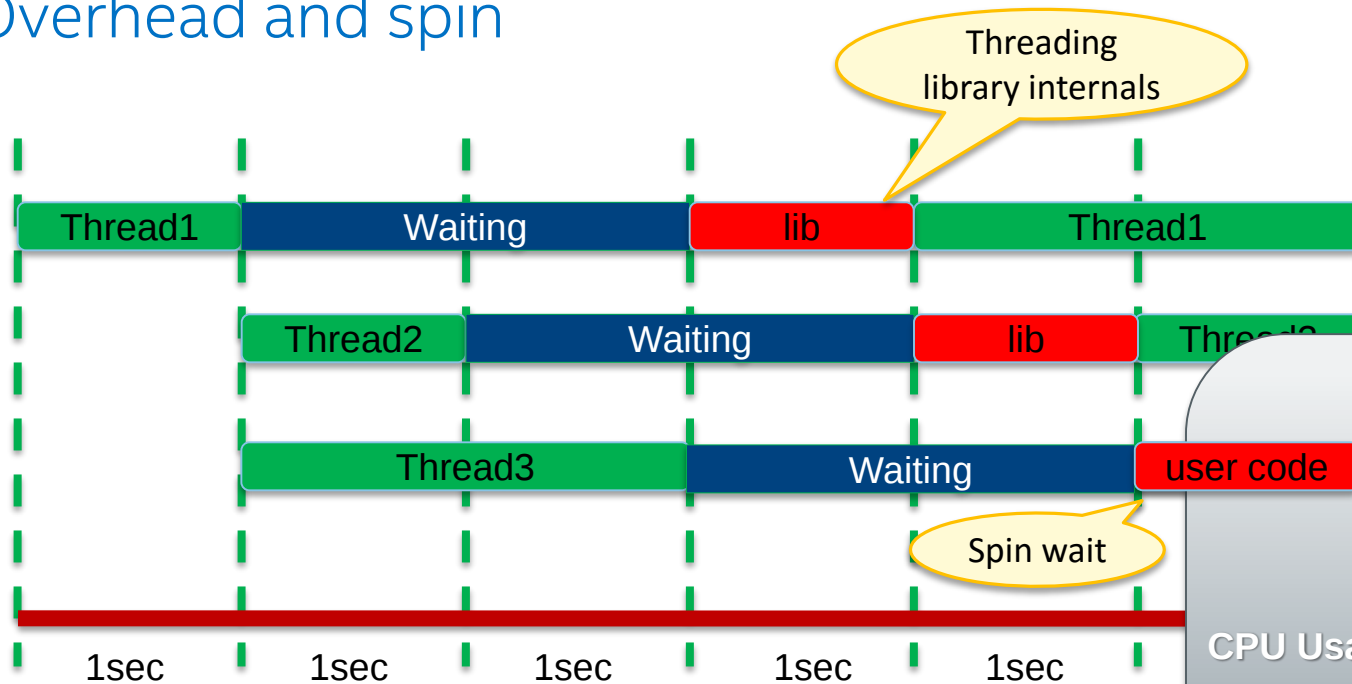
Only CPU Time measured
Wait Time is not counted
in Hotspots

Bottom-Up View: CPU Time

Function	CPU Time	By CPU Utilization
My_Func()	10 s	

Performance profiling

Overhead and spin



Elapsed Time: 6 seconds

CPU Time: $T1 (4s) + T2 (3s) + T3 (3s) = 10 \text{ seconds}$

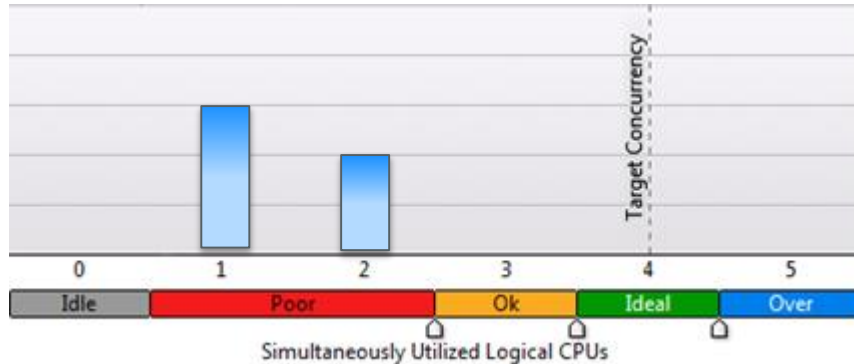
Wait Time: $T1(2s) + T2(2s) + T3 (2s) = 6 \text{ seconds}$

Overhead and spin Time: $T1(1s) + T2(1s) + T2(1s) = 3 \text{ s}$

CPU Usage

How it's presented by VTune Amplifier

Summary View: CPU Usage Histogram

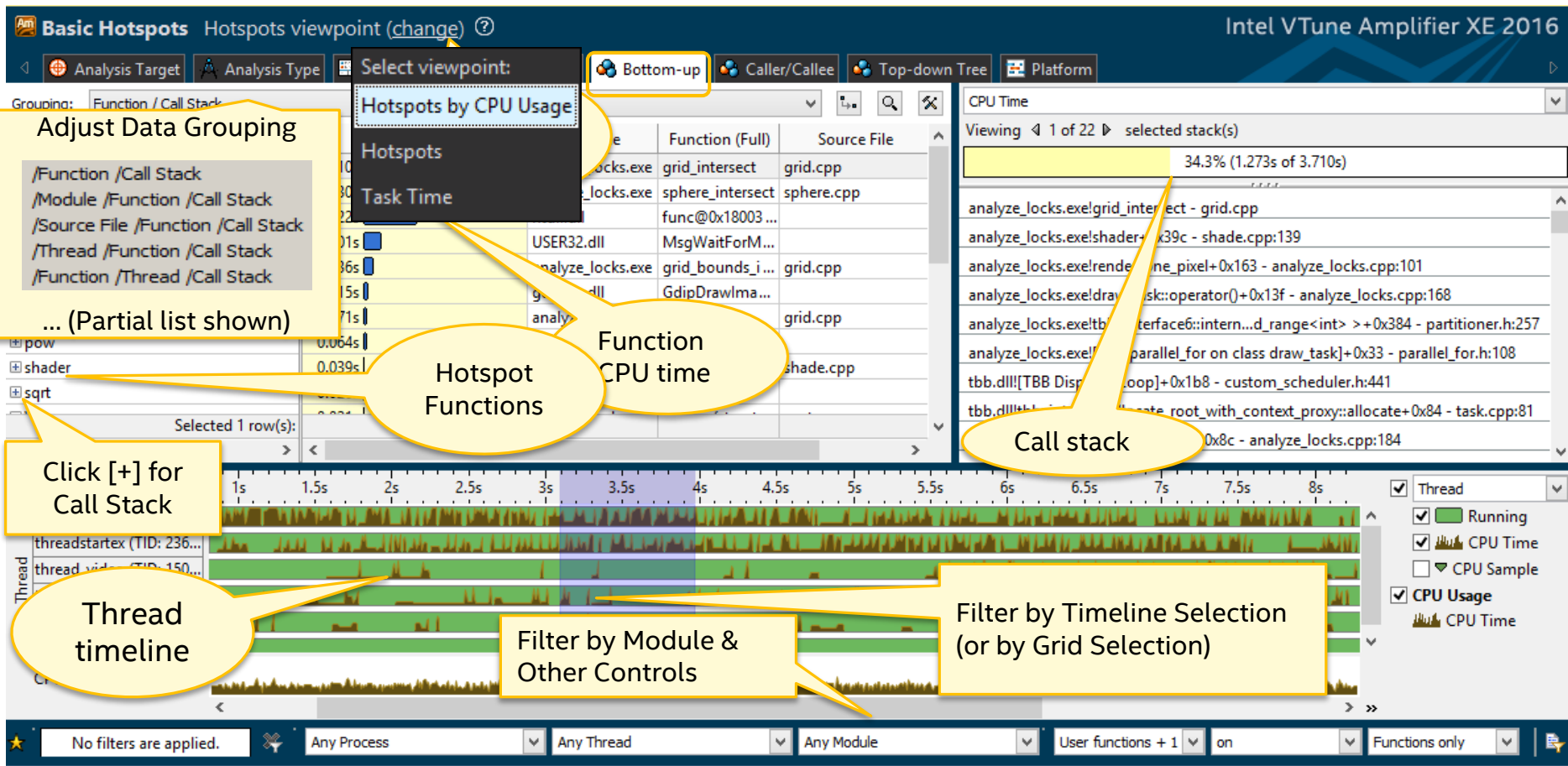


Overhead and Spin Time is not counted for CPU Usage

Bottom-Up View: CPU Time

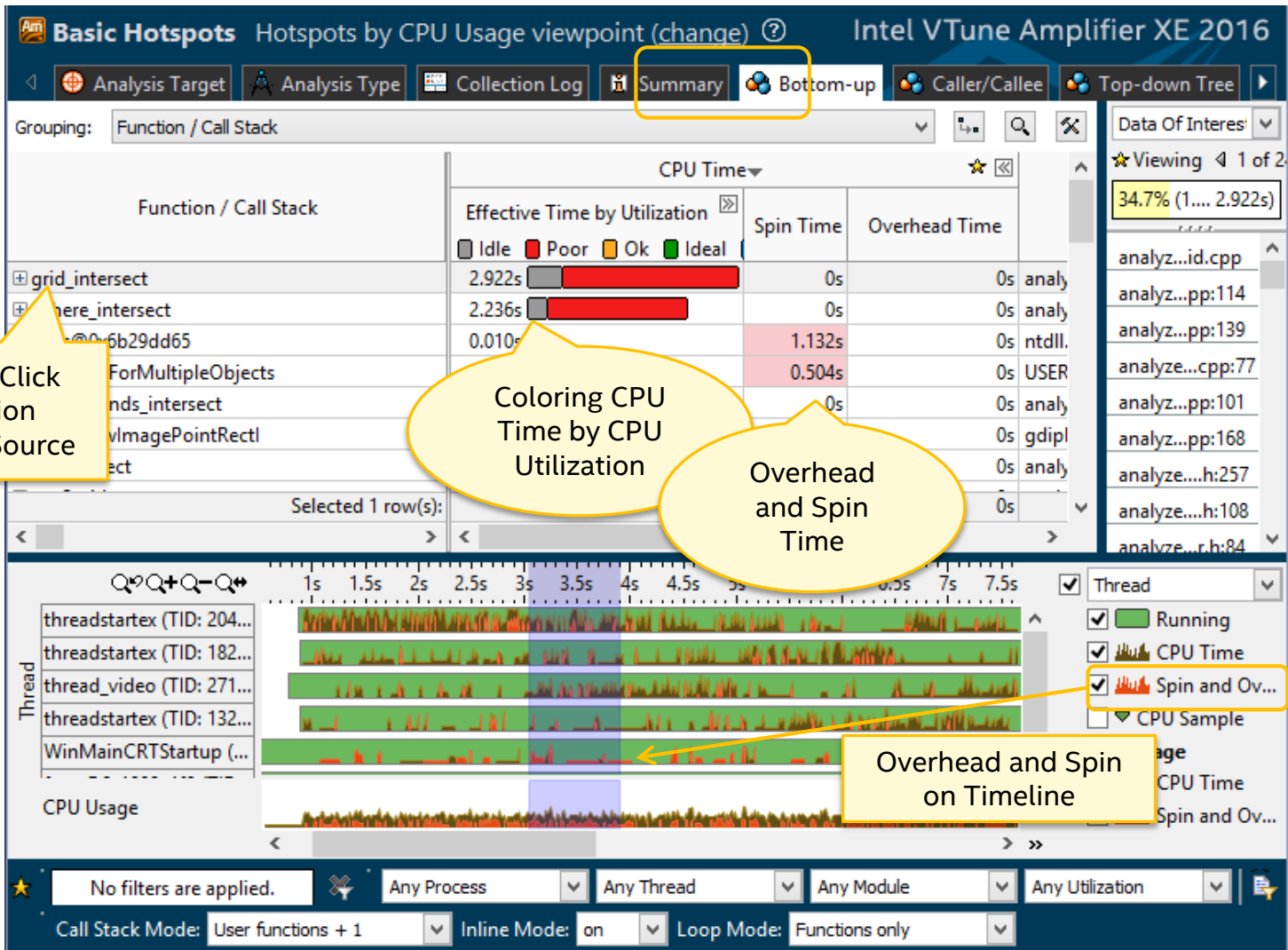
Function	CPU Time	By CPU Utilization	Overhead and Spin Time
My_Func()	10 s		4 s

Hotspot viewpoint



Hotspots analysis

Hotspot functions by CPU usage



Hotspots analysis

Source View

The screenshot displays the Intel VTune Source View and Assembly View side-by-side. The Source View on the left shows C++ code with CPU time measurements. The Assembly View on the right shows the corresponding assembly instructions. Annotations highlight key features for hotspot analysis.

