Performance Tuning Methodology (Intel® VTune™ Amplifier XE)

Dr.-Ing. Michael Klemm
Software and Services Group
Intel Corporation
(michael.klemm@intel.com)

Legal Disclaimer & Optimization Notice

Optimization Notice

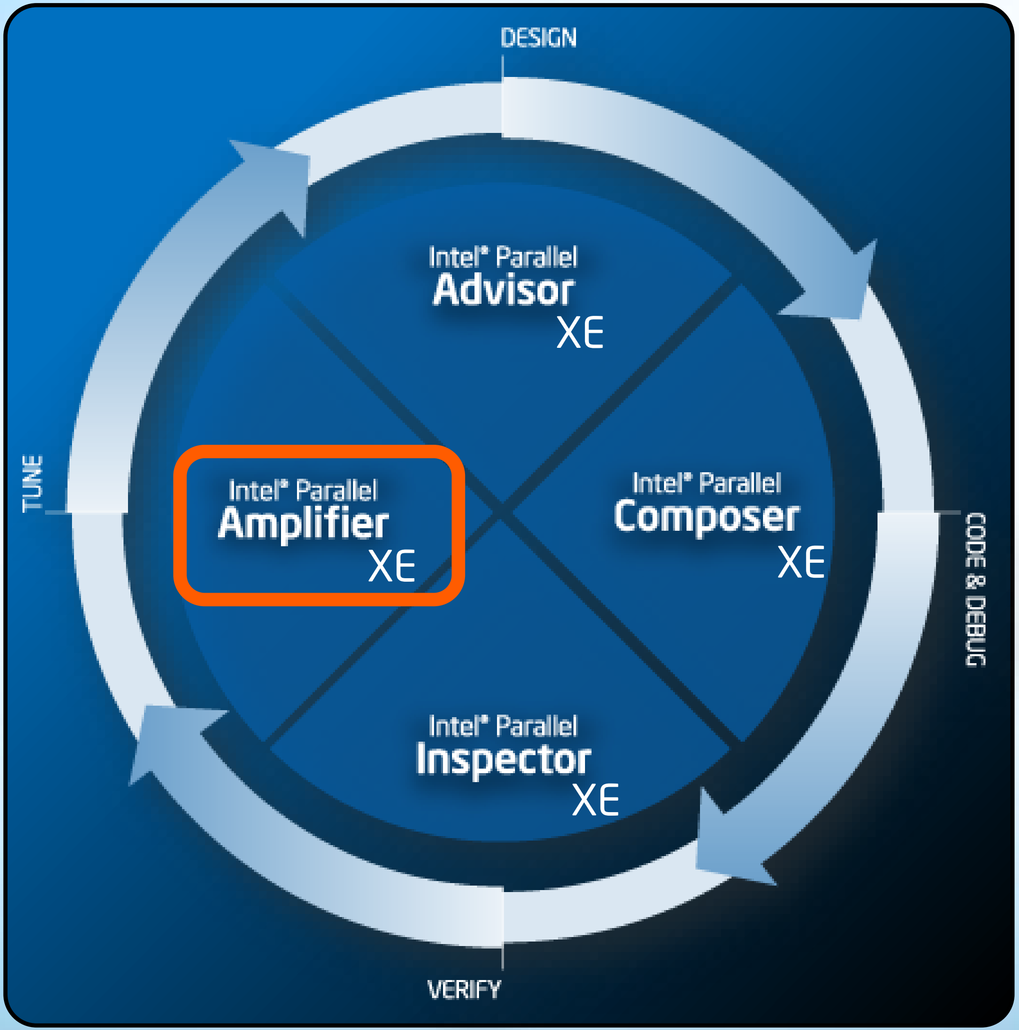
Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE: User Interface
- Fundamental Analysis: Hotspots
- Finding Issues in Parallel Applications
- Using the Performance Monitoring Unit

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE: User Interface
- Fundamental Analysis: Hotspots
- Finding Issues in Parallel Applications
- Using the Performance Monitoring Unit

Intel® Parallel Studio XE



Holistic toolset for the parallel software development lifecycle

- DESIGN
- CODE & DEBUG
- VERIFY
- TUNE

Intel® Cluster Studio XE adds:

- Intel® MPI
- Intel® Trace Analyzer and Collector

The Software Optimization Process

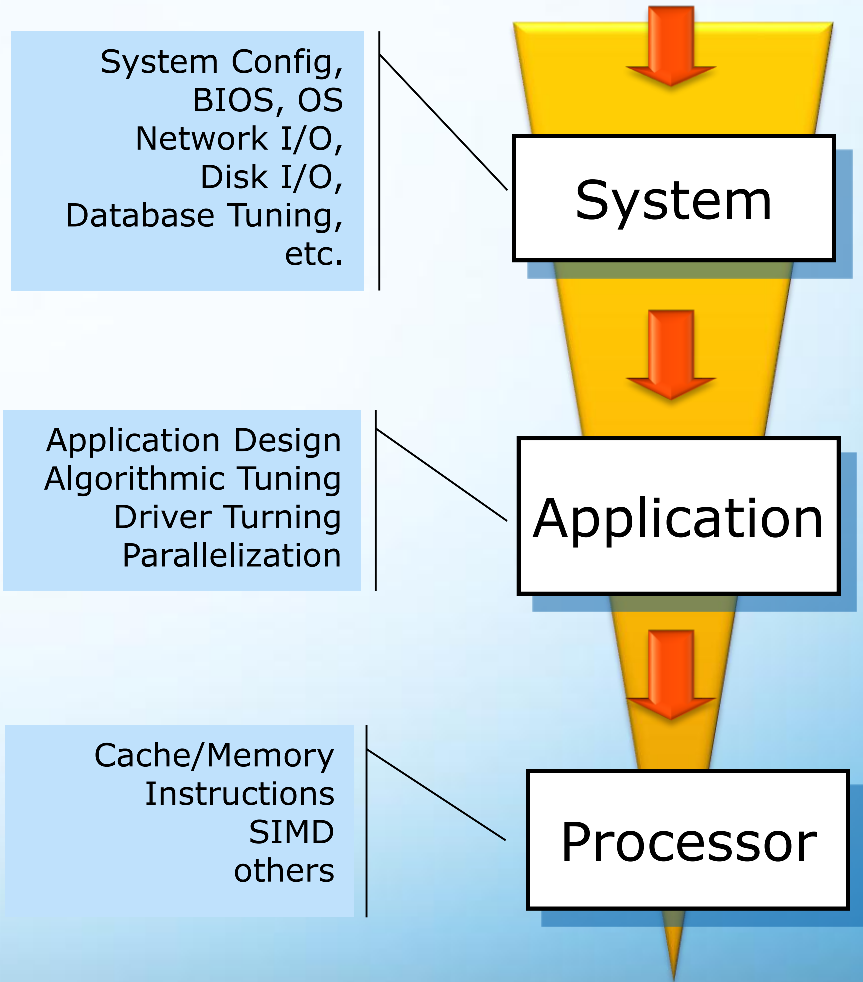
- The process of improving the software by eliminating bottlenecks so that it operates more efficiently on a given hardware and uses resources optimally
  - Identifying bottlenecks in the target application and eliminating them appropriately is the key to an efficient optimization
- There are many optimization methodologies, which help developers answer the questions of
  - Why to optimize?
  - What to optimize?
  - To what to optimize?
  - How to optimize?

These methods aid developers to reach their performance requirements.

Performance Analysis Methodology

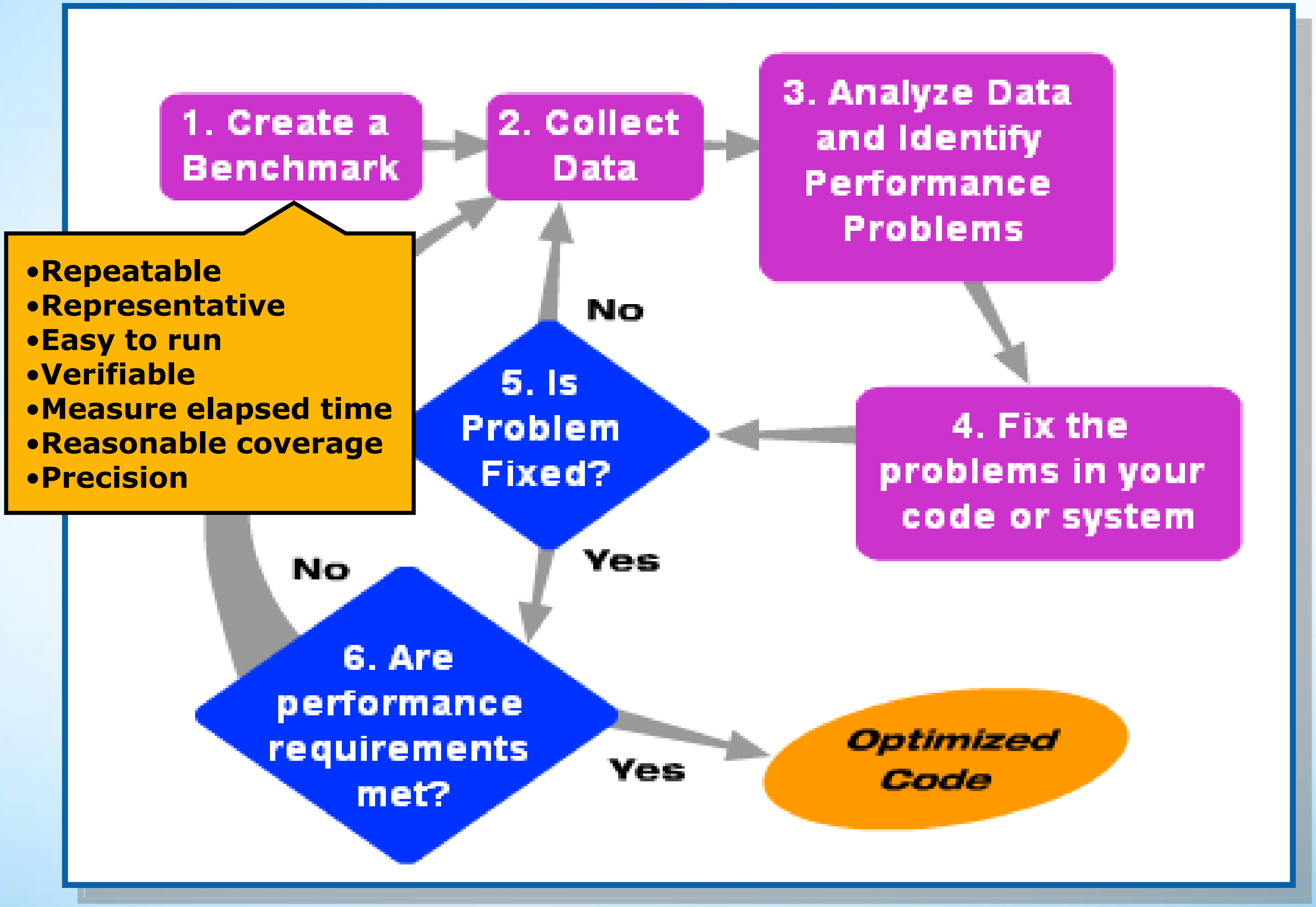
Optimization: A Top-down Approach

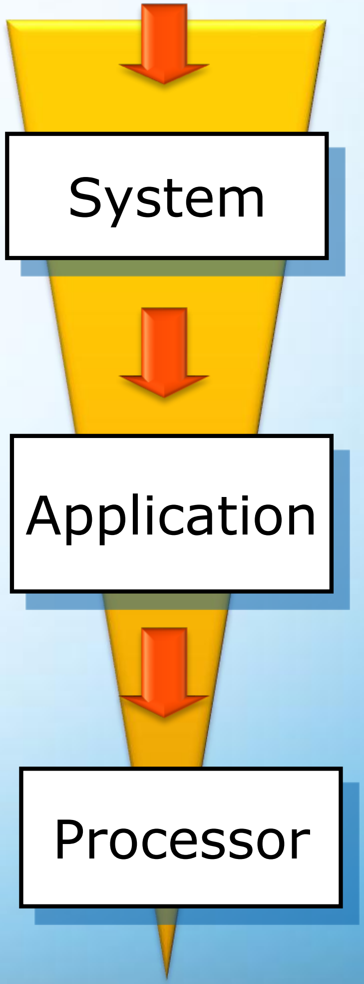
- Use top down approach
- Understand application and system characteristics
  - Use appropriate tools at each level



Performance Analysis Methodology

Optimization: A Top-down Approach





When to Stop

- Is architecture at maximum efficiency?
  - What this means: calculating theoretical maximum.
  - Know about best values for CPI or IPC.
  - Know the maximum FLOPS for the data type used.
- Is the performance requirement fulfilled?
  - What are the performance requirements?
  - Incrementally complete optimizations until done.
  - Key question: Are you “happy” with it?

