Advanced Tuning Methodologies

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

Legal Disclaimer & Optimization Notice

Optimization Notice

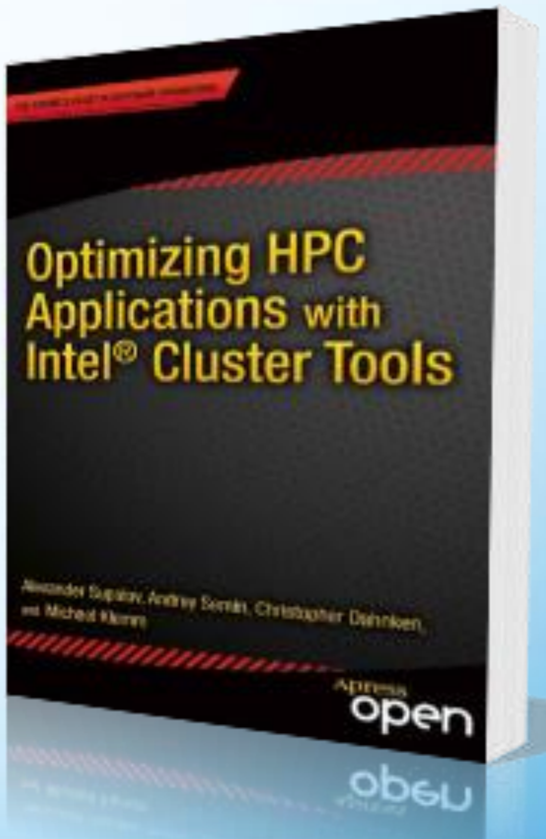
Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

Shameless Plug…

Authors: Alexander Supalov, Andrey Semin, Michael Klemm, Chris Dahnken

Table of Contents:


Optimizing HPC Applications with Intel® Cluster Tools
Alexander Supalov, Andrey Semin, Christopher Dahnken and Michael Klemm
Apress open

ISBN-13 (pbk): 978-1-4302-6496-5
ISBN-13 (electronic): 978-1-4302-6497-2

Order now at http://www.apress.com/9781430264965

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

IN R 805

The Software Optimization Process

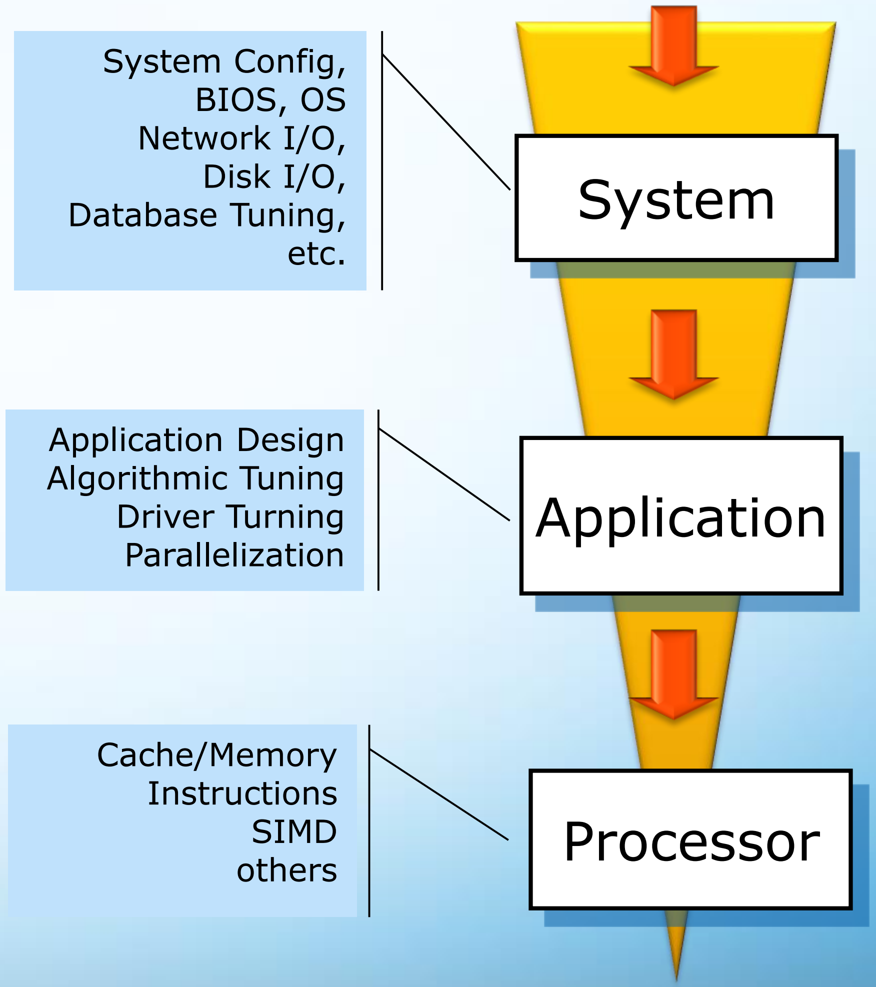
- The process of improving the software by eliminating bottlenecks so that it operates more efficiently on a given hardware and uses resources optimally
  - Identifying bottlenecks in the target application and eliminating them appropriately is the key to an efficient optimization