Source View

So. ▲	Source	CPU Time: Total ▲
572	tmax.x += tdelta.x;	
573	curpos = nXp;	
574	nXp.x += pdeltaX.x;	
575	nXp.y += pdeltaX.y;	
576	nXp.z += pdeltaX.z;	
577	}	
578	else if (tmax.z < tmax.x) {	
579	cur = g->cells[voxindex];	
580	while (cur != NULL) {	
581	if (ry->mbox[cur->obj->id] != ry->mbox[cur->obj->id]) {	863.901ms
582	ry->mbox[cur->obj->id] = ry->mbox[cur->obj->id];	470.596ms
583	cur->obj->methods->intersect(cur->obj->methods->intersect);	344.992ms
584	}	
585	}	365.913ms
586	}	
587	curvox.z += step.z;	11.005ms
588	if (ry->maxdist < tmax.z curvox.z > tmax.z) {	20.980ms
589	break;	
590	voxindex += step.z;	
591	tmax.z += tdelta.z;	

Assembly View

Address ▲	Sour...	Assembly	CPU Time: Total ▲
0x40e0f1	581	cmp dword ptr [eax+edx*4], ecx	542.360ms
0x40e0f4	581	jz 0x40e10d <Block 49>	
0x40e0f6		Block 47:	
0x40e0f6	582	mov edx, dword ptr [esi+0x4]	375.228ms
0x40e0f9	582	mov edx, dword ptr [edx]	52.595ms
0x40e0fb	582	mov dword ptr [eax+edx*4], ecx	42.774ms
0x40e0fe	583	mov dword ptr [esi+0x4], ecx	54.972ms
0x40e101	583	mov ecx, dword ptr [eax+edx*4]	37.383ms
0x40e104	583	mov ecx, dword ptr [eax+edx*4]	6.785ms
0x40e106	583	push edi	32.718ms
0x40e107	583	push eax	31.020ms
0x40e108	583	call edx	204.404ms
0x40e10a		Block 48:	
0x40e10a	583	add esp, 0x8	
0x40e10d		Block 49:	
0x40e10d	585	mov ecx, dword ptr [eax+edx*4]	69.080ms
0x40e10f	585	mov ecx, dword ptr [eax+edx*4]	96.833ms
0x40e111	585	jnz 0x40e0e6 <Block 46>	
0x40e113		Block 50:	
0x40e113	580	moved xmm0, qword ptr [esp+0x4]	9.909ms

Annotations:

- Source View:**
 - Self and Total Time on Source / Asm (points to CPU Time column)
 - Quick Asm navigation: Select source to highlight Asm (points to line 582)
 - Quickly scroll to hot spots. Scroll Bar "Heat Map" is an overview of hot spots (points to scroll bar)
- Assembly View:**
 - Right click for instruction reference manual (points to instruction at 0x40e0fb)
 - Click jump to scroll Asm (points to jump instruction at 0x40e111)



Find the Performance Hotspot

Lab 1

Advanced Hotspot analysis

Uses Intel's CPU hardware performance collectors

Higher resolution of sampling (~1 /ms)

Capable for system wide analysis (all processes running in a system)

OS modules and drivers profiling (ring 0 level)

OS context switches and threads synchronization issues

Choose Analysis Type

Analysis Type

- Algorithm Analysis
 - Basic Hotspots
 - Advanced Hotspots**
 - Concurrency
- CPU Specific Analysis
- SoC Advanced Analysis
- Knights Corner Platform Analysis
- Custom

Advanced Hotspots Copy

Identify time-consuming code in your application. Advanced Hotspots analysis (formerly, Lightweight Hotspots) uses the OS kernel support or VTune Amplifier kernel driver to extend the Hotspots analysis by collecting call stacks, context switch and statistical call count data as well as analyzing the CPI (Cycles Per Instruction) metric. By default, this analysis uses higher frequency sampling at lower overhead compared to t...

CPU sampling interval, ms:

Select a level of details provided with event-based sampling collection levels cause higher overhead.

- ☐ Hotspots
- ☒ Hotspots, stacks and context switches
- ☐ Hotspots, call counts, stacks and context switches

Start

Start Paused

Project Properties

Start the Analysis

Advanced Hotspot Analysis

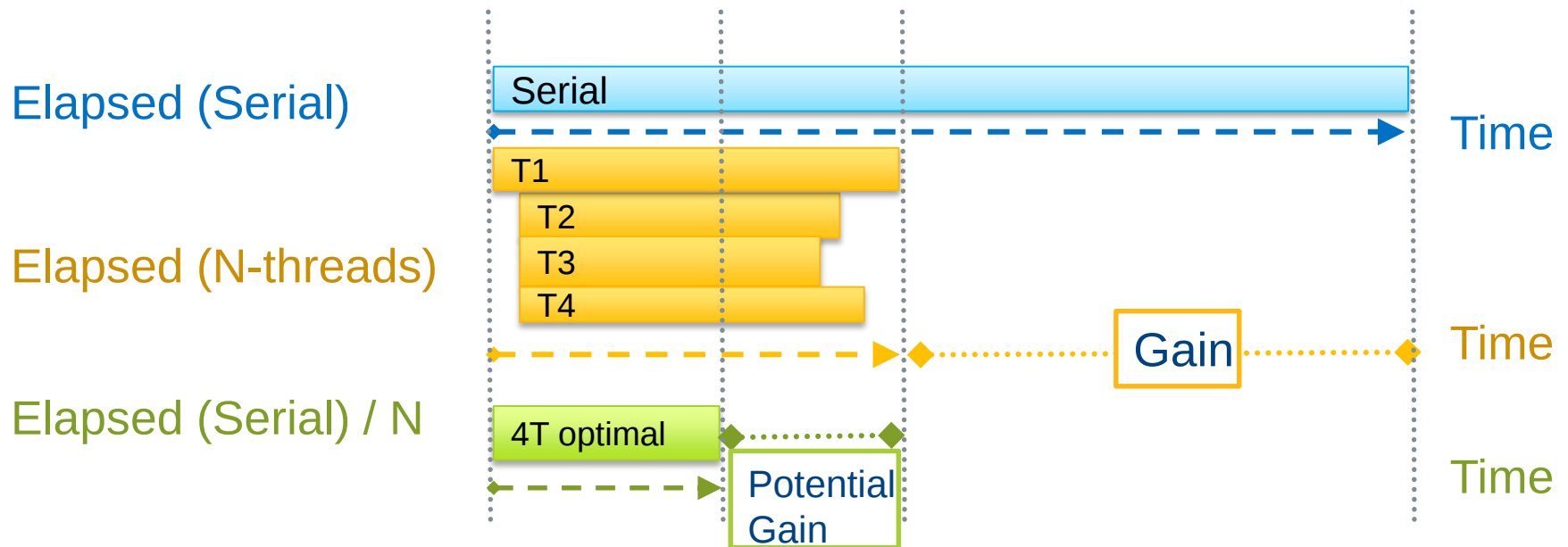
Select level of data collected

Select sampling interval

Reminding methodology of performance profiling and tuning

How to optimize the Hotspots?

- Maximize CPU utilization and minimize elapsed time
 - Ensure CPU is busy all the time
 - All Cores busy – parallelism (high concurrency)

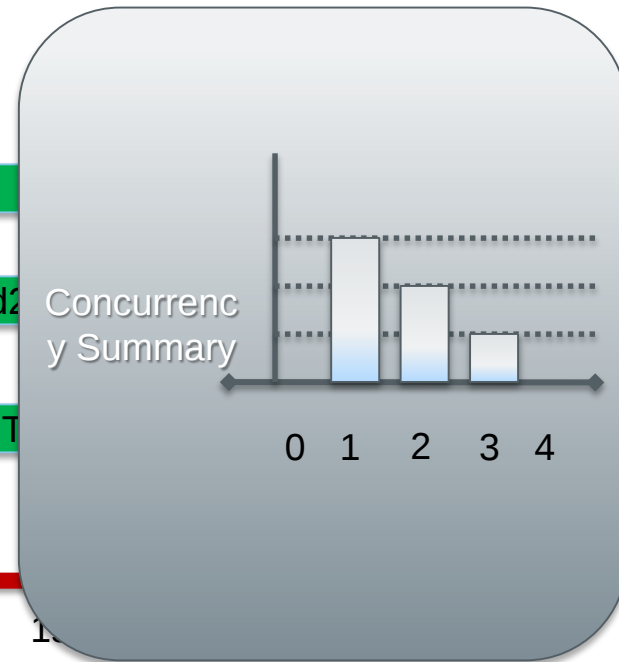
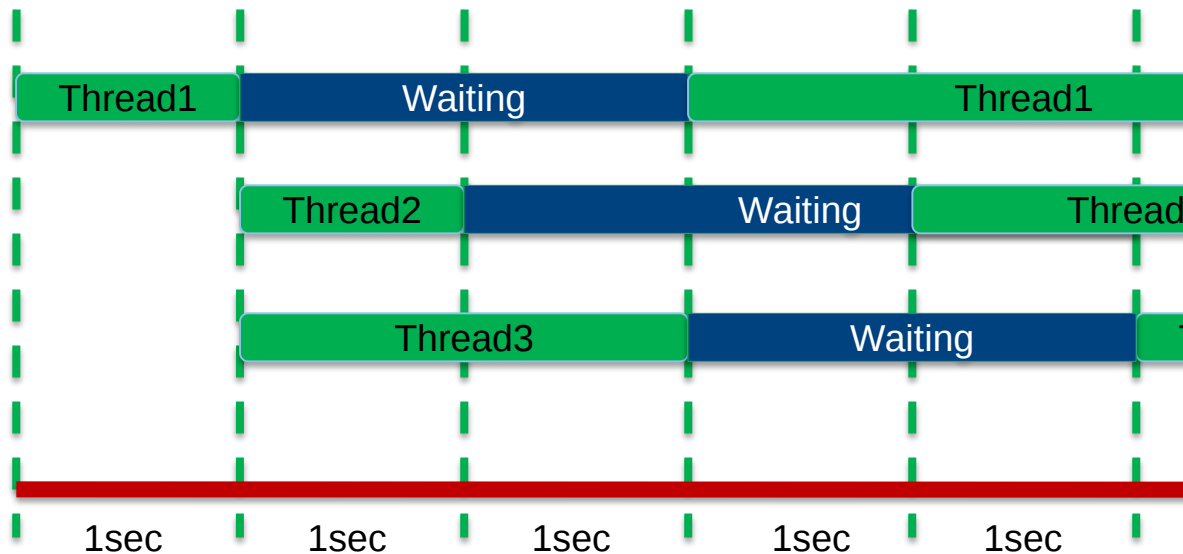


Performance profiling

Concurrency

Concurrency -

Is a measurement of the number of active threads



Parallelism/Concurrency Analysis

For Parallelism / Concurrency analysis,

- Stack sampling is done just like in Hotspots analysis
- Wait functions are instrumented (e.g. WaitForSingleObject, EnterCriticalSection)
- Signal functions are instrumented (e.g. SetEvent, LeaveCriticalSection)
- I/O functions are instrumented (e.g. ReadFile, socket)

Choose Analysis Type

Analysis Type

Algorithm Analysis

- Basic Hotspots
- Advanced Hotspots
- Concurrency**
- Locks and Waits

Microarchitecture Analysis

- Platform Analysis
- System Platform Analysis
- Custom Analysis

Concurrency

Analyze how your application is using available logical CPUs, discover where parallelism is incurring synchronization overhead, and identify potential candidates for parallelization. This analysis type uses user-mode sampling and tracing collection. Press F1 for more details.