CPI: Cycles per Instructions
IPC: Instructions per Cycle
FLOPS: Floating-Point Oper. Per Sec.

Questions to Ask Yourself

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

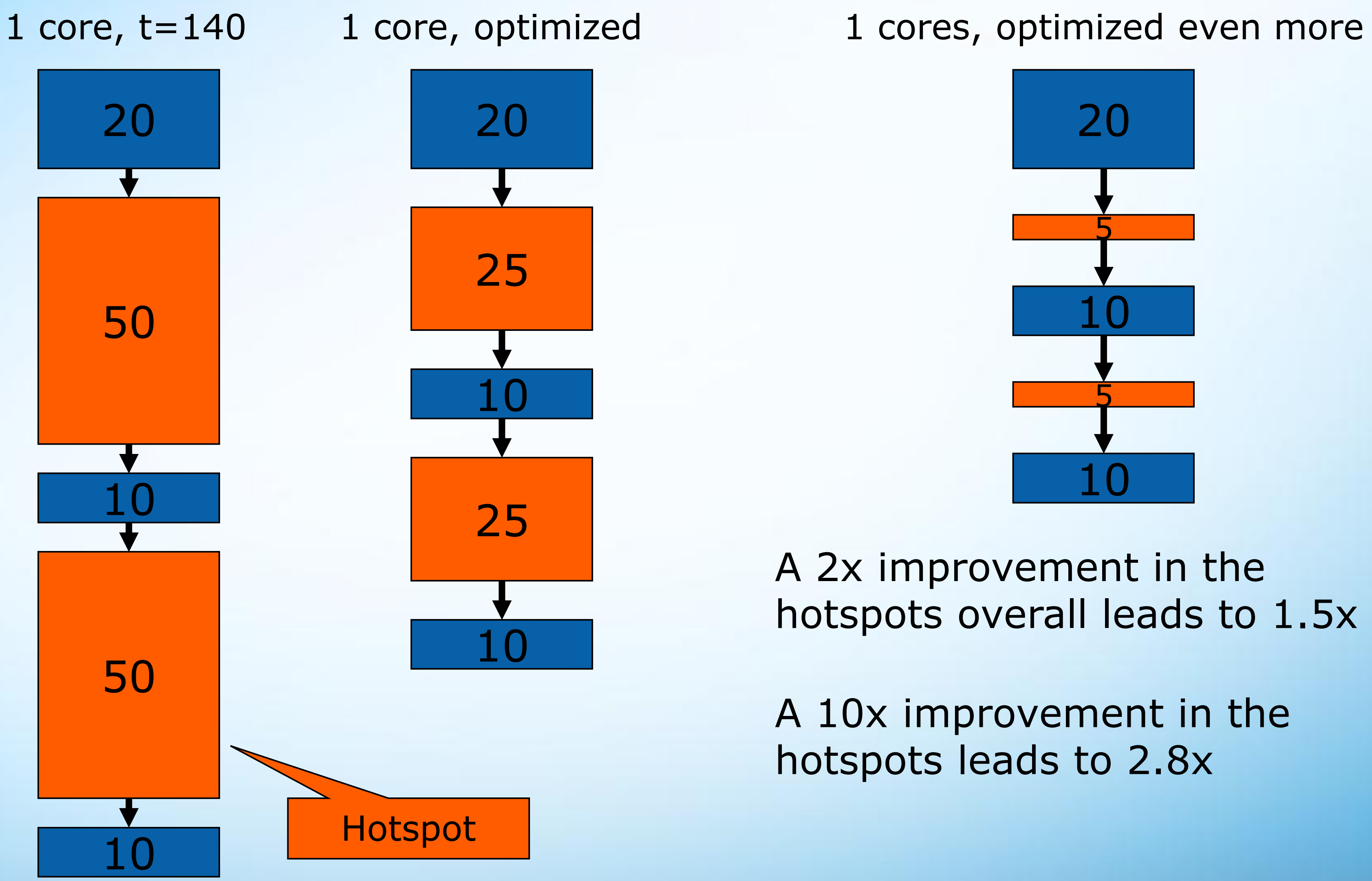
— Donald Knuth

Quality code is:

- Portable
- Readable
- Maintainable
- Reliable

Intelligently sacrifice quality for performance

Amdahl's Law Slightly Reinterpreted



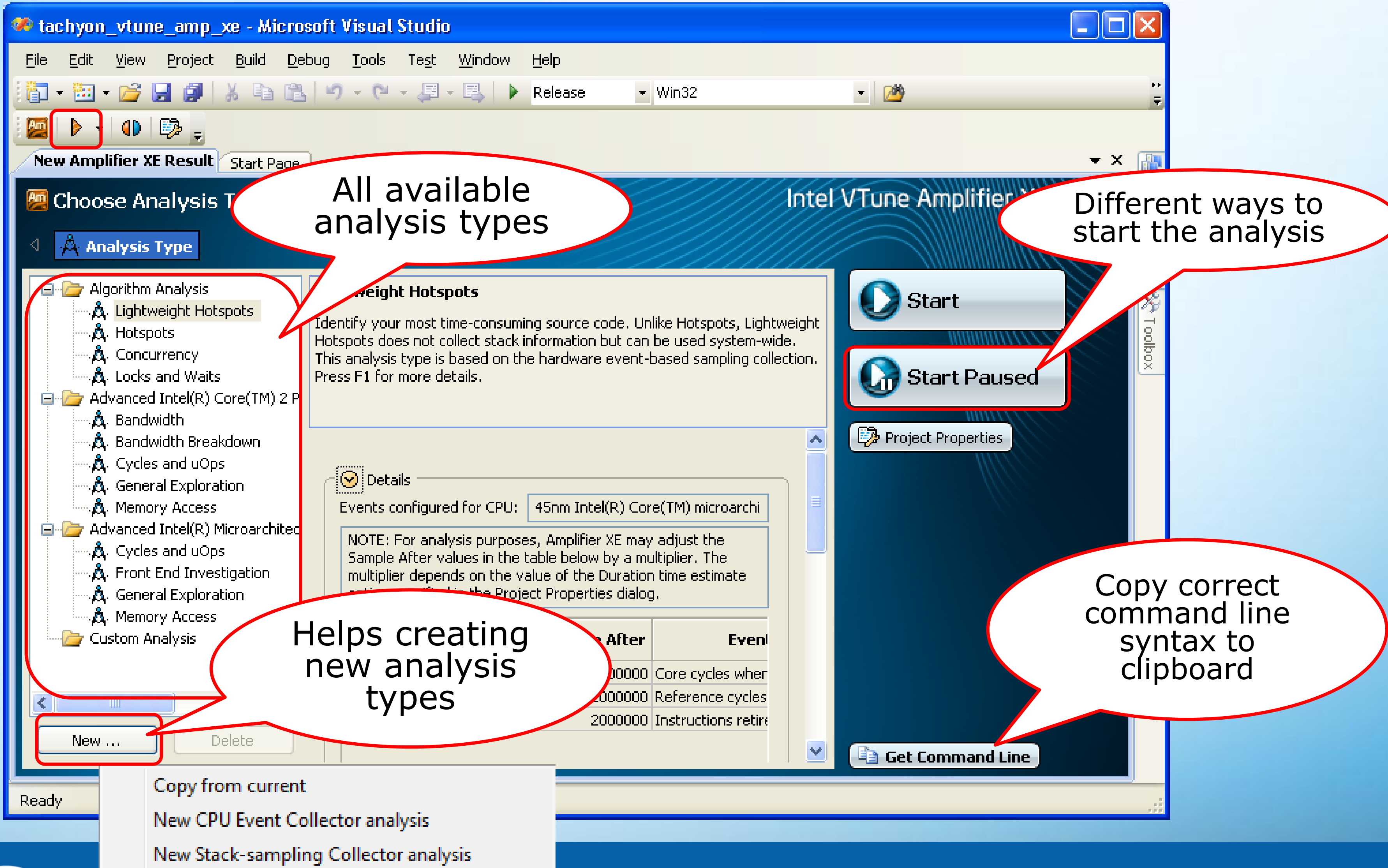
A 2x improvement in the hotspots overall leads to 1.5x

A 10x improvement in the hotspots leads to 2.8x

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE: User Interface
- Fundamental Analysis: Hotspots
- Finding Issues in Parallel Applications
- Using the Performance Monitoring Unit

Selecting type of data collection



VTune™ Amplifier XE

GUI Layout

Adjust Data Grouping

- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack

… (Partial list shown)