- 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.

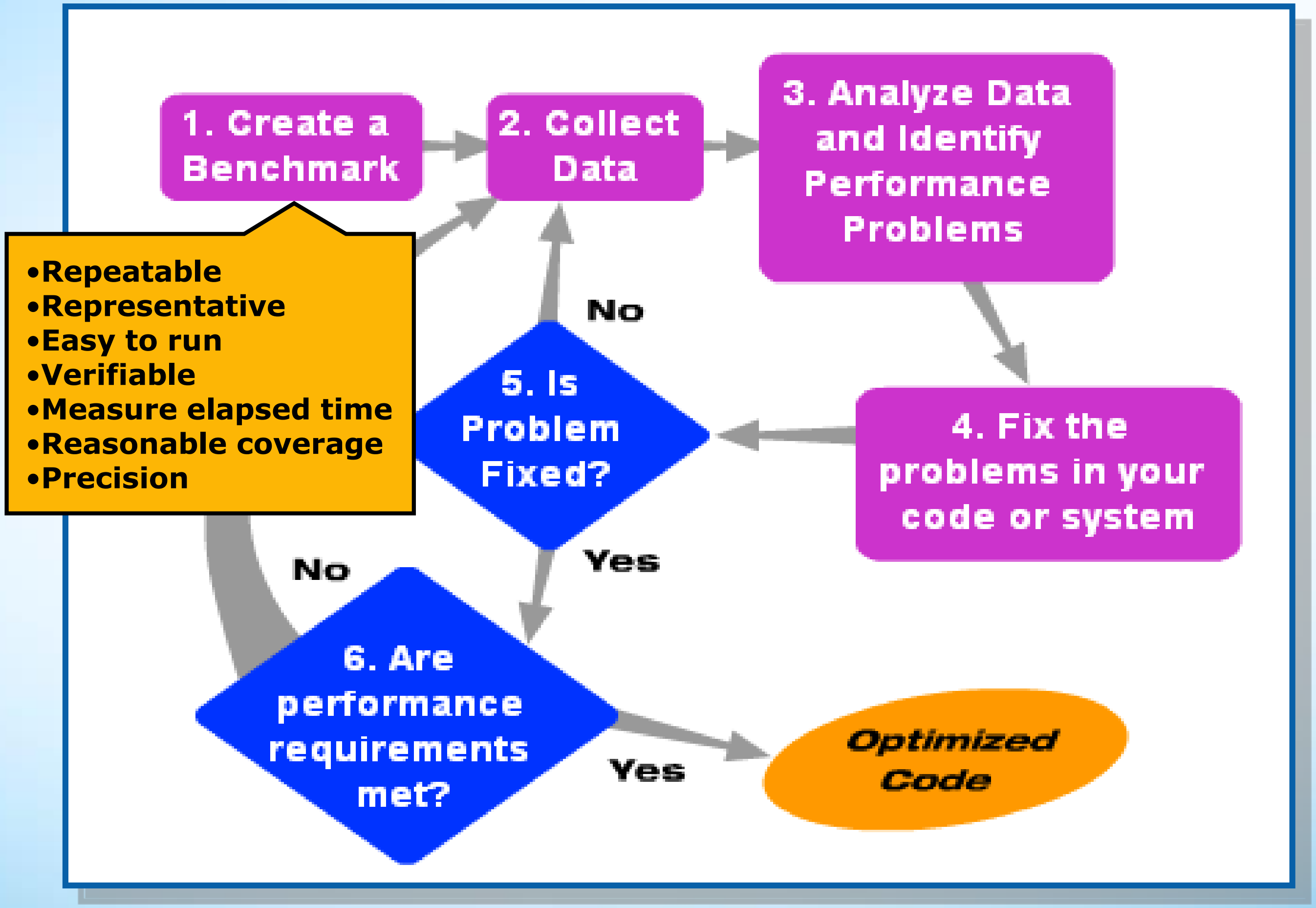
Performance Analysis Methodology
Optimization: A Top-down Approach