CPU sampling interval, ms: 10

- ☐ Analyze user tasks
- ☐ Analyze Intel runtimes and user synchronization
- ☐ Analyze GPU usage

Analyze Processor Graphics hardware events: None

- ☐ Trace OpenCL kernels on Processor Graphics

Start

Start Paused

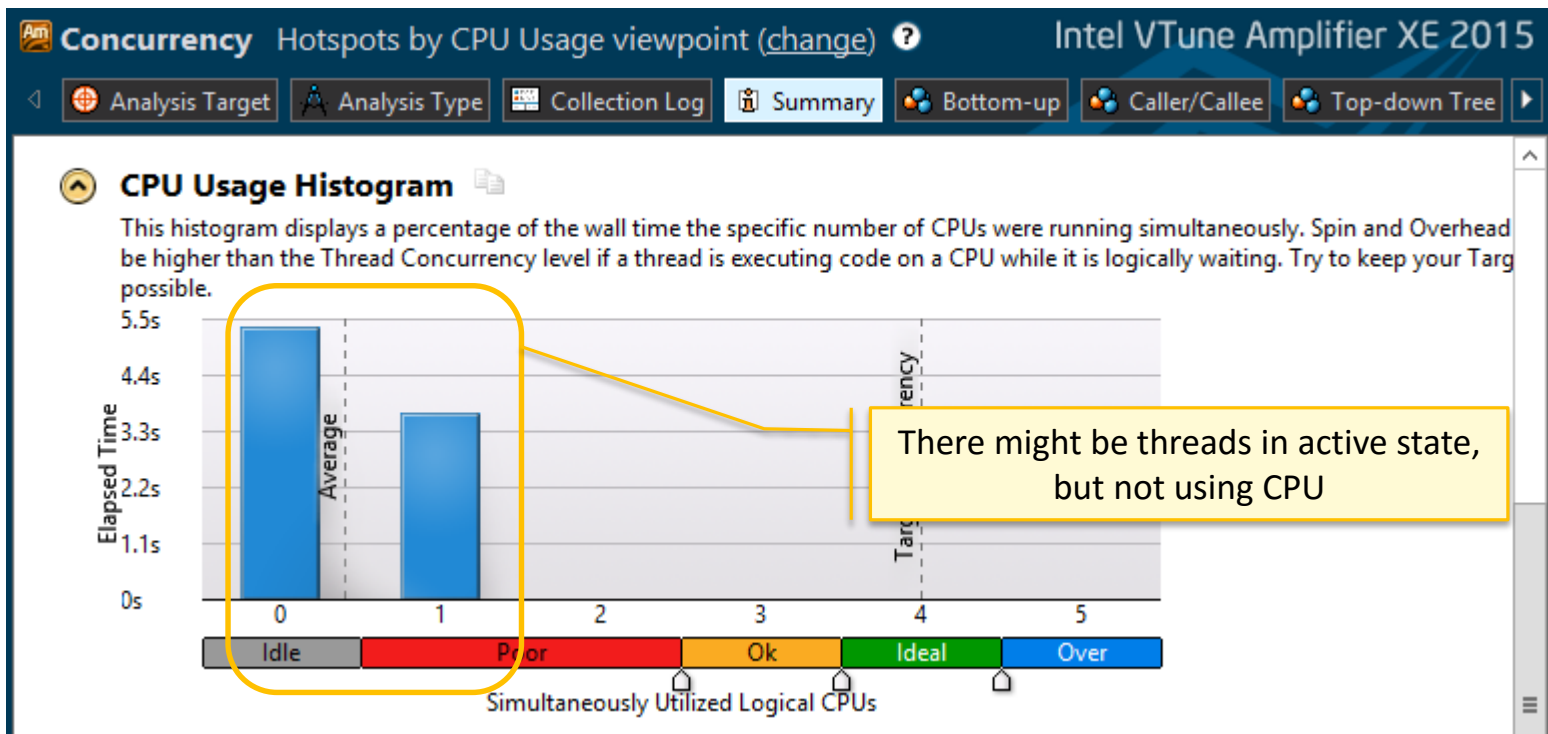
Project Properties

Start the Analysis

Concurrency Analysis

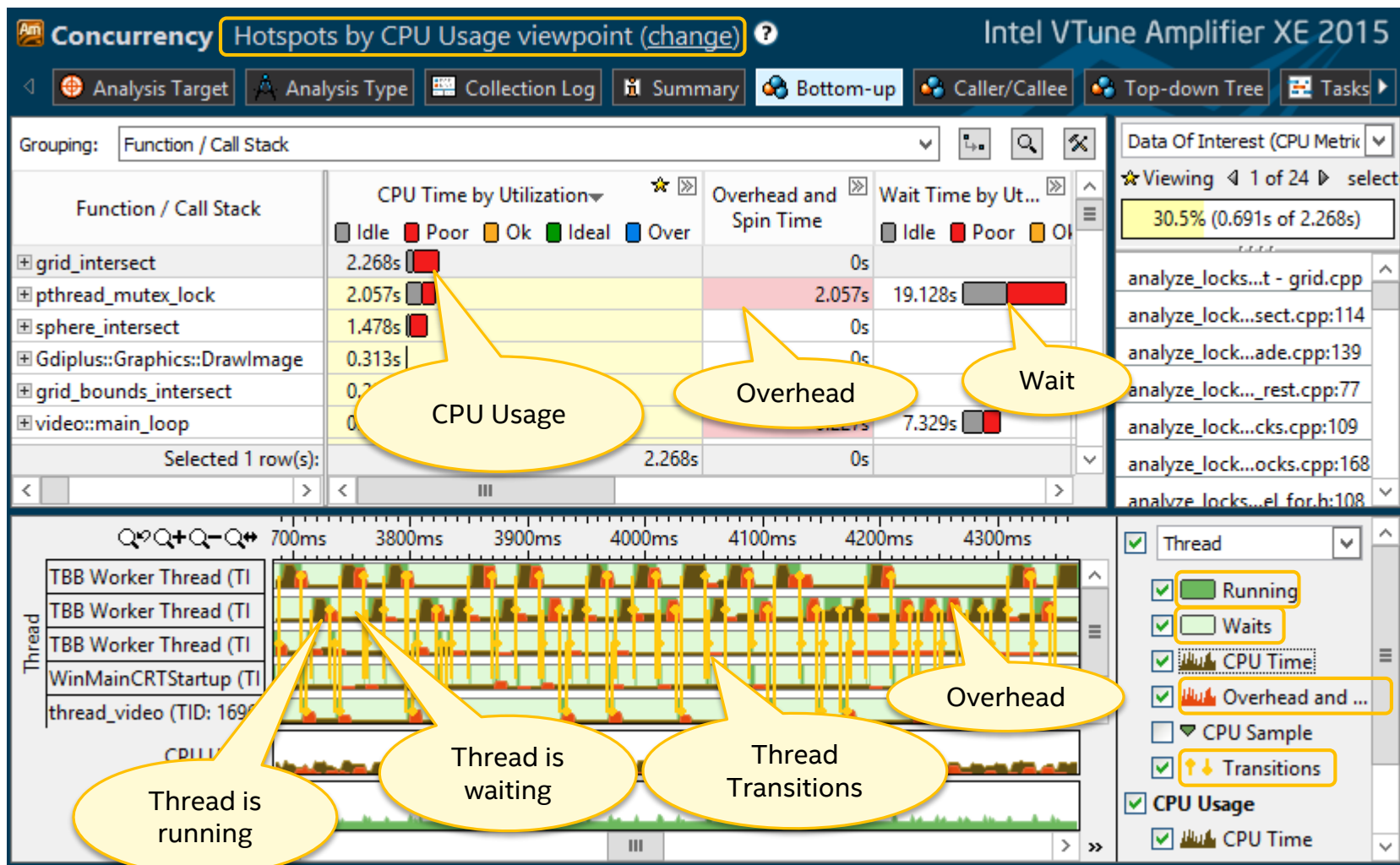
Concurrency Analysis

Summary view. CPU Usage Histogram



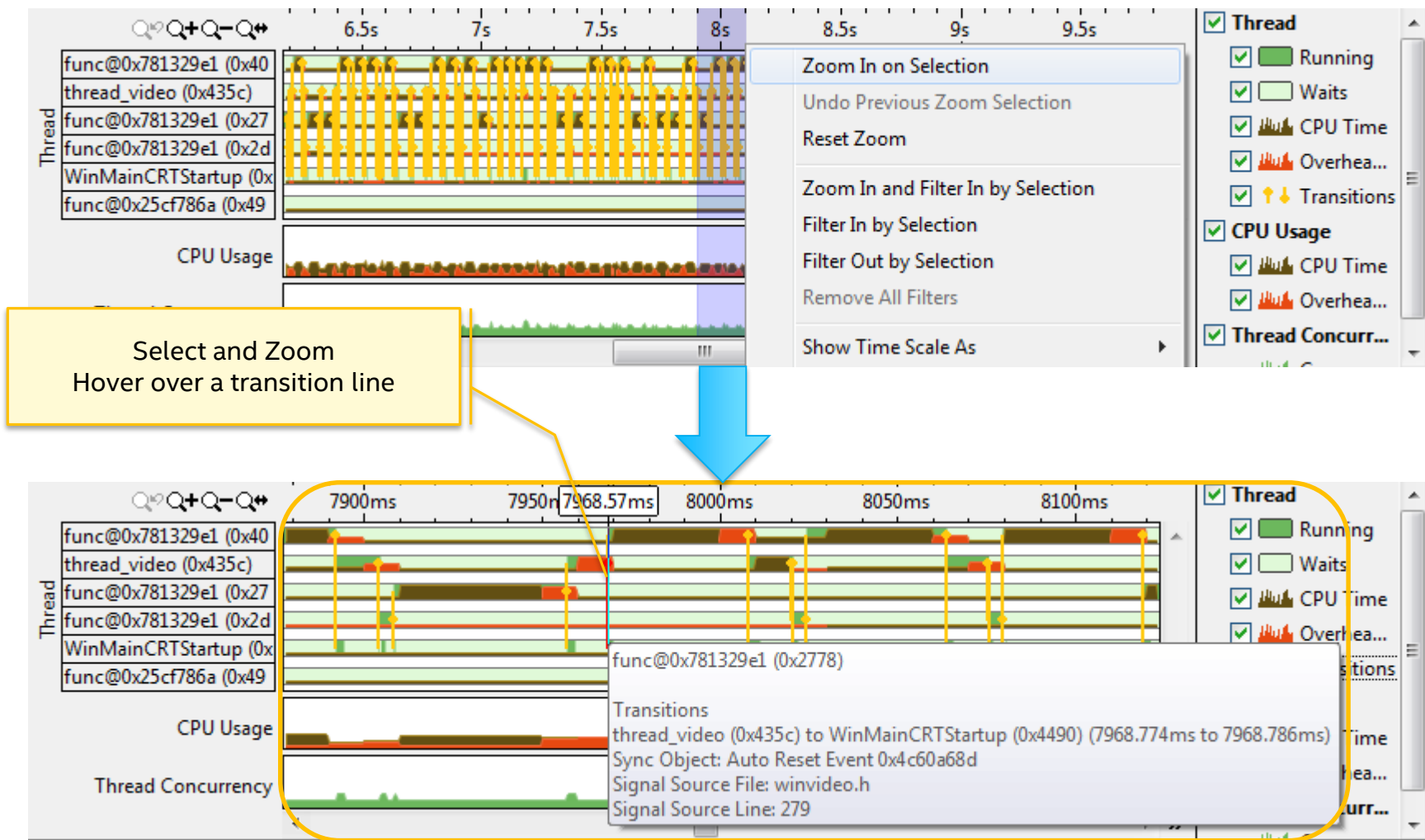
Concurrency Analysis

Bottom-Up view. CPU Usage



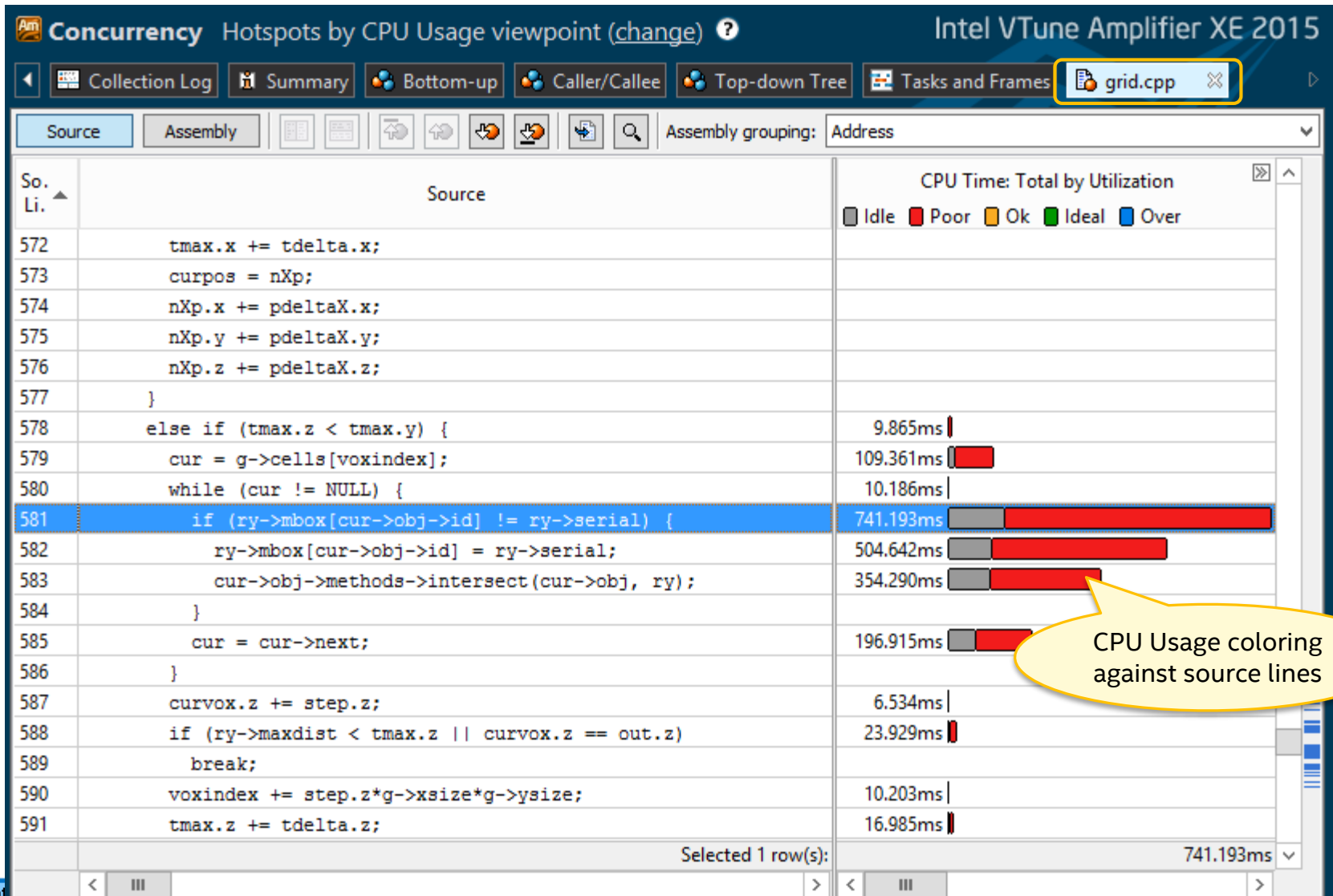
Concurrency Timeline

Investigate reasons for transitions



Concurrency Analysis

Source Code View



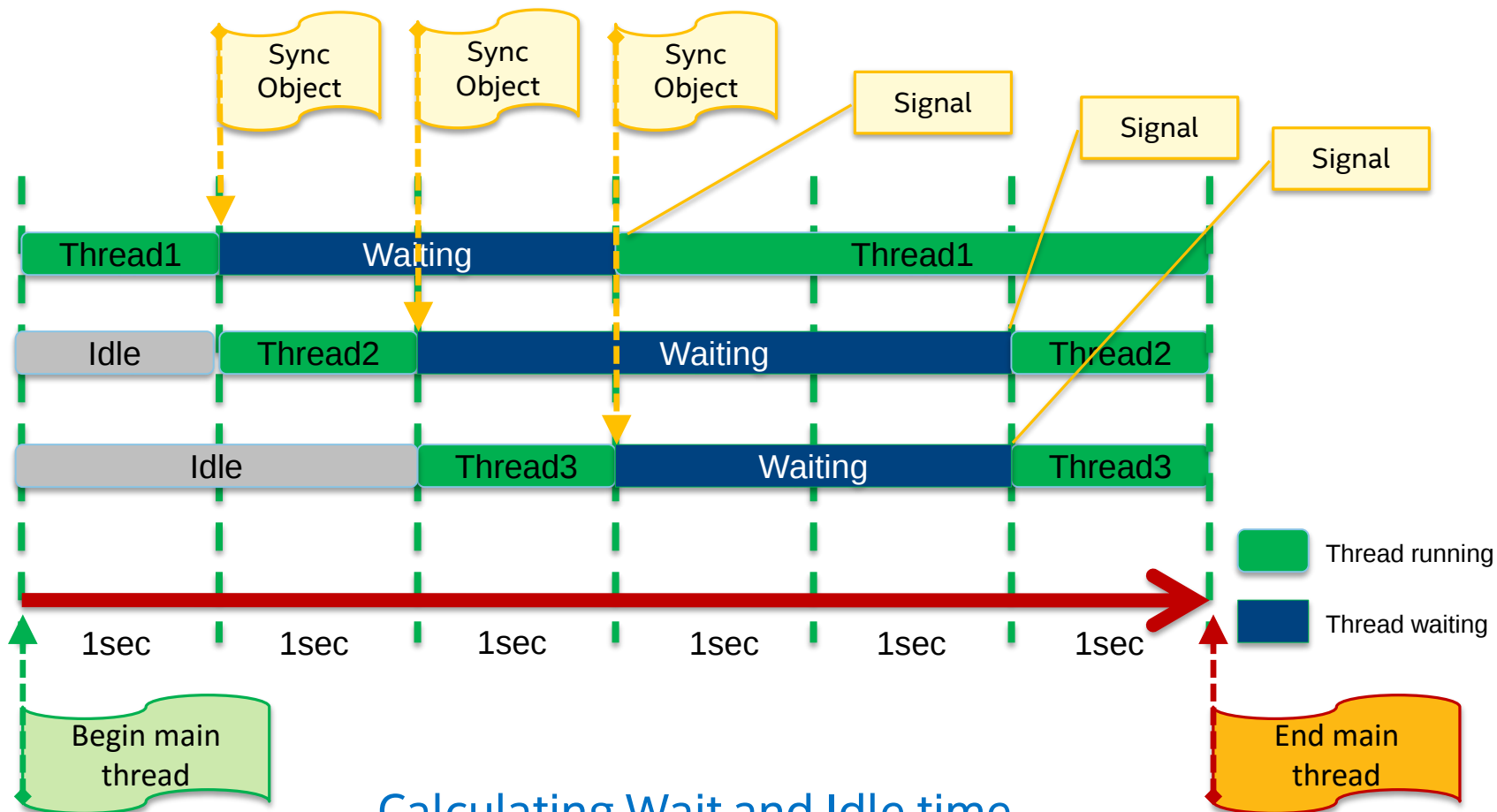


Analyzing Parallelism

Lab 2

Performance profiling

Waiting on locks

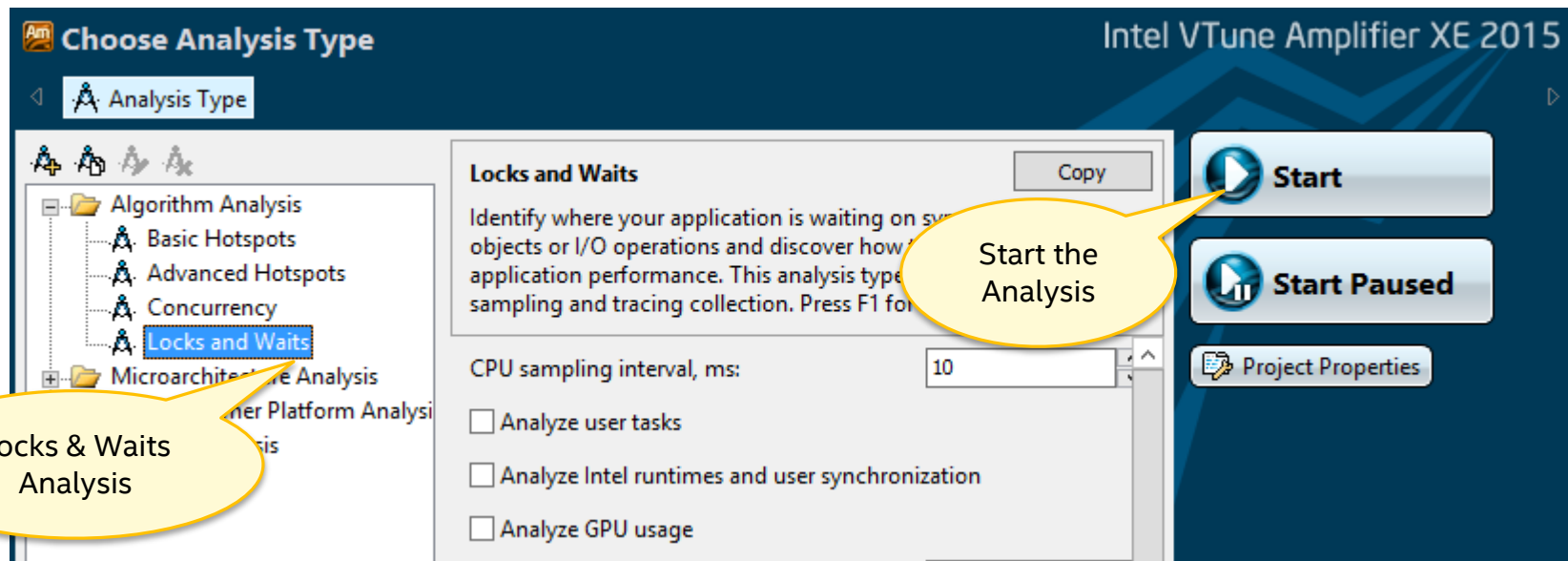


Calculating Wait and Idle time

Locks and Waits Analysis

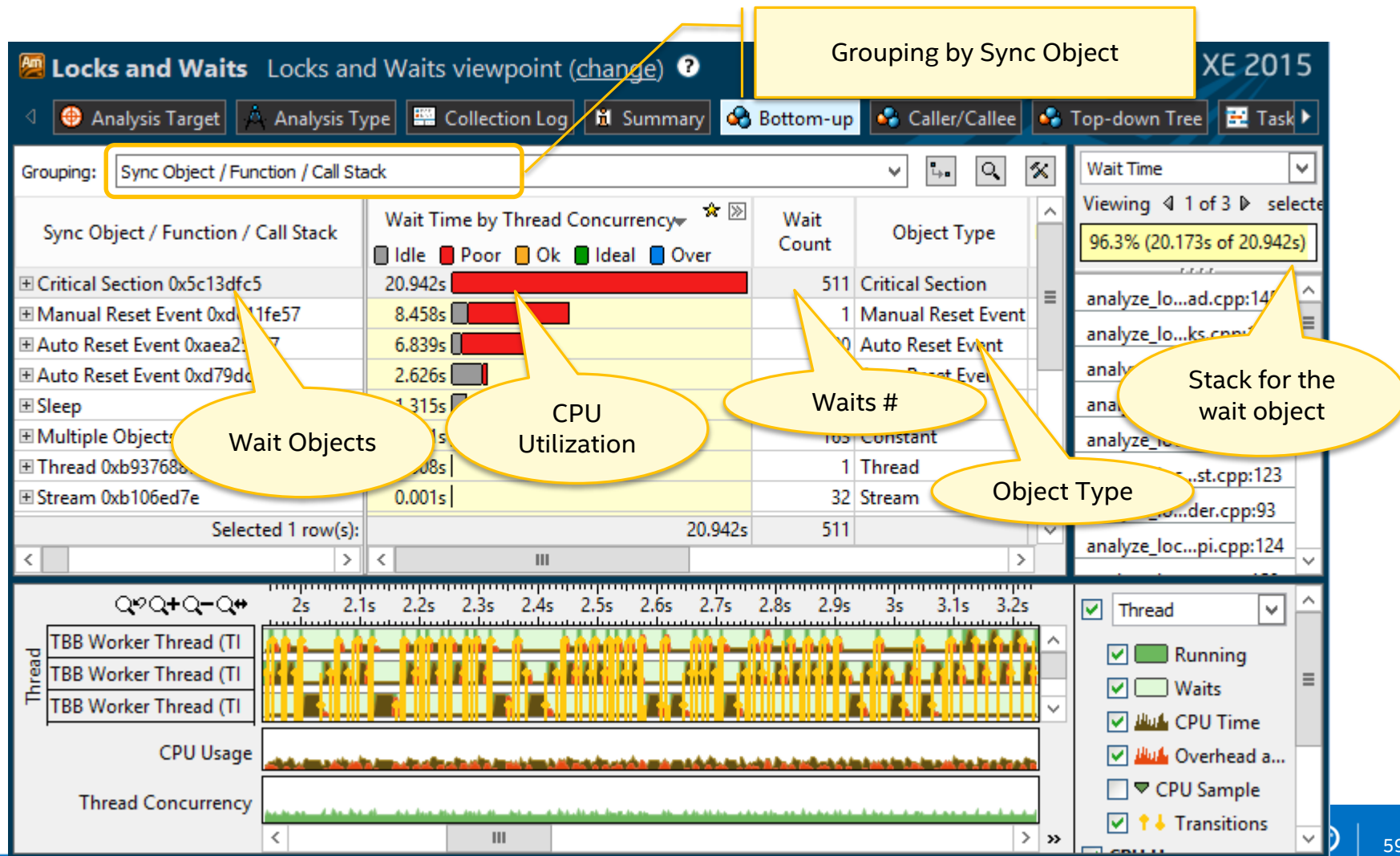
Identifies those threading items that are causing the most thread block time

- Synchronization locks
- Threading APIs
- I/O

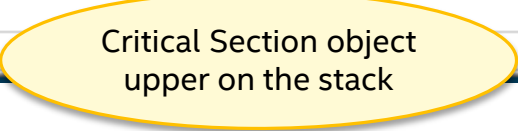


Locks and Waits Analysis

Sync/Wait objects



Source View





Finding Parallelism Issues

Lab 3

Intel® VTune™ Amplifier XE

User APIs

User APIs

- Collection Control API
- Thread Naming API
- User-Defined Synchronization API
- Task API
- User Event API
- Frame API
- JIT Profiling API

User API

Enable you to