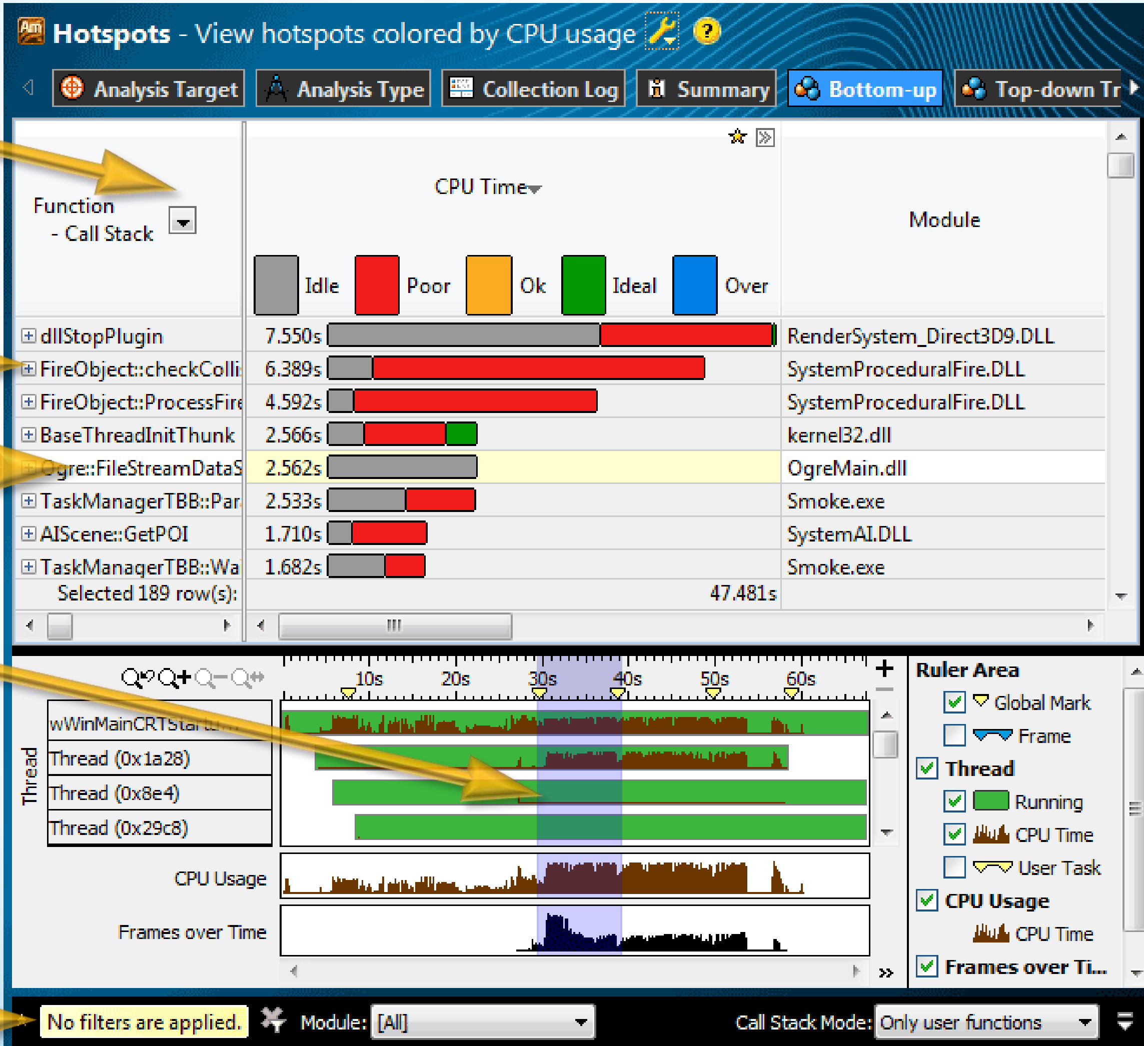
Click [+] for Call Stack

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

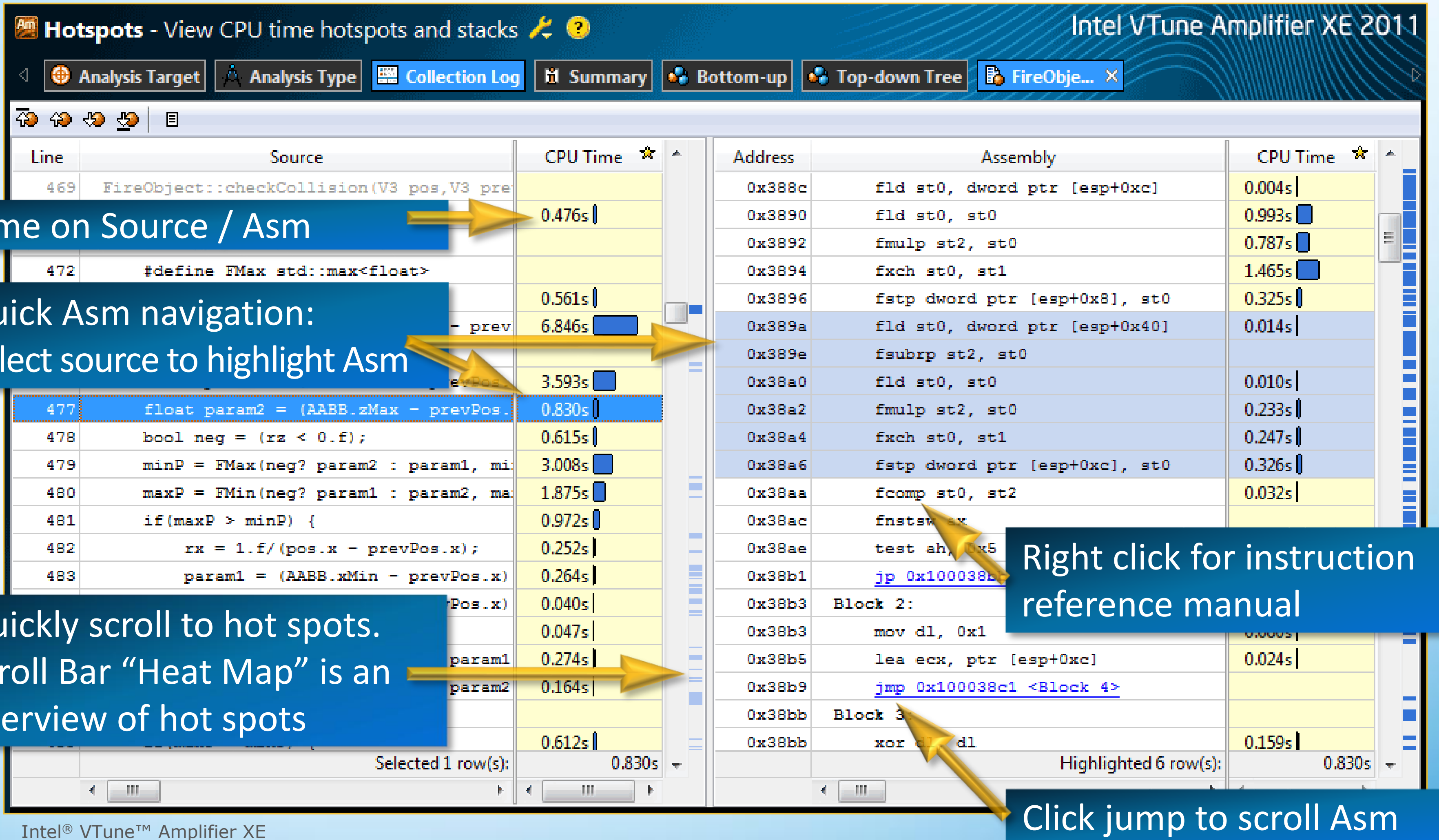
- Zoom In And Filter On Selection
- Filter In by Selection
- Remove All Filters

Filter by Module & Other Controls



VTune™ Amplifier XE

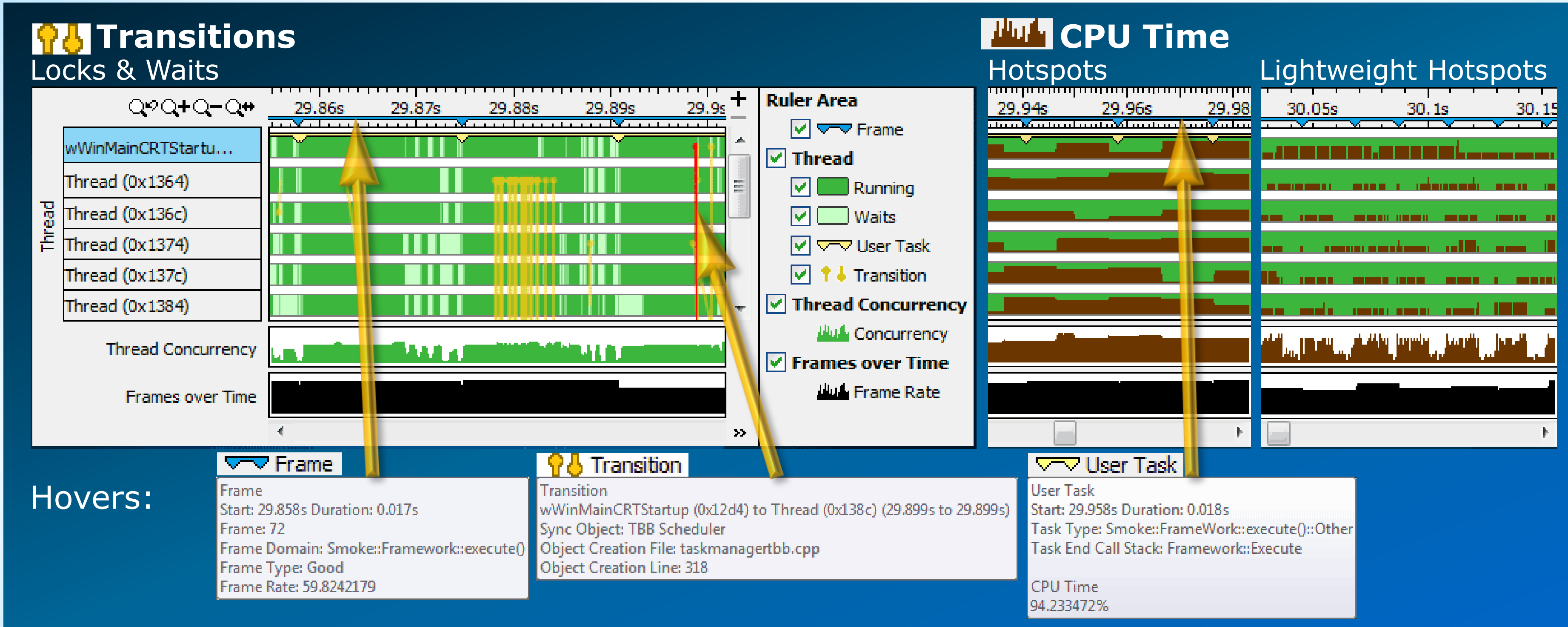
GUI Layout



Intel® VTune™ Amplifier XE

VTune™ Amplifier XE

GUI Layout



- Optional: Use API to mark frames and user tasks
- Optional: Add a mark during collection [Mark Timeline]

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE: User Interface
- Fundamental Analysis: Hotspots
- Finding Issues in Parallel Applications
- Using the Performance Monitoring Unit

Readying Your Application for Intel VTune Amplifier XE

- You should run Amplifier XE on a "Released/Optimized" build.