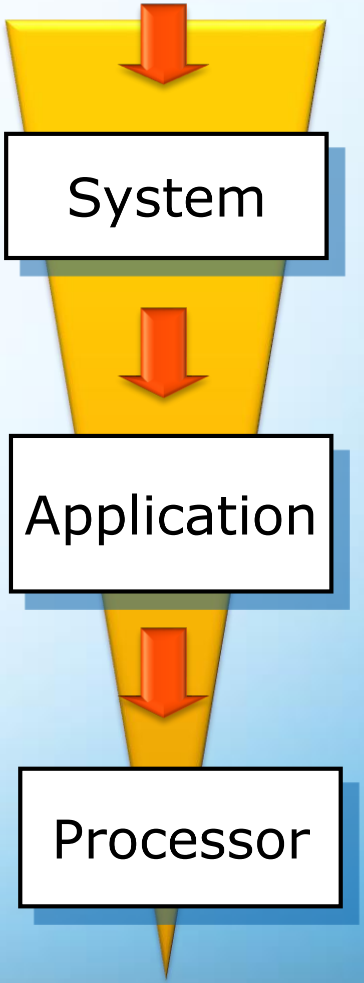
- Use top down approach
- Understand application and system characteristics
  - Use appropriate tools at each level


System Config, BIOS, OS Network I/O, Disk I/O, Database Tuning, etc.
System
Application Design Algorithmic Tuning Driver Turning Parallelization
Application
Cache/Memory Instructions SIMD others
Processor

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."