- control collection
- set marks during the execution of the specific code
- specify custom synchronization primitives implemented without standard system APIs

To use the user APIs, do the following:

- Include **ittnotify.h**, located at <install_dir>/include
- Insert **__itt_*** notifications in your code
- Link to the **libittnotify.lib** file located at <install_dir>/lib

Windows & Linux Versions Available

Stand-alone GUI, Command line, Visual Studio Integration

Microsoft Windows* OS

- Windows 7*, Windows 8 and 8.1 Desktop*
- Windows Server* 2003, 2008
- Microsoft Visual Studio* 2010, 2012 and 2013
- Standalone GUI and command line
- IA32 and Intel® 64



Linux* OS

- RHEL*, Fedora*, SUSE*, CentOS*, Ubuntu*
- Additional distributions may also work
- Standalone GUI and command line
- IA32 and Intel® 64



Single user and floating licenses available

Installation

Windows

- Integrated into Microsoft Visual Studio, or Standalone
- Administrative privileges required for full package
- GUI and command line versions are both installed

Linux

- Standalone GUI and command line versions
- Root access not required but won't install "Event-based sampling" collectors and power collectors
 - Software collectors will work
- Data collection-only installation option
 - Enables collection with no license
 - Collection results then transferred to system with license for viewing
- Driver for event-based sampling is built at install time and can be "insmod'd" by scripts at install time and boot time

Command Line Interface

Command line (CLI) versions exist on Linux* and Windows*

- **CLI use cases:**

- Test code changes for performance regressions
- Automate execution of performance analyses

- **CLI features:**

- Fine-grained control of all analysis types and options
- Text-based analysis reports
- Analysis results can be opened in the graphical user interface

Command Line Interface

Examples

Display a list of available analysis types and preset configuration levels

```
amplxe-cl -collect-list
```

Run Hot Spot analysis on target *myApp* and store result in default-named directory, such as *r000hs*

```
amplxe-cl -c hotspots -- myApp
```

Run the Cuncurrency analysis, store the result in directory *r001par*

```
amplxe-cl -c concurrency -result-dir r001par -- myApp
```

Command Line Interface

Reporting

```
$> ampxe-cl -report summary -r  
/home/user1/examples/lab2/r003cc
```

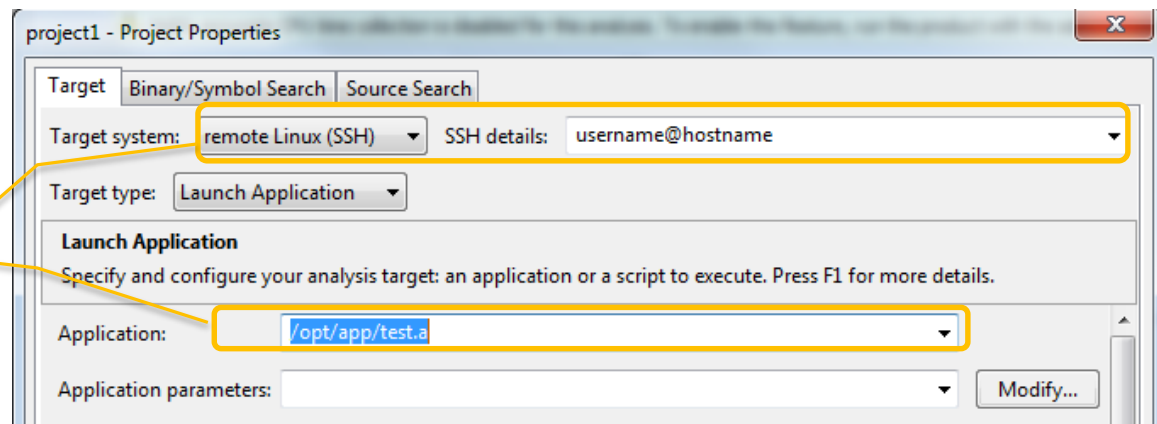
Summary

Average Concurrency:	9.762
Elapsed Time:	158.749
CPU Time:	561.030
Wait Time:	190.342
CPU Usage:	3.636
Executing actions	100 % done

Remote Data Collection



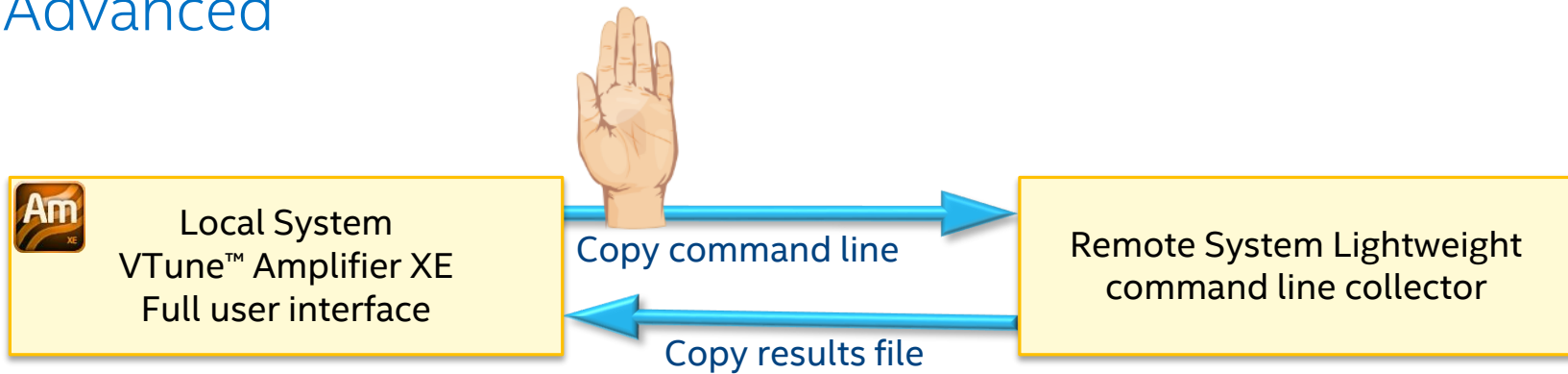
1. Setup the experiment using GUI locally
2. Configure remote target connection*
3. Specify application to run
4. Run analysis and get results copied to the Host automatically.