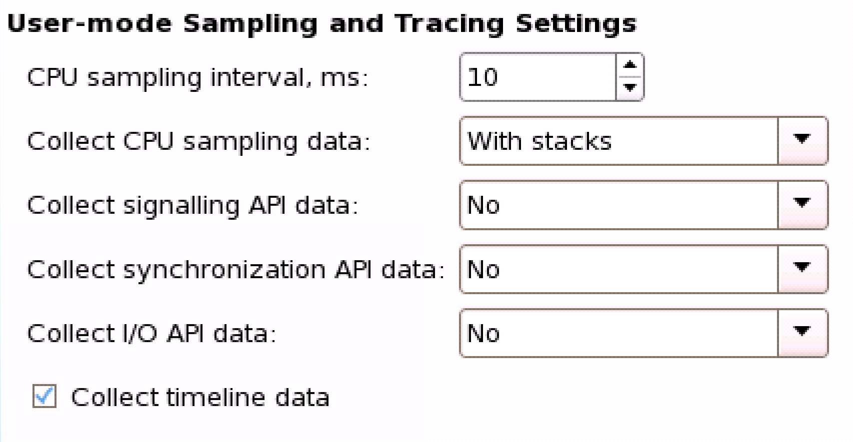
- Symbols Allow you to view the source (not just the assembly)
  - Windows: /Zi
  - Linux: –g

- Intel Threading Runtimes need instrumented runtimes
  - TBB: Define TBB_USE_THREADING_TOOLS
  - OpenMP: Use Intel Dynamic Version of OpenMP

- Call Stack Mode – Requires use of the dynamic version of the C Runtime library to properly attribute System Calls
  - Windows use:/MD(d)
  - Linux do not use: -static

Analysis Types
Hotspots

- For each sample, capture execution context, time passed since previous sample and thread CPU time
- Allows time spent in system calls to be attributed to user functions making the call
- Provides additional knobs:
  - The defaults for Hotspot analysis are configurable and can be done so by creating a custom analysis type inherited from Hotspots



Analysis Types
Lightweight Hotspots

- Similar to Hotspot Analysis
  - Sampling is performed with the SEP collector
  - Driver is required
- Stack walking is not performed
  - Only hotspots are reported
- Samples are taken more frequently, but may have less accurate timing information
- Analysis may be performed for a single application or for the entire system

Lightweight Hotspots vs. Hotspots

Smoke

Function	CPU_CLK_UN... CORE
FireObject::checkCollision	13.3%
FireObject::ProcessFireCollisionsRange	7.6%
D3DXSHProjectCubeMap	3.7%
tbb::internal::task_scheduler_observer_v3::obser	3.1%
D3DDisassemble10Effect	2.9%
std::vector<class FireObject *,class std::allocator	2.8%
Ogre::TextureUnitState::getDesiredFormat	2.8%
RtlInitUnicodeString	2.3%
modf	2.3%
ParticleEmitter::ParticleSystem::update	1.5%
Animal::UpdateFear	1.5%
[fmodex.dll]	1.4%
[nvd3dum.dll]	1.3%
LdrUnlockLoaderLock	1.3%
hkpMoppObbVirtualMachine::queryAabbOnTre	1.1%
[nvlddmkm.sys]	1.1%
ParticleEmitter::FirePatch::initParticle	1.1%
KeSynchronizeExecution	1.1%

Function - Call Stack	CPU Time (Idle / Poor / Ok)
FireObject::checkCollision	13.1%
dllStopPlugin	10.7%
FireObject::ProcessFireCollisionsRa	9.4%
BaseThreadInitThunk	5.6%
TaskManagerTBB::WaitForSystemT	5.2%
Animal::UpdateFear	3.1%
Ogre::FileStreamDataStream::read	3.0%
Ogre::TextureUnitState::getDesiredF	2.5%
AIScene::GetPOI	2.4%
ParticleEmitter::FirePatch::initPartic	1.7%
ParticleEmitter::ParticleSystem::upd	1.5%
TaskManagerTBB::ParallelFor	1.4%
Ogre::FileSystemArchive::open	1.2%
hkpMoppObbVirtualMachine::quer	1.2%
Ogre::ZipArchive::findFileInfo	1.2%
hkSolveConstraintsByTheSteps	1.1%
Instrumentation::Instrumentation	1.1%
FireObject::EmitterCollisionCheck	1.1%

- Mostly correlates, however the default attribution of system time in Hotspots is to the user function making the system call

Call Stack Mode: Only user functions

Lightweight Hotspots vs. Hotspots

Smoke

Function	CPU_CLK_UN... CORE
FireObject::checkCollision	13.3%
FireObject::ProcessFireCollisionsRange	7.6%
D3DXSHProjectCubeMap	3.7%
tbb::internal::task_scheduler_observer_v3::obser	3.1%
D3DDisassemble10Effect	2.9%
std::vector<class FireObject *,class std::allocator	2.8%
Ogre::TextureUnitState::getDesiredFormat	2.8%
RtlInitUnicodeString	2.3%
modf	2.3%
ParticleEmitter::ParticleSystem::update	1.5%
Animal::UpdateFear	1.5%
[fmodex.dll]	1.4%
[nvd3dum.dll]	1.3%
LdrUnlockLoaderLock	1.3%
hkpMoppObbVirtualMachine::queryAabbOnTre	1.1%
[nvlddmkm.sys]	1.1%
ParticleEmitter::FirePatch::initParticle	1.1%
KeSynchronizeExecution	1.1%

Function - Call Stack	CPU Time (Idle / Poor / Ok)
FireObject::checkCollision	13.1%
FireObject::ProcessFireCollisionsRange	6.9%
D3DXSHProjectCubeMap	3.6%
NtYieldExecution	3.1%
NtReadFile	3.1%
D3DDisassemble10Effect	3.0%
tbb::internal::task_scheduler_observer_v3	2.5%
Ogre::TextureUnitState::getDesiredForm	2.5%
std::vector<class FireObject *,class std::a	2.5%
WaitForSingleObject	2.5%
RtlInitUnicodeString	2.4%
modf	2.4%
tbb::captured_exception::~captured_exc	1.9%
Animal::UpdateFear	1.4%
NtWaitForMultipleObjects	1.4%
ParticleEmitter::ParticleSystem::update	1.4%
NtCreateFile	1.3%
hkpMoppObbVirtualMachine::queryAab	1.2%

- Setting Call Stack Mode: User/system functions gives you better correlation to the hotspot report by Lightweight Hotspot analysis type

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE: User Interface
- Fundamental Analysis: Hotspots
- Finding Issues in Parallel Applications
- Using the Performance Monitoring Unit

Issues in Parallel Applications

- Load imbalance
  - Work distribution is not optimal
  - Some threads are heavily loaded, while others idle
  - Slowest thread determines total speed-up
- Locking issues
  - Locks prohibit threads to concurrently enter code regions
  - Effectively serialize execution
- Parallelization overhead
  - With large no. of threads, data partition get smaller
  - Overhead might get significant (e.g. OpenMP startup time)

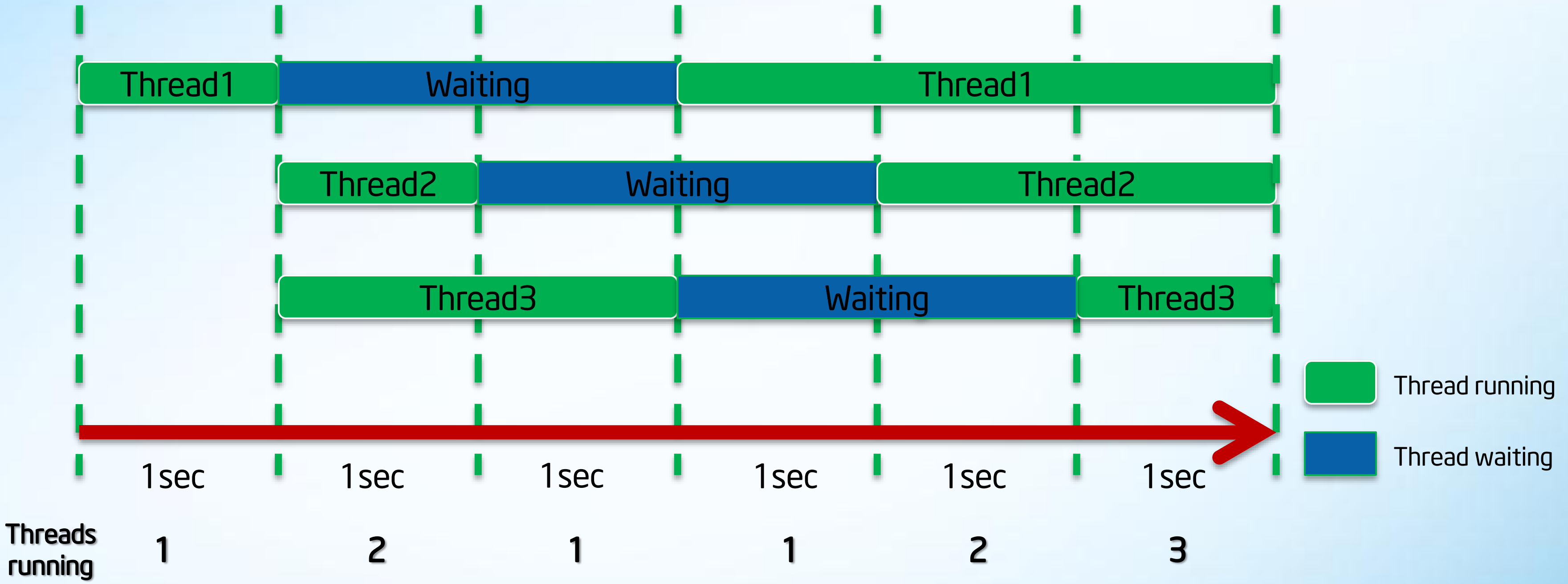
Issues in Parallel Applications

- Load imbalance
  - Work distribution is not optimal
  - Some threads are heavily loaded, while others idle
  - Slowest thread determines total speed-up

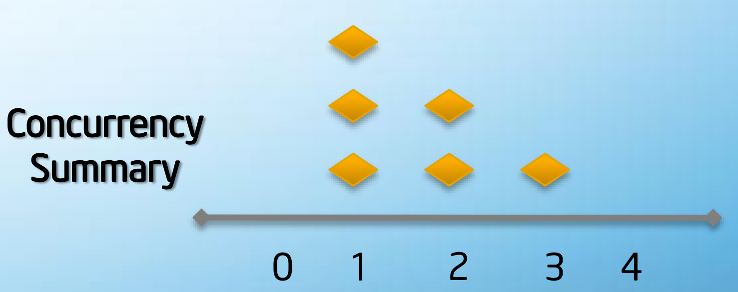
- Locking issues
  - Locks prohibit threads to concurrently enter code regions
  - Effectively serialize execution

- Parallelization overhead
  - With large no. of threads, data partition get smaller
  - Overhead might get significant (e.g. OpenMP startup time)