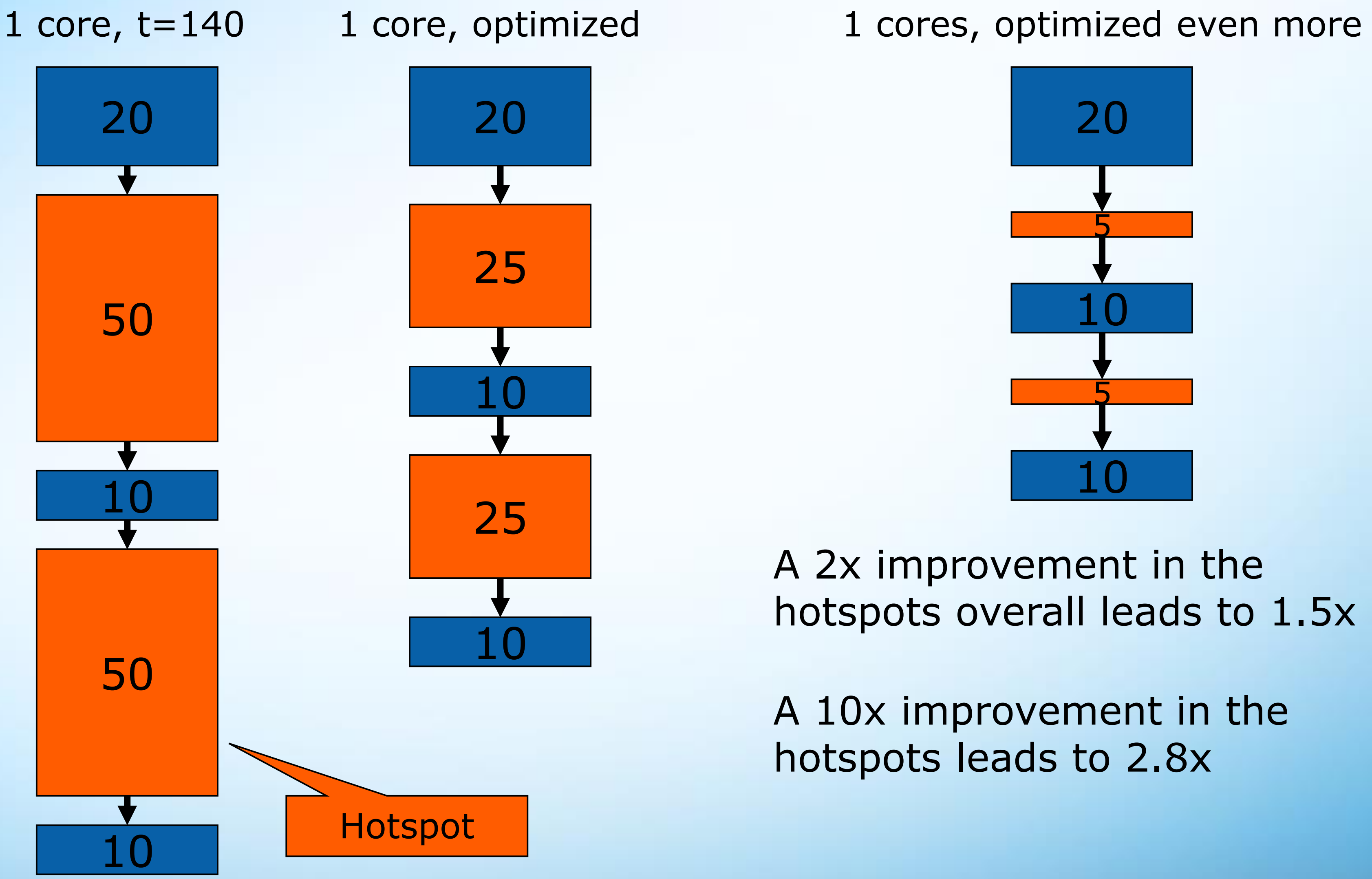
— Donald Knuth

Quality code is:

- Portable
- Readable
- Maintainable
- Reliable

Intelligently sacrifice quality for performance

Amdahl's Law Slightly Reinterpreted



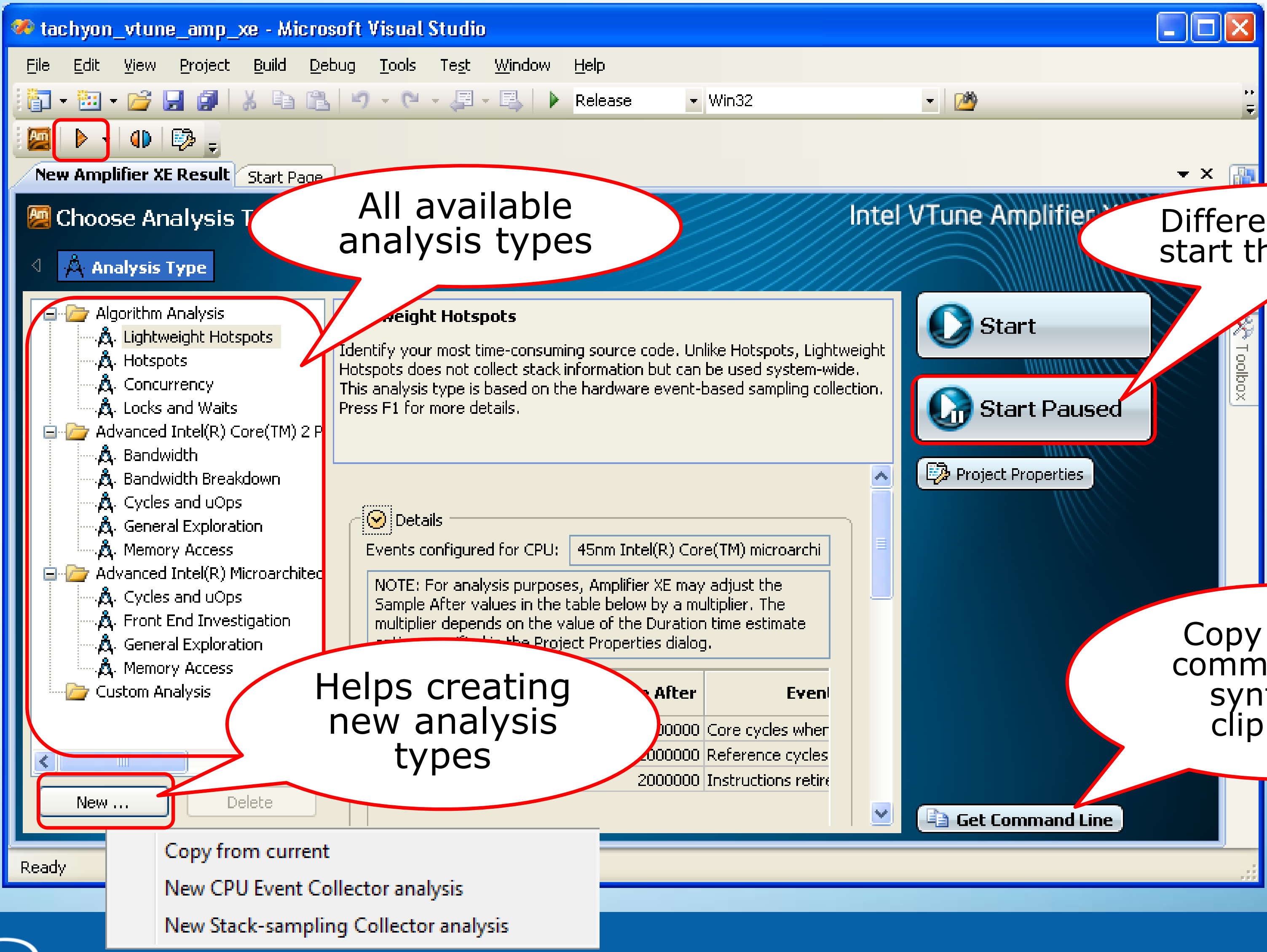
A 2x improvement in the hotspots overall leads to 1.5x

A 10x improvement in the hotspots leads to 2.8x

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

Selecting type of data collection



weight Hotspots

Identify your most time-consuming source code. Unlike Hotspots, Lightweight Hotspots does not collect stack information but can be used system-wide. This analysis type is based on the hardware event-based sampling collection. Press F1 for more details.

Details

Events configured for CPU: 45nm Intel(R) Core(TM) microarchi

NOTE: For analysis purposes, Amplifier XE may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate

the Project Properties dialog.