*Need to establish a passwordless ssh-connection

Remote Data Collection

Advanced



1. Setup the experiment using GUI locally
2. Copy command line instructions to paste buffer
3. Open remote shell on the target system
4. Paste command line, run collection
5. Copy result to your system
6. Open file using local GUI

One typical model

- Collect on Linux, analyze and display on Windows
 - The Linux machine is target
- Collect data on Linux system using command line tool
 - Doesn't require a license
- Copy the resulting performance data files to a Windows* system
- Analyze and display results on the Windows* system
 - Requires a license

Summary

The Intel® VTune Amplifier XE can be used to find:

- Source code for performance bottlenecks
- Characterize the amount of parallelism in an application
- Determine which synchronization locks or APIs are limiting the parallelism in an application
- Understand problems limiting CPU instruction level parallelism
- Instrument user code for better understanding of execution flow defined by threading runtimes

Questions?



Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

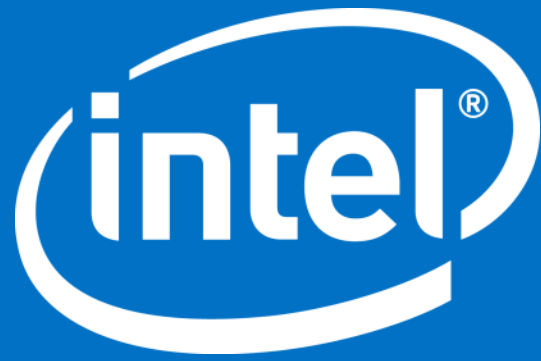
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



Backup

User API

Collection control and threads naming

Collection Control APIs

void __itt_pause (void)

Run the application without collecting data. VTune™ Amplifier XE reduces the overhead of collection, by collecting only critical information, such as thread and process creation.

void __itt_resume (void)

Resume data collection. VTune™ Amplifier XE resumes collecting all data.

Thread naming APIs

**void __itt_thread_set_name (const
__itt_char *name)**

Set thread name using char or Unicode string, where *name* is the thread name.

void __itt_thread_ignore (void)

Indicate that this thread should be ignored from analysis. It will not affect the concurrency of the application. It will not be visible in the Timeline pane.

User API

Collection Control Example

```
int main(int argc, char* argv[])
{
    doSomeInitializationWork();

    __itt_resume();
    while(gRunning) {
        doSomeDataParallelWork();
    }
    __itt_pause();

    doSomeFinalizationWork();
    return 0;
}
```

User API

User defined synchronization API example

```
long spin = 1;
. . . .
. . . .
__itt_sync_prepare((void *) &spin );
while(ResourceBusy);
    // spin wait;
__itt_sync_acquired((void *) &spin );
    // Use shared resource
__itt_sync_releasing((void *) &spin );
    // Code here should free the resource
```

User API

User Event APIs

- Useful to observe when certain events occur in your application or identify how long certain regions of code take to execute
- Event APIs enables you to annotate an application when certain events occur

```
__itt_event __itt_event_create(char *, int);  
__itt_event_start(__itt_event);  
__itt_event_end(__itt_event);
```

User API

User Event APIs reference

```
__itt_event __itt_event_create(const  
__itt_char *name, int namelen );
```

Create a user event type with the specified name. This API returns a handle to the user event type that should be passed into the following APIs as a parameter. The `namelen` parameter refers to the number of characters, not the number of bytes.

```
int __itt_event_start( __itt_event event  
);
```

Call this API with an already created user event handle to register an instance of that event. This event appears in the Timeline pane display as a tick mark.

```
int __itt_event_end( __itt_event event );
```

Call this API following a call to `__itt_event_start()` to show the user event as a tick mark with a duration line from start to end. If this API is not called, the user event appears in the Timeline pane as a single tick mark.

User API

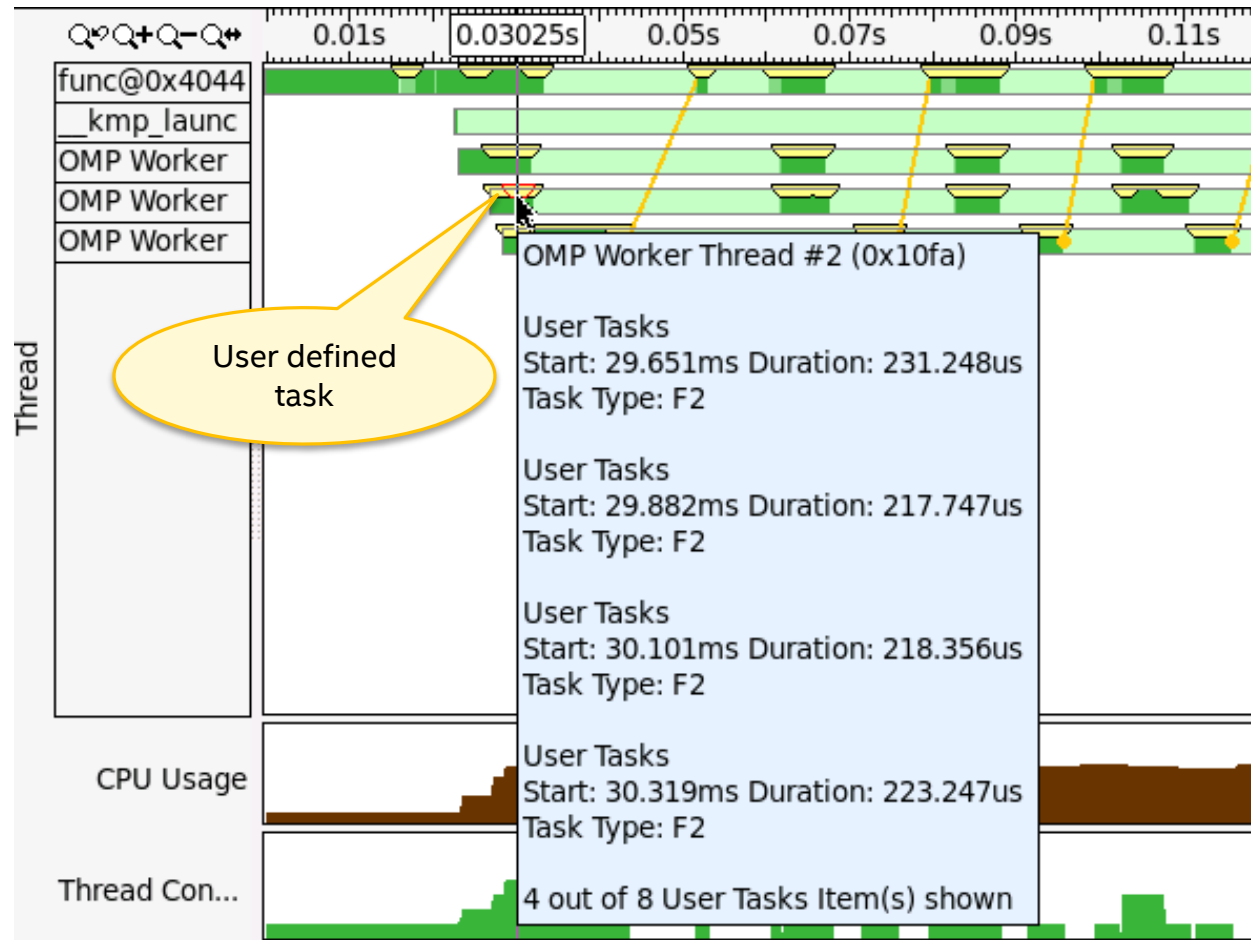
User Event APIs example

```
DWORD WINAPI aiWork(LPVOID lpArg)
{
    int tid = *((int*)lpArg);
    __itt_event aiEvent;
    aiEvent = __itt_event_create("AI Thread Work",14);

    while(gRunning) {
        WaitForSingleObject(bSignal[tid], INFINITE);
        __itt_event_start(aiEvent);
        doSomeDataParallelWork();
        __itt_event_end(aiEvent);
        SetEvent(eSignal[tid]);
    }
    return 0;
}
```

User API

Visualizing Events in the Timeline View



Performance Profiling

Frame Analysis

Frame Analysis – Analyze Long Latency Activity

Frame: a region executed repeatedly (non-overlapping)

- API marks start and finish
- Auto detect DirectX frames

Examples:

- Game – Compute next graphics frame
- Simulator – Time step loop
- Computation – Convergence loop

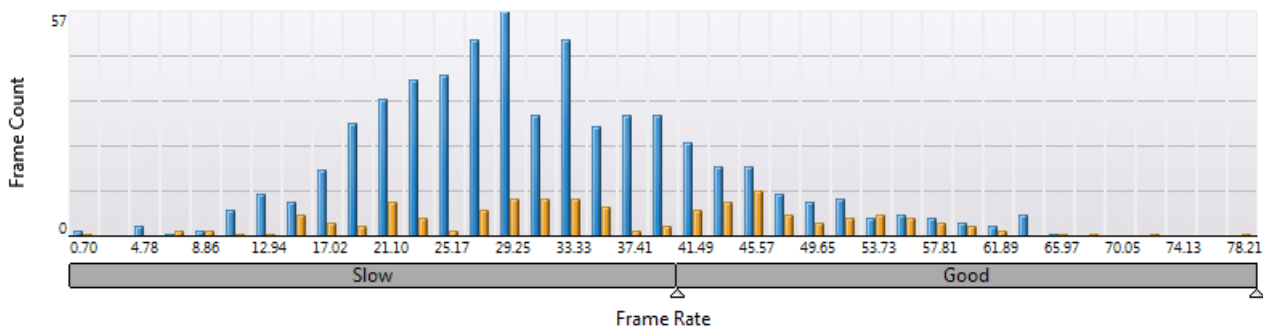
Application

```
voidalgorithm_1();  
voidalgorithm_2(int myid);  
doubleGetSeconds();  
DWORD WINAPI do_xform (void * lpmyid);  
bool checkResults();  
__itt_domain* pD = __itt_domain_create ("myDomain");
```

Region (Frame)

```
while( gRunning ) {  
    __itt_frame_begin_v3(pD, NULL);  
    ...  
    //Do Work  
    ...  
    __itt_frame_end_v3(pD, NULL);  
}
```

```
for (int k = 0; k < N; ++k) {  
    int ik = i*N + k;  
    int kj = k*N + j;  
    c2[ij] += a[ik]*b[kj];  
}
```



User API

Frame APIs reference

```
__itt_domain* __itt_domain_create(  
const __itt_char *name );
```

Create a domain with a domain name.

Since the domain is expected to be static over the application's execution time, 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.

```
void __itt_frame_begin_v3(const  
__itt_domain *domain, __itt_id *id);
```

Define the beginning of the frame instance.

A `__itt_frame_begin_v3` call must be paired with a `__itt_frame_end_v3` call.

Successive calls to `__itt_frame_begin_v3` with the same ID are ignored until a call to `__itt_frame_end_v3` with the same ID.

- domain is the domain for this frame instance.
- id is the instance ID for this frame instance, or NULL.

```
void __itt_frame_end_v3(const  
__itt_domain *domain, __itt_id *id);
```

Define the end of the frame instance.

A `__itt_frame_end_v3` call must be paired with a `__itt_frame_begin_v3` call. The first call to `__itt_frame_end_v3` with a given ID ends the frame. Successive calls with the same ID are ignored, as are calls that do not have a matching `__itt_frame_begin_v3` call.

- domain - The domain for this frame instance
- id - The instance ID for this frame instance, or NULL for the current instance.