Threading Analysis Terminology

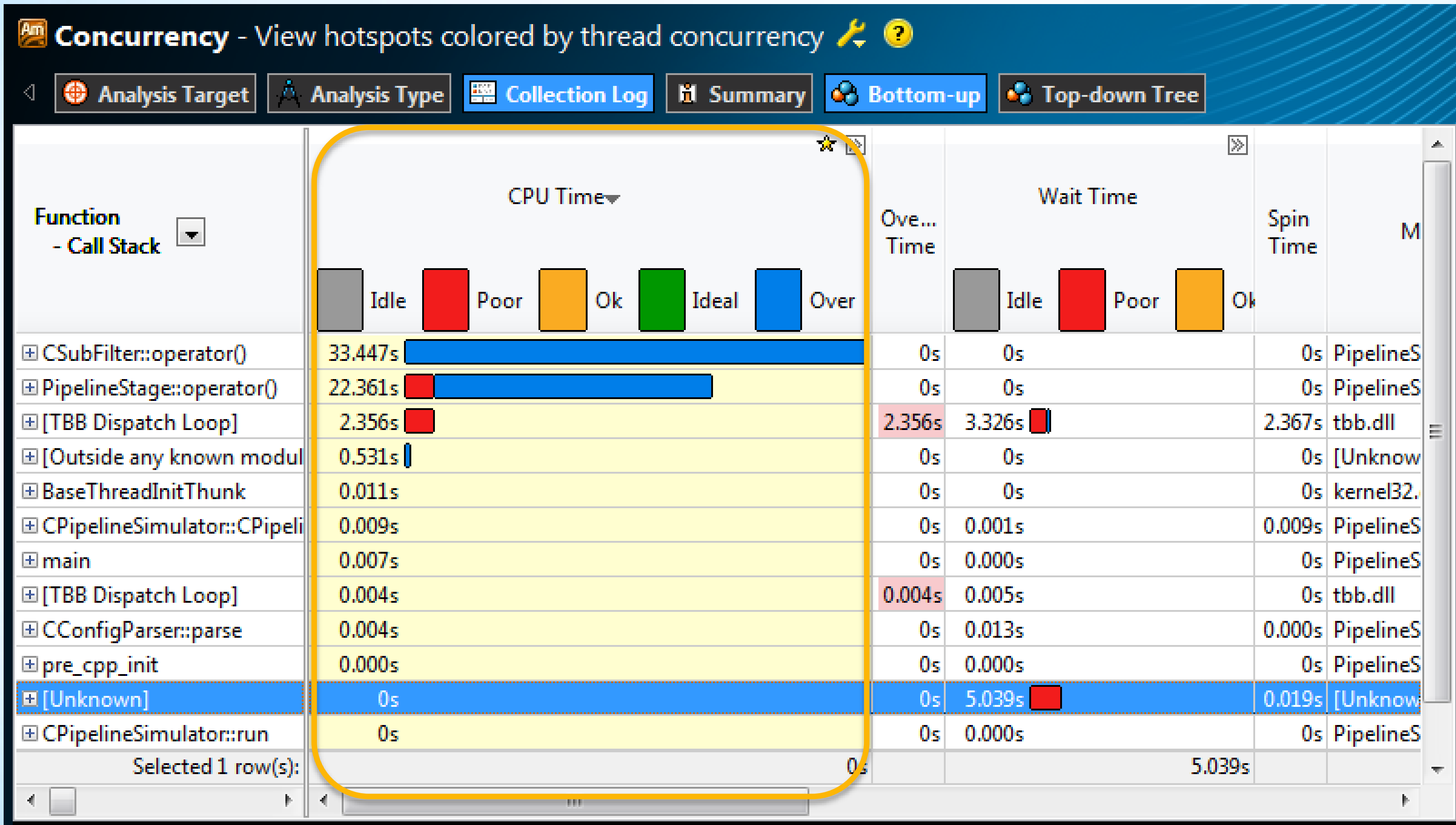


- **Elapsed Time**: 6 seconds
- **CPU Time**: T1 (4s) + T2 (3s) + T3 (3s) = 10 seconds
- **Wait Time**: T1(2s) + T2(2s) + T3 (2s) = 6 seconds



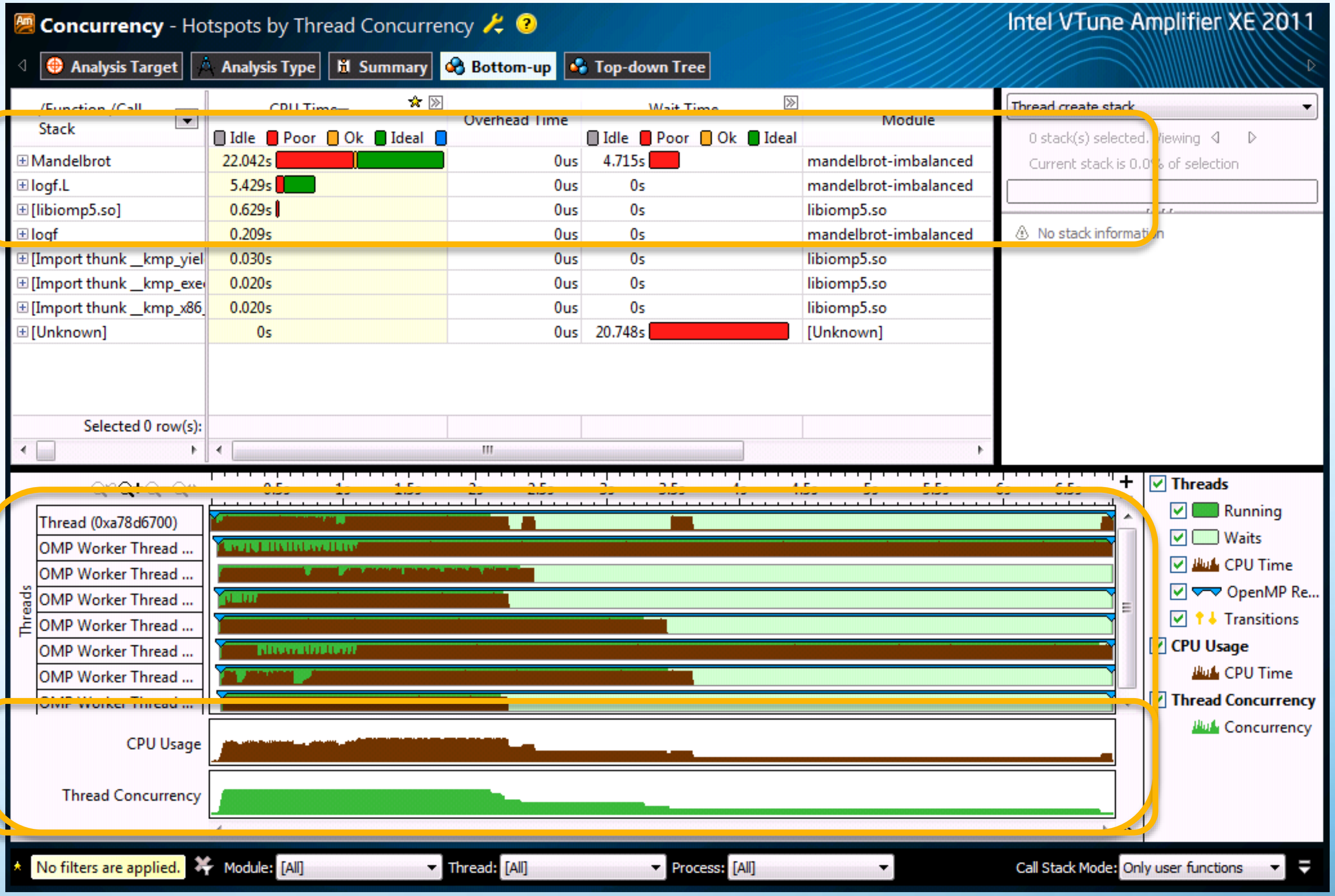
Analysis Types

Concurrency

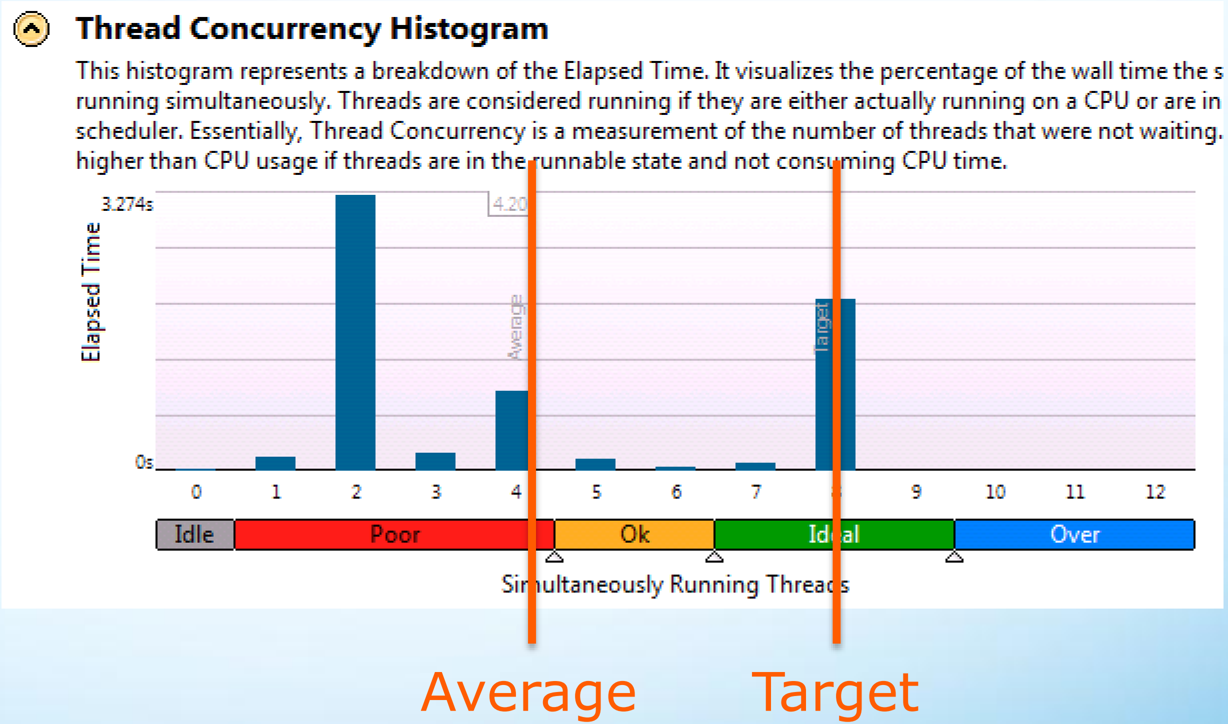


Hotspots Analysis vs. Concurrency Analysis

- Hotspot Analysis and Concurrency Analysis are similar:



Thread Concurrency Histogram



Issues in Parallel Applications

- Load imbalance
  - Work distribution is not optimal
  - Some threads are heavily loaded, while others idle
  - Slowest thread determines total speed-up
- Locking issues
  - Locks prohibit threads to concurrently enter code regions
  - Effectively serialize execution
- Parallelization overhead
  - With large no. of threads, data partition get smaller
  - Overhead might get significant (e.g. OpenMP startup time)