User API

Frame APIs example

```
__itt_domain* pD = __itt_domain_create ("SimDomain");

while(gRunning) {
    __itt_frame_begin_v3(pD, NULL);

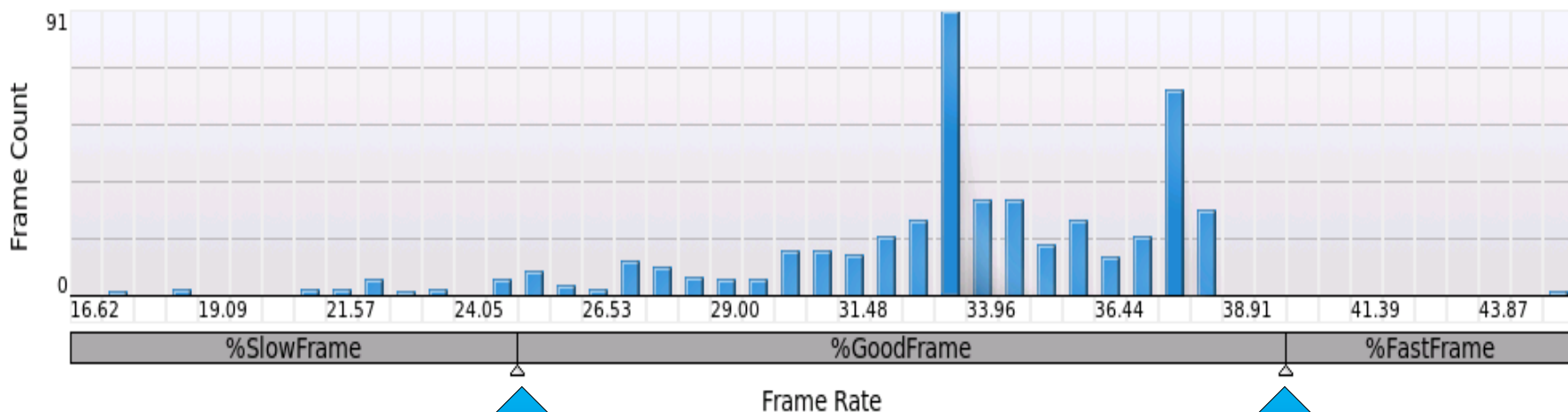
    start = clock();
    //Wait all threads before moving into the next frame
    WaitForMultipleObjects(FUNCTIONAL_DOMAINS, eSignal, TRUE,
INFINITE);
    stop = clock();
    //Give all threads the "go" signal
    for (int i = 0; i < FUNCTIONAL_DOMAINS; i++)
        SetEvent(bSignal[i]);
    if (frame % NETWORKCONNECTION_FREQ == 0) {
        //Start network thread
        SetEvent(bNetSignal);
    }
    __itt_frame_end_v3(pD, NULL);
}
```

Frame Analysis

Summary View / Frame Rate Chart

Frame Rate Chart

This histogram shows 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. Explore the data provided in the Bottom-up, Top-down Tree, and Timeline panes to identify code regions with the high/slow frame rate. Try to optimize your code to keep the frame rate constant (for example, from 30 to 60 frames per second).



Adjust the frame rate then changes

Frame Analysis

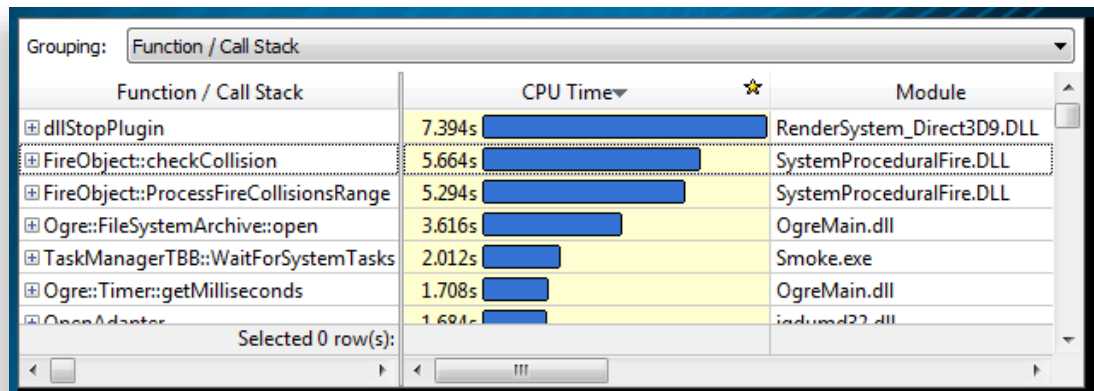
Find Slow Frames With One Click

(1) Regroup Data

Function - Call Stack
Module - Function - Call Stack
Source File - Function - Call Stack
Thread - Function - Call Stack
Function - Thread - Call Stack
OpenMP Region - Function - Call Stack
Task Type - Function - Call Stack
Frame Domain - Frame - Function - Call Stack
Frame Domain - Frame Type - Function - Call Stack

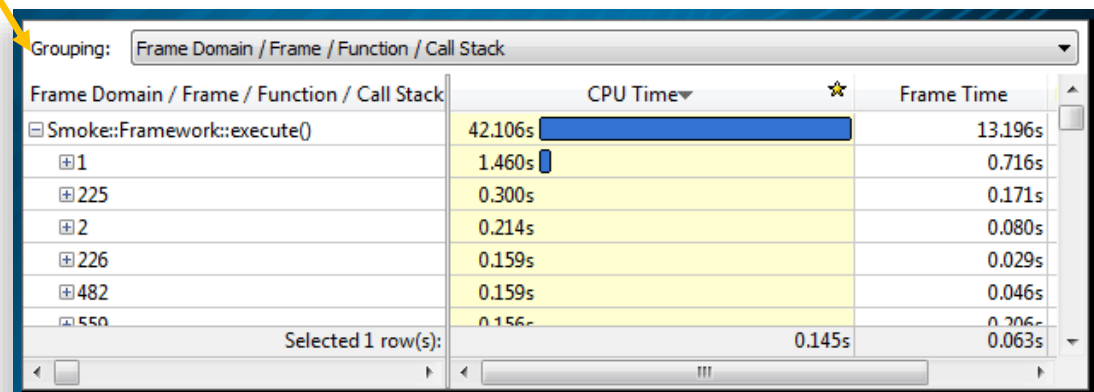
... (Partial list shown)

Before: List of Functions Taking Time



Function / Call Stack	CPU Time	Module
dllStopPlugin	7.394s	RenderSystem_Direct3D9.DLL
FireObject::checkCollision	5.664s	SystemProceduralFire.DLL
FireObject::ProcessFireCollisionsRange	5.294s	SystemProceduralFire.DLL
Ogre::FileSystemArchive::open	3.616s	OgreMain.dll
TaskManagerTBB::WaitForSystemTasks	2.012s	Smoke.exe
Ogre::Timer::getMilliseconds	1.708s	OgreMain.dll
OpenAdapter	1.694s	indum22.dll

After: List of Slow Frames

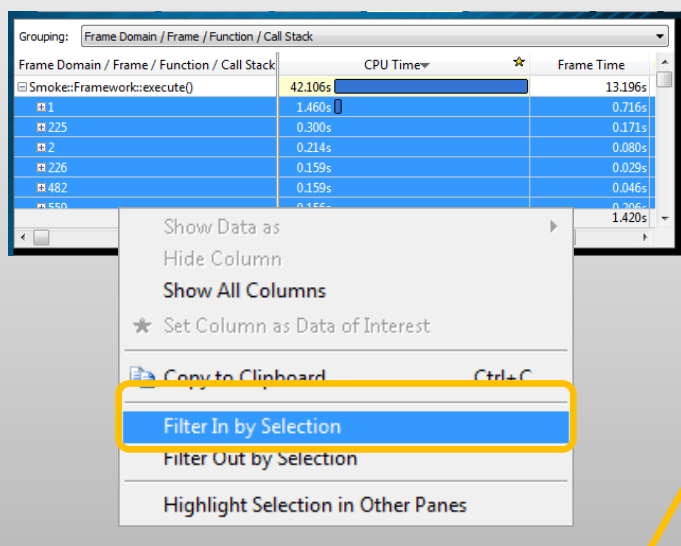


Frame Domain / Frame / Function / Call Stack	CPU Time	Frame Time
Smoke::Framework::execute()	42.106s	13.196s
1	1.460s	0.716s
225	0.300s	0.171s
2	0.214s	0.080s
226	0.159s	0.029s
482	0.159s	0.046s
550	0.156s	0.063s

Frame Analysis

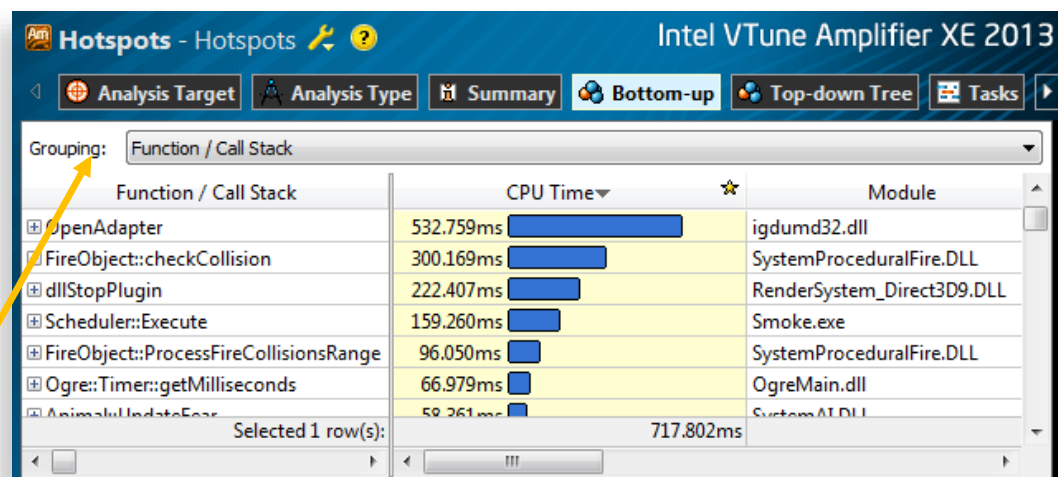
Find Slow functions in slow frames

(1) Only show slow frames



Result:

Functions taking a lot of time in slow frames



(2) Regroup: Show functions

Function - Call Stack

Module - Function - Call Stack

Source File - Function - Call Stack

Thread - Function - Call Stack

User API

Task APIs

- A **task** is a logical **unit of work** performed by a particular thread
- Tasks can be **nested**
- You can use task APIs to **assign tasks** to threads
- **One thread** executes **one task** at a given time
- Tasks may correspond to **functions**, **scopes**, or a **case block** in a switch statement

User API

Task APIs reference

Use This Primitive	To Do This
<code>void __itt_task_begin (const __itt_domain *domain, __itt_id taskid, __itt_id parentid, __itt_string_handle *name)</code>	Create a task instance on a thread. This becomes the current task instance for that thread. A call to <code>__itt_task_end()</code> on the same thread ends the current task instance.
<code>void __itt_task_end (const __itt_domain *domain)</code>	End a task instance on a thread.

Parameter	Description
<code>__itt_domain</code>	The domain of the task.
<code>__itt_id taskid</code>	This is a reserved parameter.
<code>__itt_id parentid</code>	This is a reserved parameter.
<code>__itt_string_handle</code>	The task string handle.

User API

Task APIs example

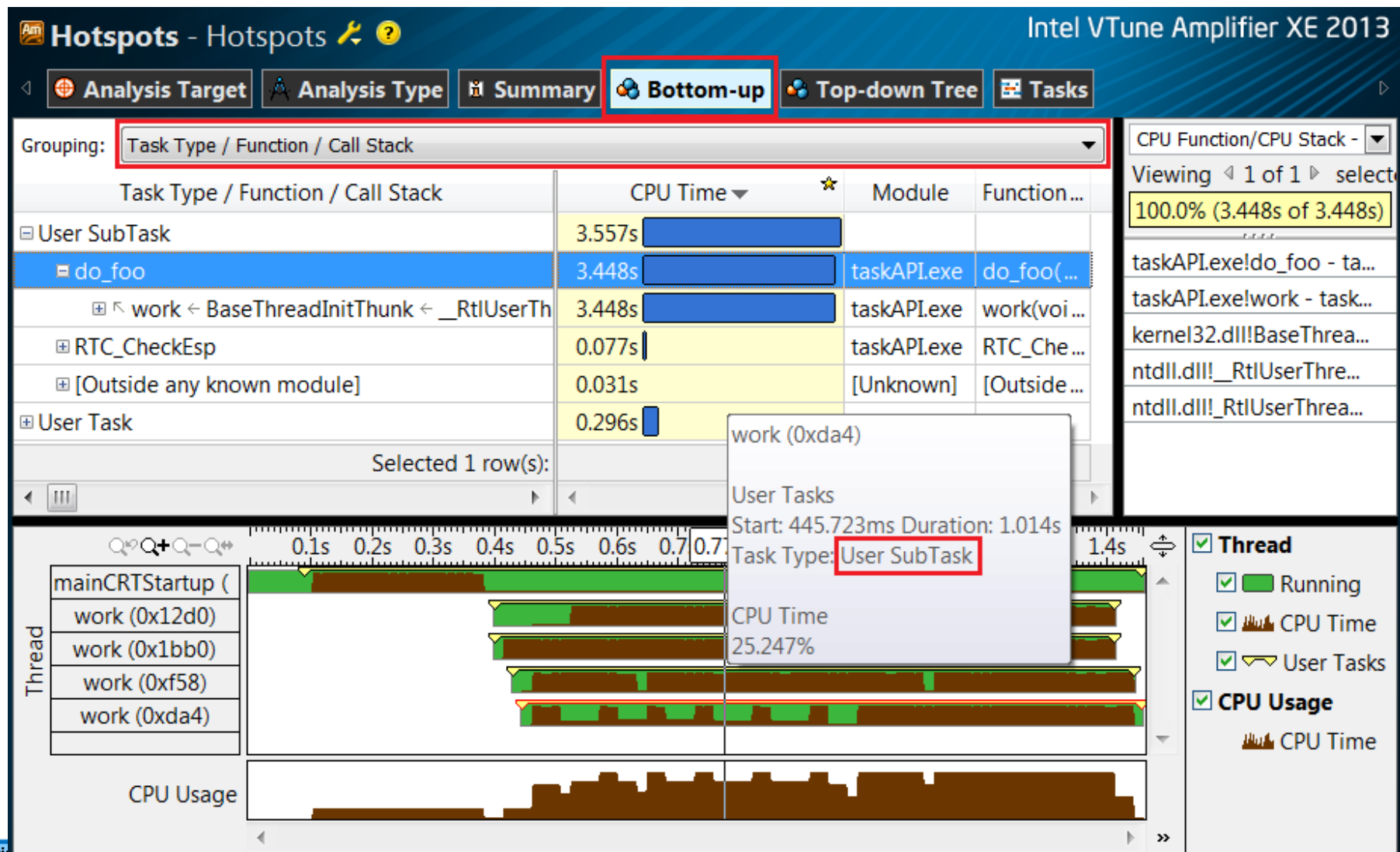
```
__itt_domain* domain = __itt_domain_create(L"Task Domain");
__itt_string_handle* UserTask = __itt_string_handle_create(L"UserTask");
__itt_string_handle* UserSubTask = __itt_string_handle_create(L"UserSubTask");

int main(int argc, char* argv[])
{
    ...
    __itt_task_begin (domain, __itt_null, __itt_null, UserTask);
    //create many threads to call work()
    __itt_task_end (domain);
    ...
}

work()
{
    __itt_task_begin (domain, __itt_null, __itt_null, UserSubTask);
    do_foo();
    __itt_task_end (domain);
    return 0;
}
```

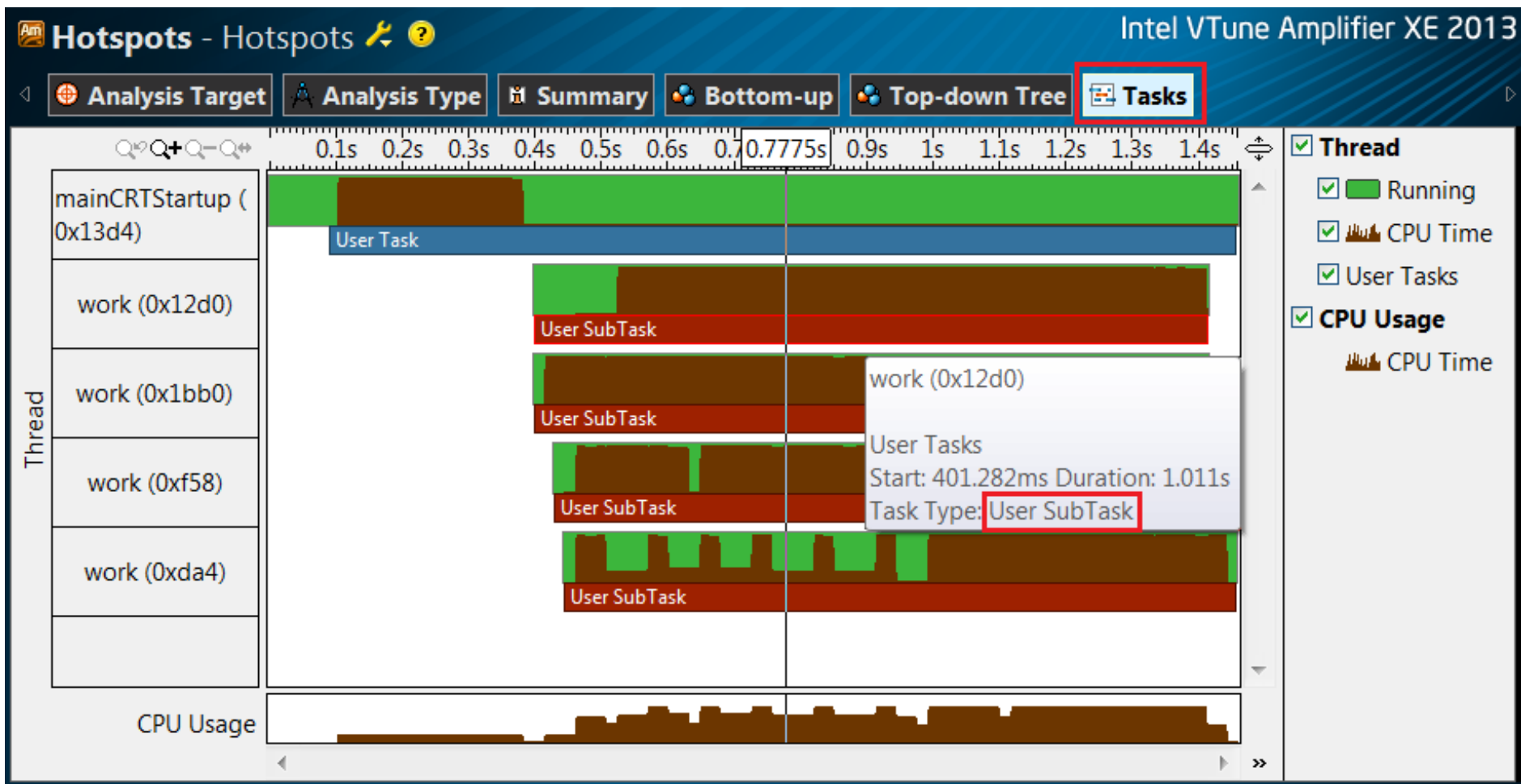
Using Task API

Hotspots analysis – Bottom-up view



Using Task API

Hotspots analysis – Task view



Command Line Interface

Gropof-like output

```
[levent@hlnasnb AXE_lab3]$ ampxe-cl -report gprof-cc -r /home/levent/examples/cern/labs/AXE_lab3/r003cc
Using result path ^/home/levent/examples/cern/labs/AXE_lab3/r003cc'
Executing actions 50 % Generating a report
```

Index	% Wait	Time:Total	Wait Time:Self	Children	Name	Index

[0]	99.88		190.104	190.104	G4RunManager::BeamOn	[23]
			190.104	0.0	ParRunManager::DoEventLoop	[0]
[1]	0.1		0.162	0.162	operator<<	[17]
			0.025	0.025	G4RunManagerKernel::G4RunManagerKernel	[11]
			0	0.001	RunAction::EndOfRunAction	[30]
			0.186	0.001	G4strstreambuf::sync	[1]
			0.001	0.001	G4MycoutDestination::ReceiveG4cout	[5]
[2]	83.08		0.033	158.141	func@0x416c28	[7]
			0.033	158.108	main	[2]
			0	158.108	G4_main	[18]
[3]	0.0		0.002	0.002	CLHEP::HepRandom::showEngineStatus	[22]
			0.002	0.0	CLHEP::RanecuEngine::showStatus	[3]
[4]	0.0		0.001	0.001	G4_main	[18]
			0.001	0.0	G4MycoutDestination::G4MycoutDestination	[4]
[5]	0.0		0.001	0.001	G4strstreambuf::sync	[1]
			0.001	0.0	G4MycoutDestination::ReceiveG4cout	[5]
[6]	0.0		0	0	G4UImanager::ExecuteMacroFile <cycle 1>	[28]
			0.0	0.0	G4UIbatch::G4UIbatch	[6]
[7]	83.08		0.0	158.141	func@0x416c28	[7]
			0.033	158.141	main	[2]
[8]	99.88		0	190.107	G4_main	[18]
			0.0	190.107	<cycle 1 as a whole>	[8]

Command Line Interface

CSV output

```
$> amplxe-cl -report hotspots -csv-delimiter=comma -format=csv -  
report-out=testing111 -r r003cc
```

```
Function,Module,CPU Time,Idle:CPU Time,Poor:CPU Time,Ok:CPU  
Time,Ideal:CPU Time,Over:CPU Time  
CLHEP::RanecuEngine::flat,test40,50.751,0,0.050,0.081,0.080,50.541  
G4UniversalFluctuation::SampleFluctuations,test40,32.730,0,0.030,0.  
070,0.010,32.620  
sqrt,test40,19.060,0,0.010,0.070,0.030,18.951  
G4Track::GetVelocity,test40,15.330,0,0.030,0.030,0.040,15.230  
G4VoxelNavigation::LevelLocate,test40,14.460,0,0.020,0.010,0.040,14  
.390  
G4Step::UpdateTrack,test40,14.090,0,0,0.030,0.020,14.040  
G4NavigationLevelRep::G4NavigationLevelRep,test40,13.721,0,0.030,0.  
020,0.040,13.631  
exp,test40,13.438,0,0.038,0.010,0.060,13.330  
log,test40,13.340,0,0.180,0.020,0.110,13.030  
G4PhysicsVector::GetValue,test40,11.970,0,0.020,0.020,0.050,11.880
```