Analysis Types

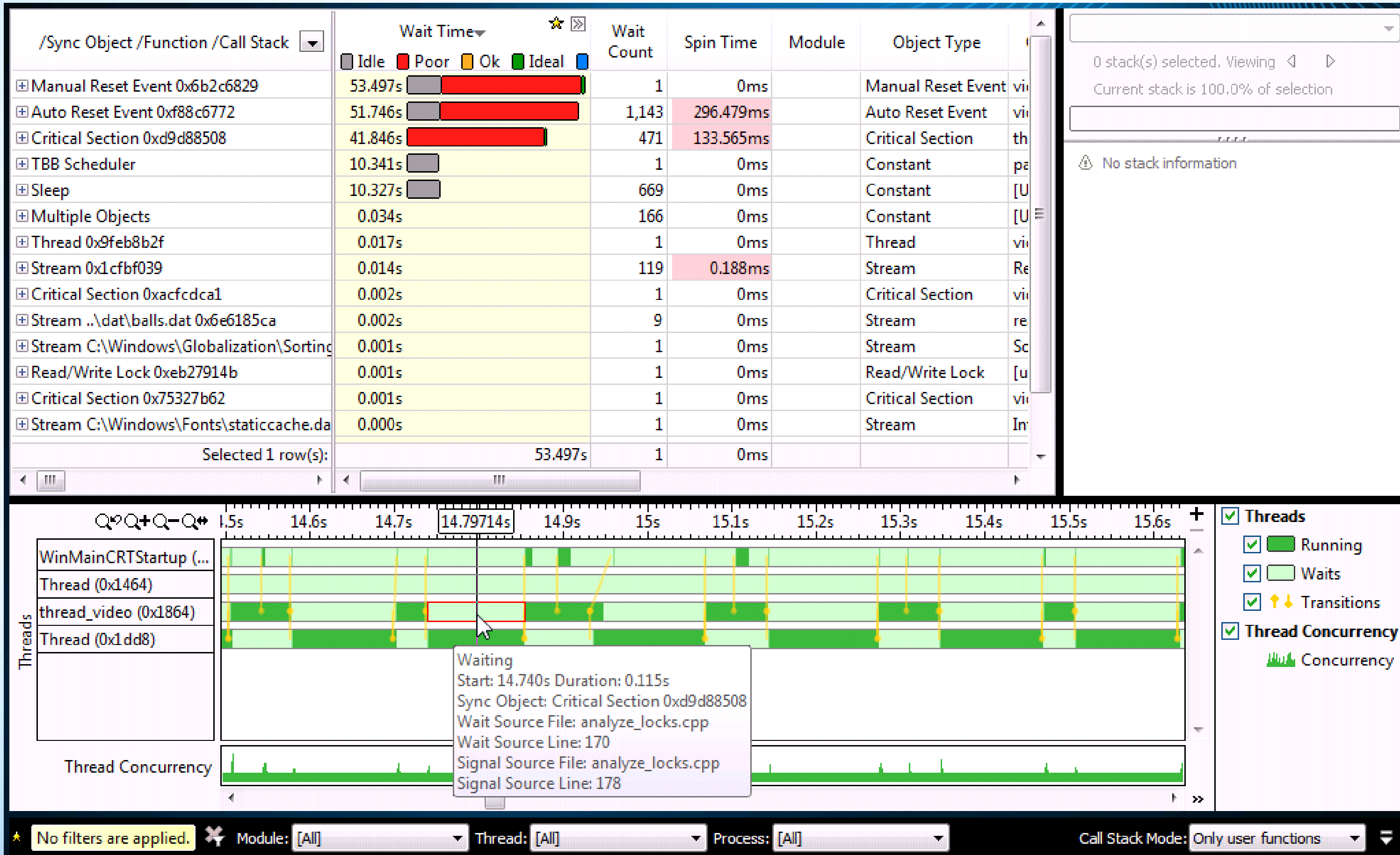
Lock and Waits

Idle | Poor | Ok

Sync Object - Function - Call Stack	Wait Time	Wait Count	Spin Time	Module	Object Type	Object Creatic
⊞ Multiple Objects	112.517s	5,325	0s	[Unknown]	Constant	[Unknown]
⊞ Manual Reset Event 0xae37	109.238s	41	0s	[Unknown]	Manual Reset Event	dllStopPlugin
⊞ Manual Reset Event 0xfa6d	74.068s	26	0s	[Unknown]	Manual Reset Event	LdrGetProcedureAddressEx
⊞ Thread Pool	57.628s	235	0s	[Unknown]	Constant	[Unknown]
⊞ Sleep	57.371s	5,234	0.193s	[Unknown]	Constant	[Unknown]
⊞ Unknown 0x991c9877	56.974s	6,337	0s	[Unknown]	Unknown	LdrGetProcedureAddressEx
⊞ TBB Scheduler	41.457s	2,200	11.301s	[Unknown]	Constant	TaskManagerTBB::Init
⊞ [Unknown]	17.061s	865	0s	[Unknown]	[Unknown]	[Unknown]
⊞ Stream ../media/graphics/	0.457s	183	0.057s	[Unknown]	Stream	Ogre::FileSystemArchive::open
⊞ Stream ../media/sounds/h	0.440s	171	0.063s	[Unknown]	Stream	Framework::GDFParser::EndEler
⊞ Stream Ogre.log 0x501382c	0.397s	193	0.059s	[Unknown]	Stream	Ogre::Log::Log
⊞ Stream Smoke.gdf 0xf2b92	0.386s	11	0.006s	[Unknown]	Stream	PlatformManager::FileSystem::S
⊞ Stream ../media/sounds/M	0.306s	119	0.037s	[Unknown]	Stream	Framework::GDFParser::EndEler
⊞ Stream ..\media\physics\D	0.247s	5	0.011s	[Unknown]	Stream	hkStdioStreamReader::hkStdioS
⊞ Stream ../media/graphics/	0.136s	41	0.022s	[Unknown]	Stream	Ogre::FileSystemArchive::open
⊞ Stream ../media/sounds/h	0.134s	13	0.018s	[Unknown]	Stream	TaskManagerTBB::ParallelFor
Selected 1 row(s):	112.517s	5,325				

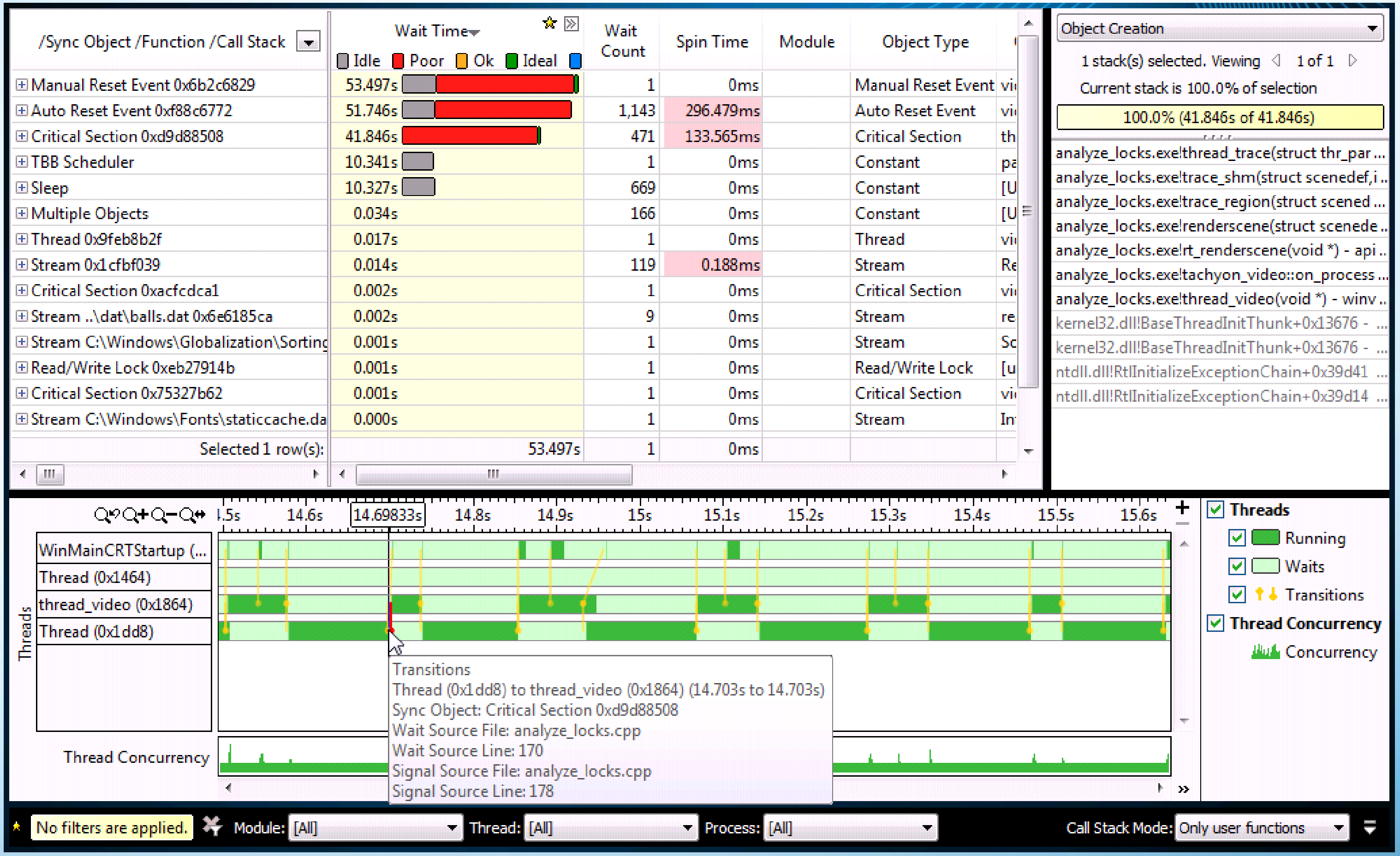
Timeline Visualizes Thread Behavior

- Retrieve additional information about waiting threads



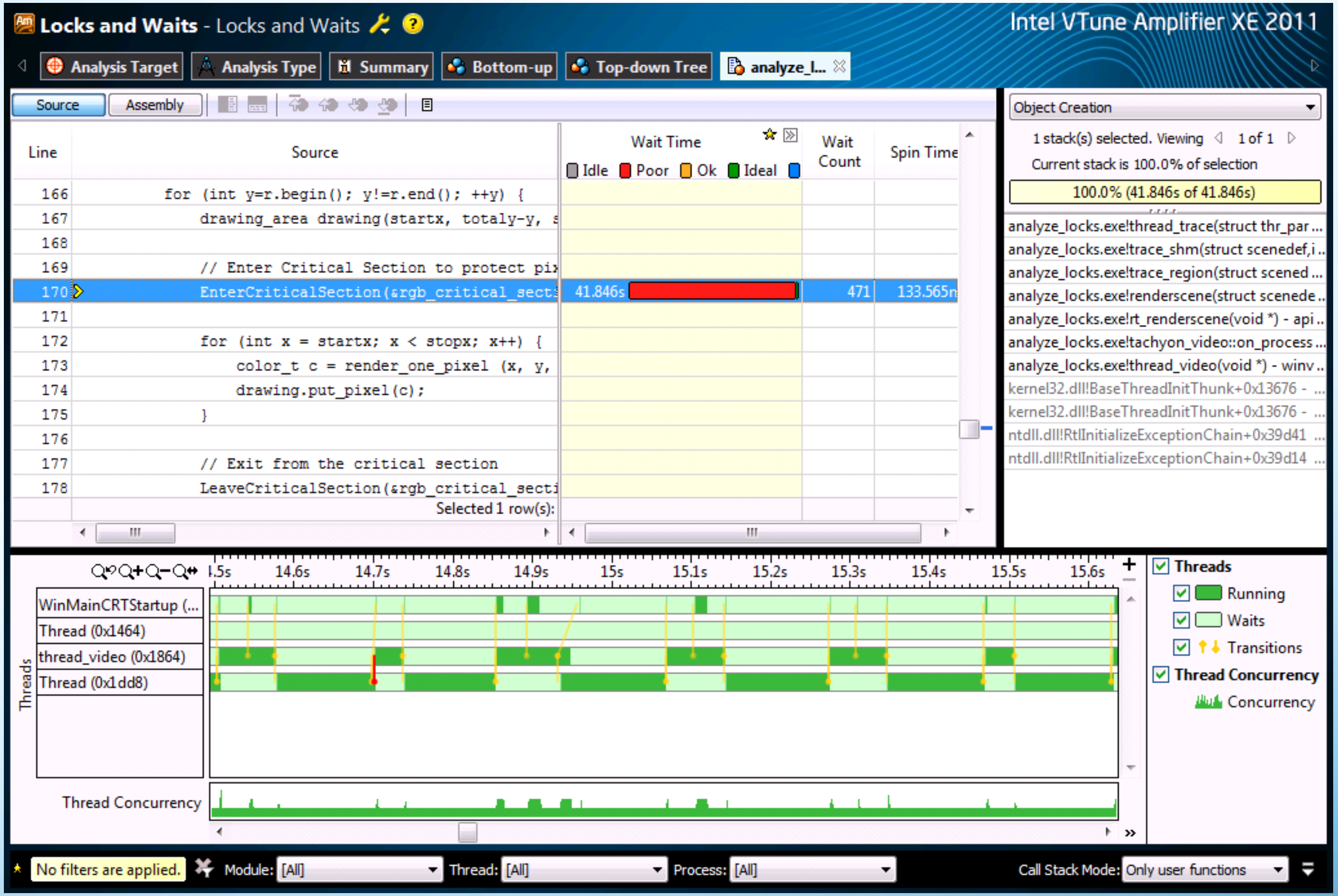
Timeline Visualizes Thread Behavior

- Retrieve additional information on thread transitions



Drilling down into Thread Behavior

- Reveal source code that causes thread transitions



Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE: User Interface
- Fundamental Analysis: Hotspots
- Finding Issues in Parallel Applications
- Using the Performance Monitoring Unit

Performance Monitoring Unit (PMU)

- **Per-core PMU**:
  - Each core provides 2 programmable counters and 1 fixed counters.
  - The programmable per-core counters can be configured to investigate front-end/micro-op flow issues, stalls inside a processor core.
- **Uncore PMU:**
  - Uncore of the coprocessor has four counters to monitor uncore events
  - Can be used to investigate memory behavior and global on-chip issues

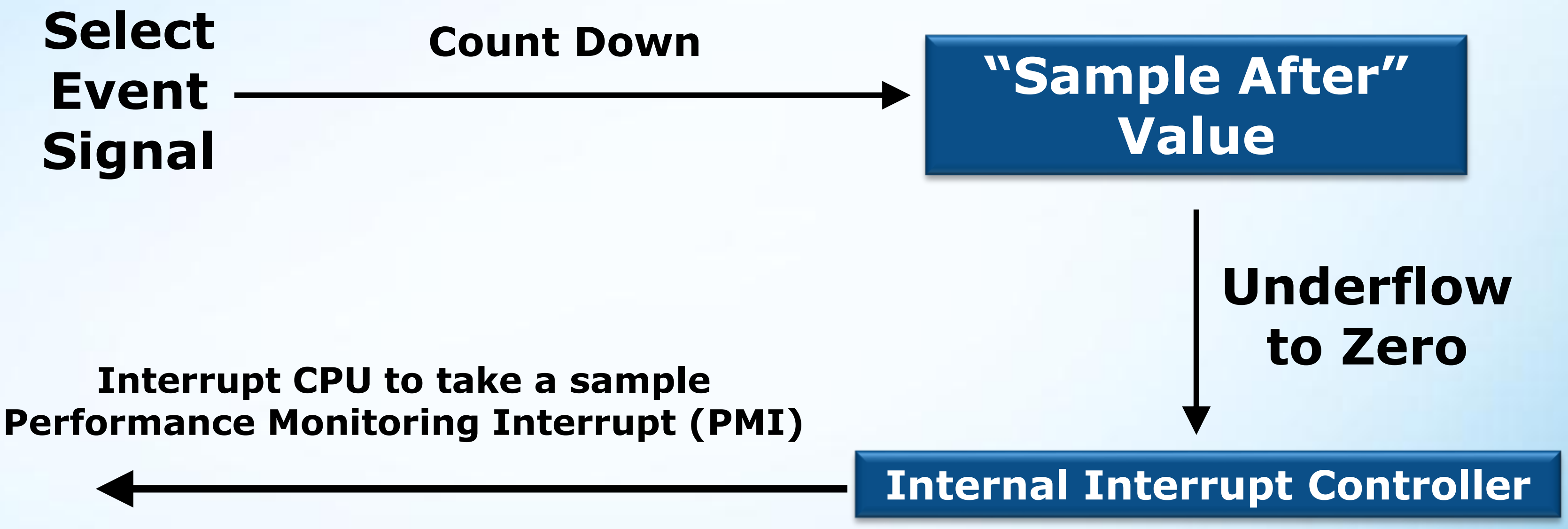
Event Based Performance Analysis

Event Based Sampling(EBS)

- Both architectural and non-architectural processor events can be monitored using sampling and counting technologies

  Sampling: Allows to profile all active software on the system, including operating system, device driver, and application software.
  - Event-based samples are collected periodically after a specific number of processor events have occurred while the program is running
  - The program is interrupted, allowing the interrupt handling driver to collect the Instruction Pointer (IP), load module, thread and process ID's
  - Instruction pointer is then used to derive the function and source line number from the debug information created at compile time

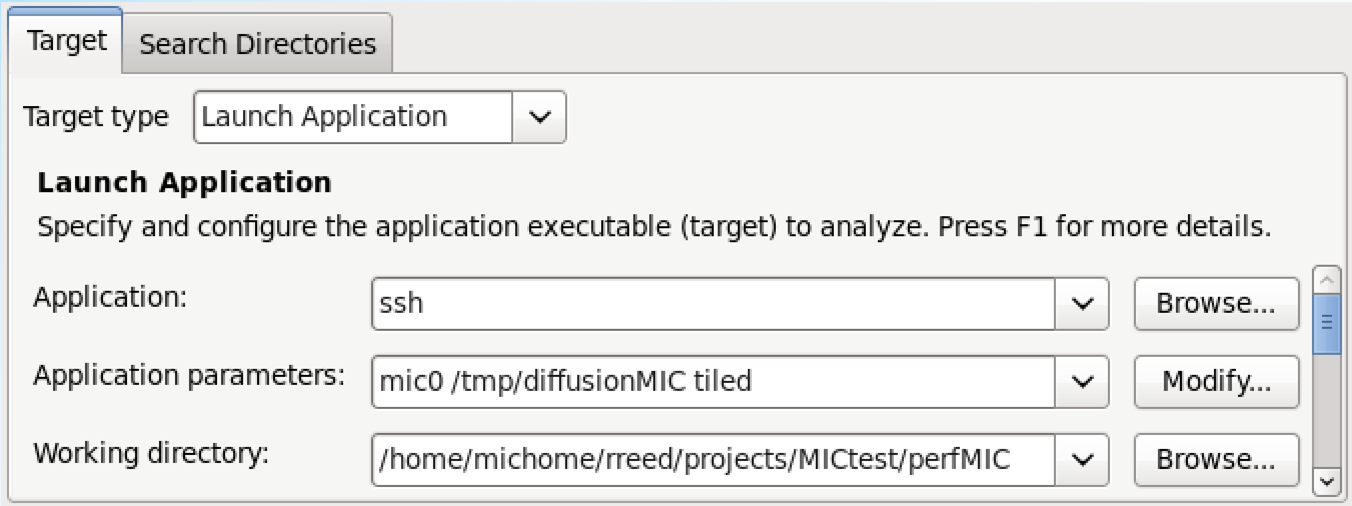
How Event Based Sampling (EBS) Works



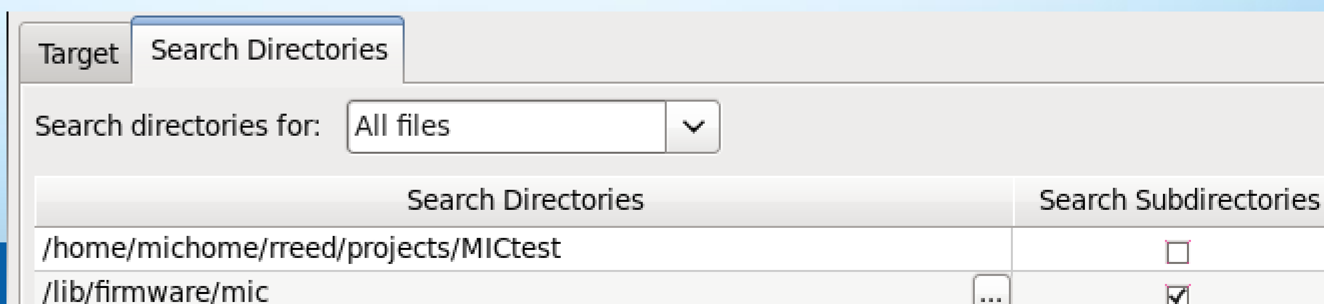
- A performance counter increments on the CPU every time an event occurs
- A sample of the execution context is recorded every time a performance counter overflows

Events = samples * sample after value

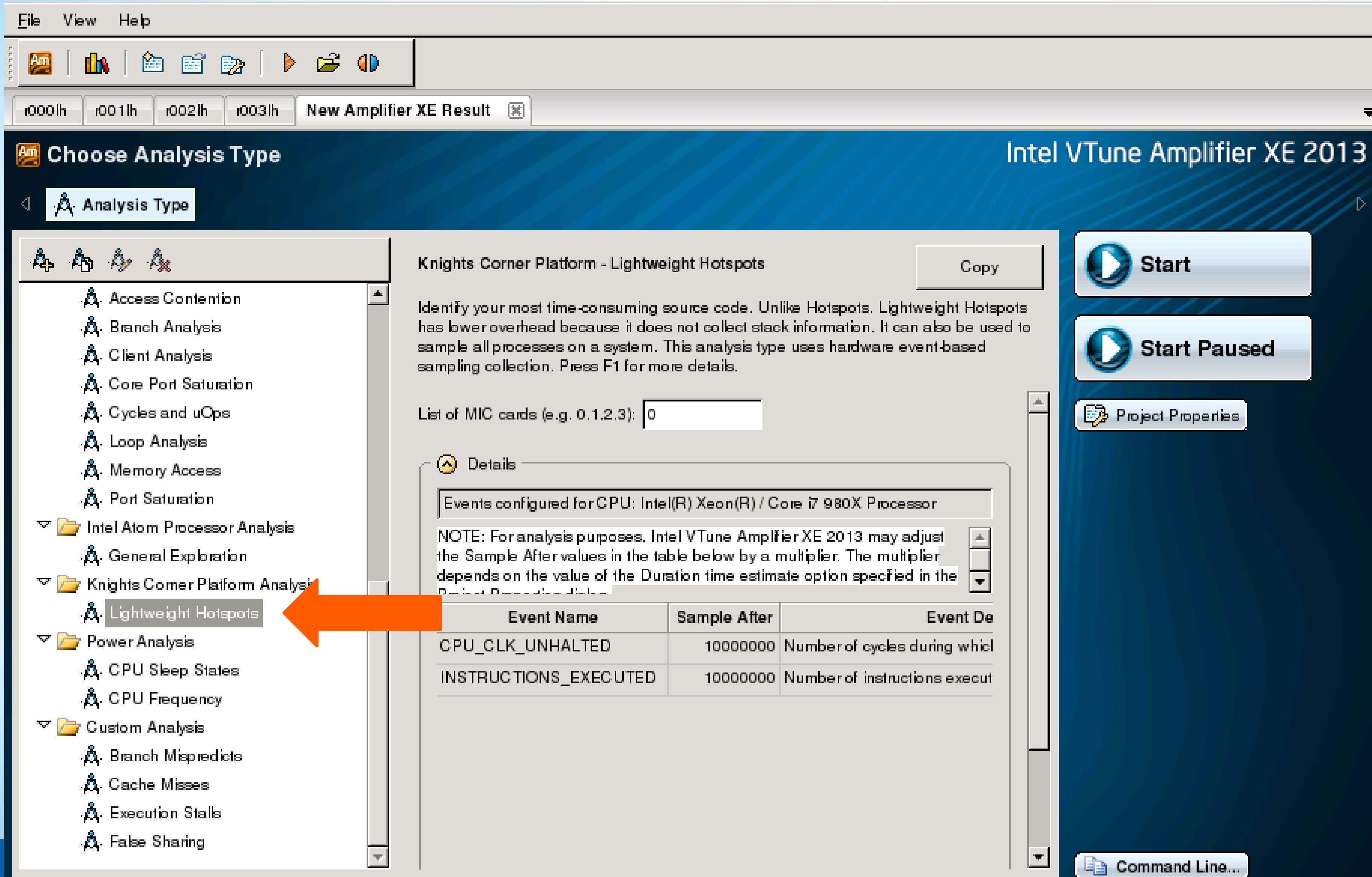
Native Launch configuration



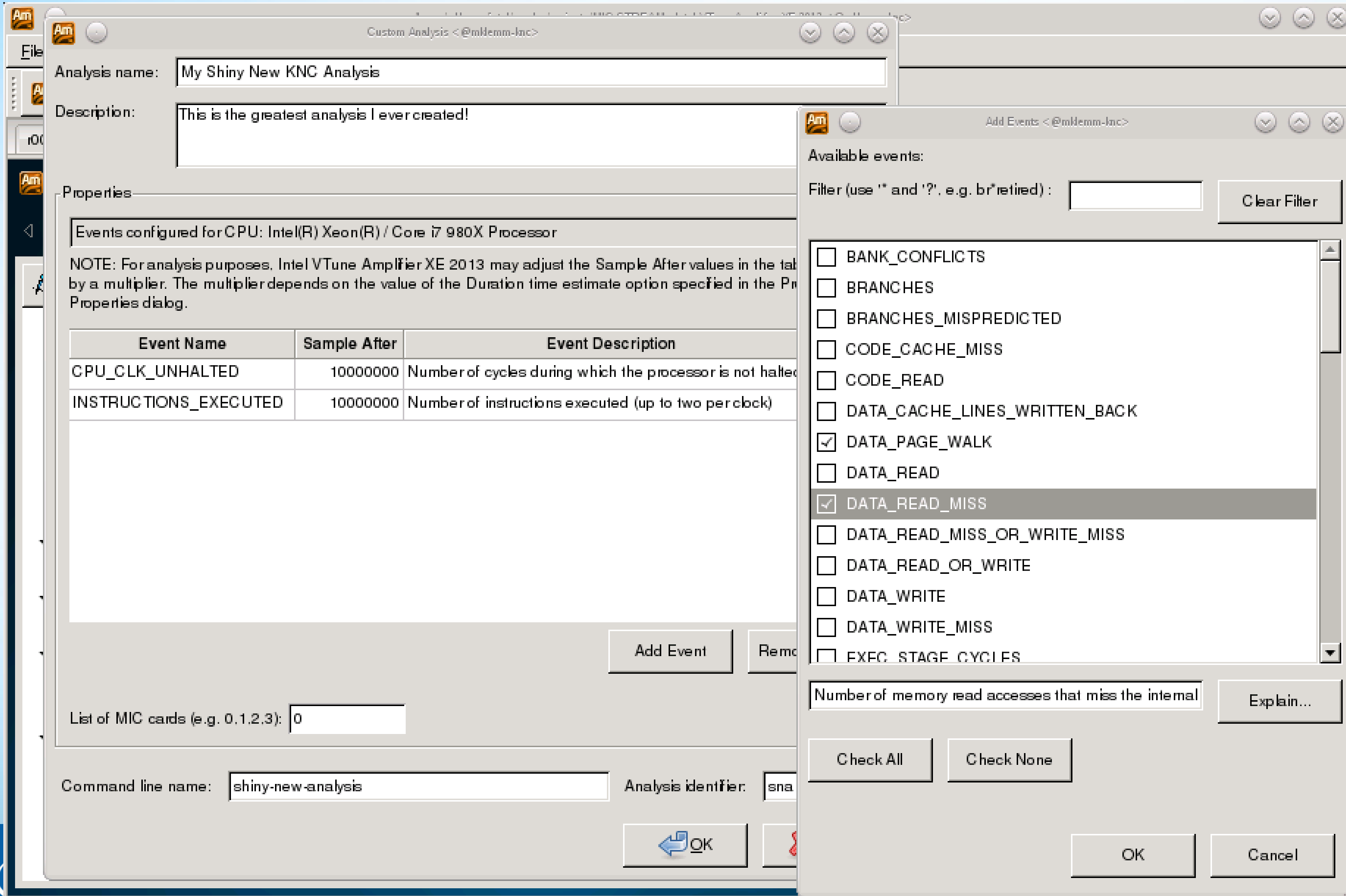
- Application settings:
  - Application: ssh
  - Parameters: mic0 “<app startup>”
  - Working directory: Usually does not matter
  - Don’t forget to set search directories under “All files”



Application Configuration



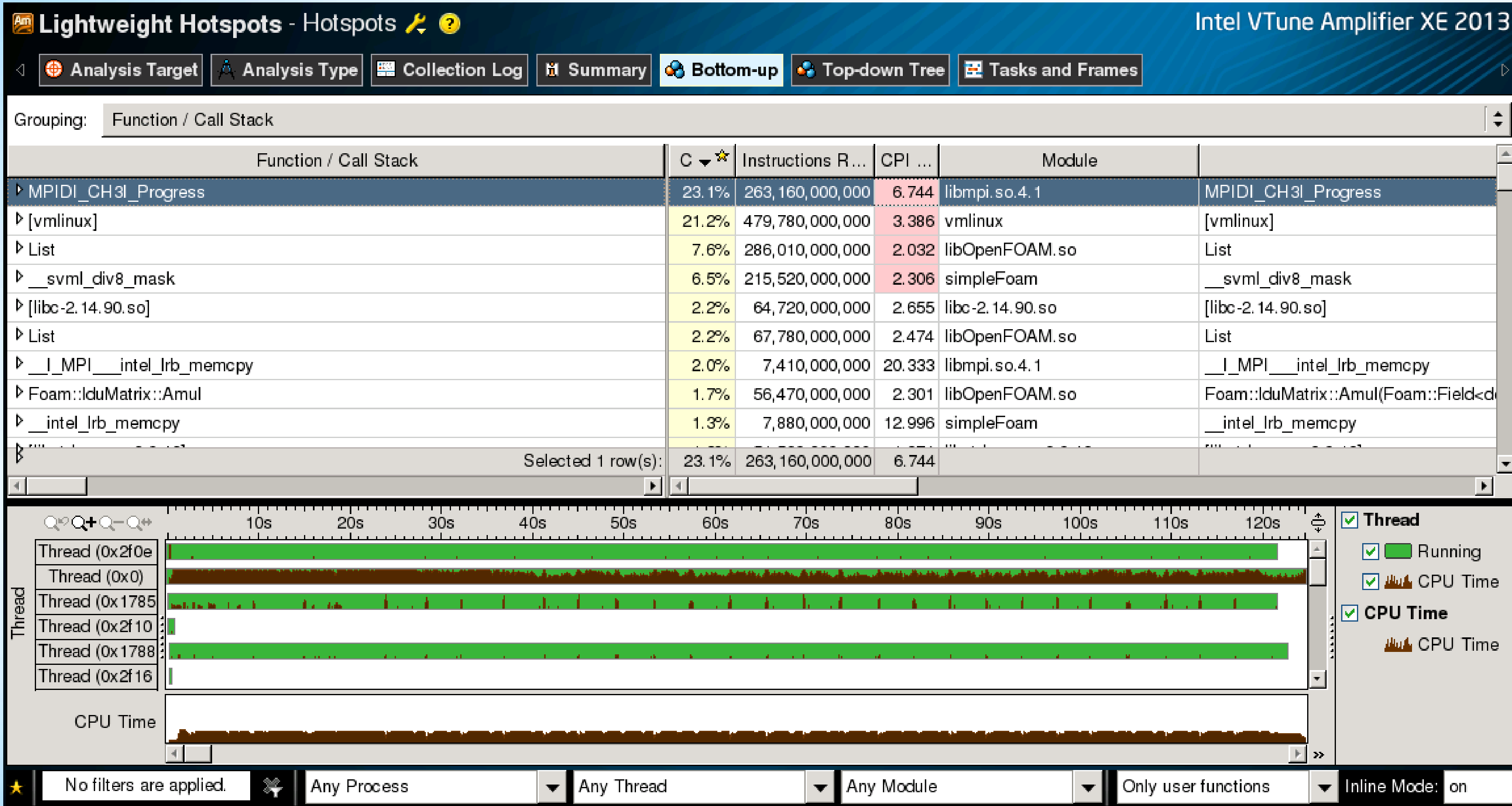
Configuring a User-defined Analysis



Some useful events and metrics

Scenario	Event name(s)
Wall-clock profiling	CPU_CLK_UNHALTED, INSTRUCTIONS_EXECUTED (or EXEC_STAGE_CYCLES)
Main memory bandwidth	L2_DATA_READ_MISS_MEM_FILL, L2_DATA_WRITE_MISS_MEM_FILL
L1 Cache misses	DATA_READ_MISS_OR_WRITE_MISS
TLB misses and page faults	DATA_PAGE_WALK, LONG_DATA_PAGE_WALK, DATA_PAGE_FAULT
Vectorized code execution	VPU_INSTRUCTIONS_EXECUTED, VPU_ELEMENTS_ACTIVE
Various hazards	BRANCHES_MISPREDICTED
Cycles per instruction	CPU_CLK_UNHALTED / INSTRUCTIONS_EXECUTED
Memory Bandwidth (used by all cores at once)	(L2_DATA_READ_MISS_MEM_FILL + L2_DATA_WRITE_MISS_MEM_FILL) * 64 / CPU_CLK_UNHALTED / Frequency

Example: Hotspots of OpenFOAM



Function / Call Stack	C	Instructions R...	CPI ...	Module	
MPIDI_CH3I_Progress	23.1%	263,160,000,000	6.744	libmpi.so.4.1	MPIDI_CH3I_Progress
[vmlinux]	21.2%	479,780,000,000	3.386	vmlinux	[vmlinux]
List	7.6%	286,010,000,000	2.032	libOpenFOAM.so	List
__svml_div8_mask	6.5%	215,520,000,000	2.306	simpleFoam	__svml_div8_mask
[libc-2.14.90.so]	2.2%	64,720,000,000	2.655	libc-2.14.90.so	[libc-2.14.90.so]
List	2.2%	67,780,000,000	2.474	libOpenFOAM.so	List
__I_MPI___intel_lrb_memcpy	2.0%	7,410,000,000	20.333	libmpi.so.4.1	__I_MPI___intel_lrb_memcpy
Foam::lduMatrix::Amul	1.7%	56,470,000,000	2.301	libOpenFOAM.so	Foam::lduMatrix::Amul(Foam::Field<d
__intel_lrb_memcpy	1.3%	7,880,000,000	12.996	simpleFoam	__intel_lrb_memcpy