After	Event
00000	Core cycles wher
000000	Reference cycles
2000000	Instructions retire

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)

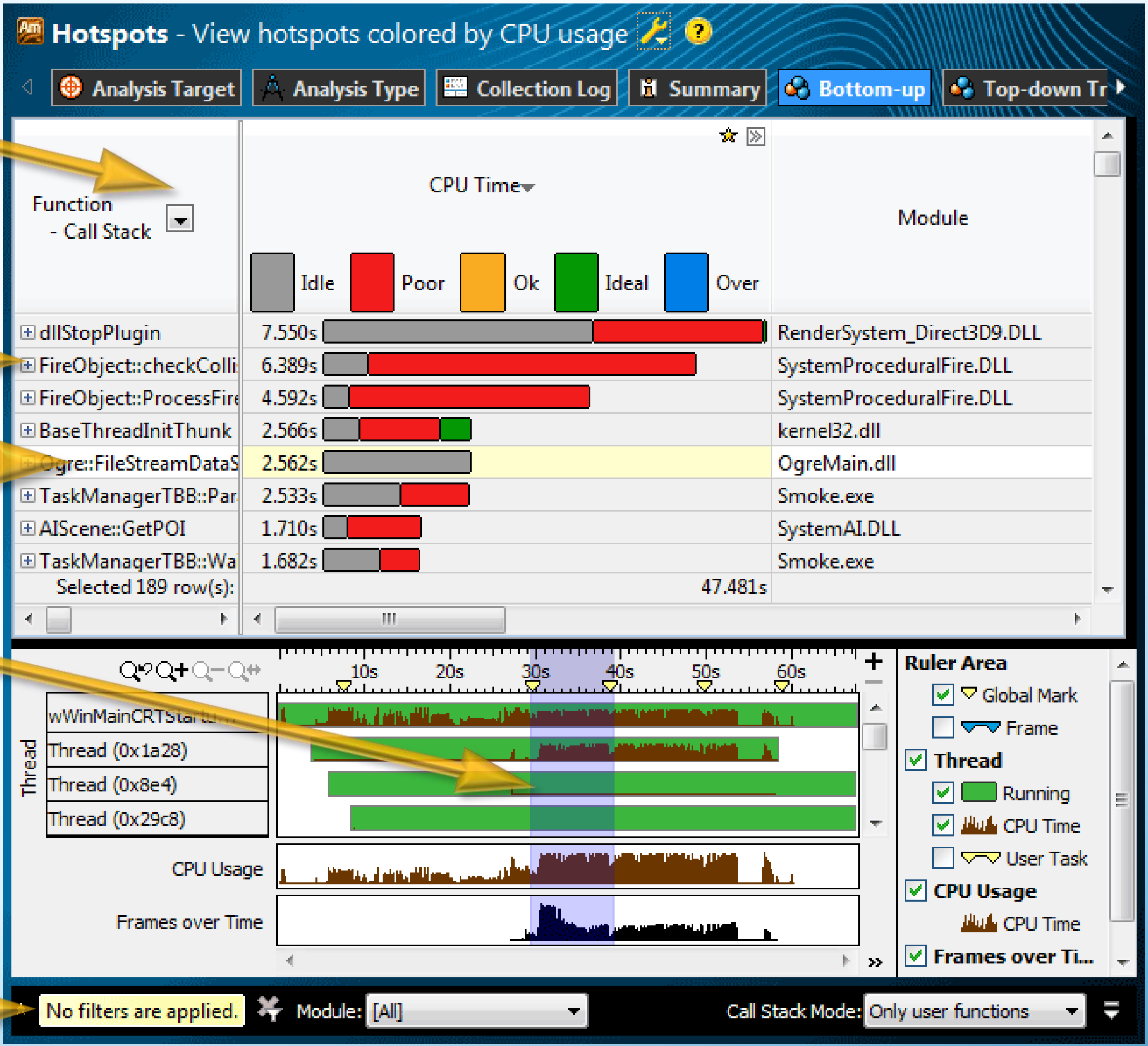
Click [+] for Call Stack

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

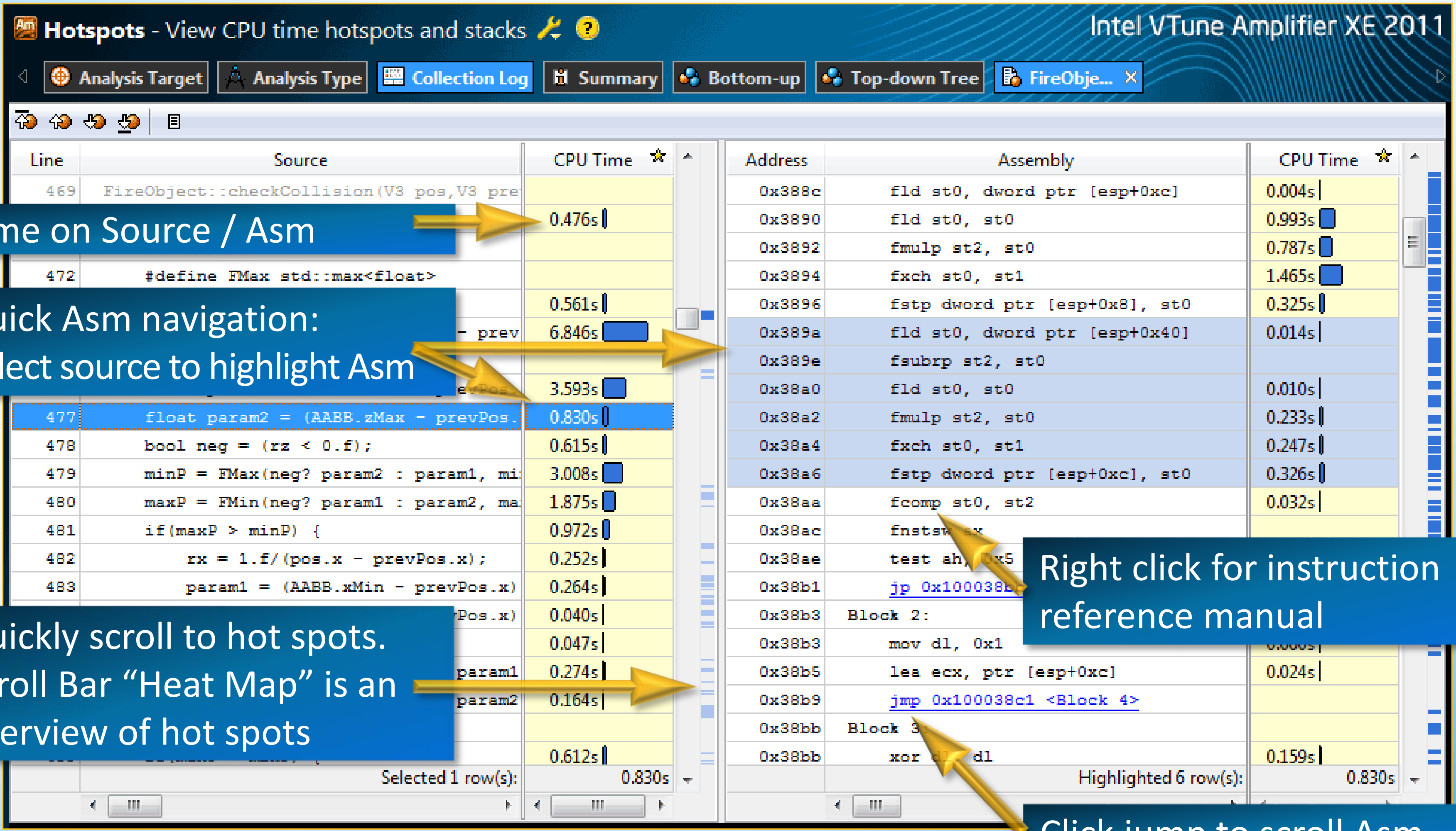

Hotspots - View hotspots colored by CPU usage

Function - Call Stack	CPU Time: 🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Module
⊞ dllStopPlugin	7.550s	RenderSystem_Direct3D9.DLL
⊞ FireObject::checkColli	6.389s	SystemProceduralFire.DLL
⊞ FireObject::ProcessFire	4.592s	SystemProceduralFire.DLL
⊞ BaseThreadInitThunk	2.566s	kernel32.dll
⊞ Ogre::FileStreamDataS	2.562s	OgreMain.dll
⊞ TaskManagerTBB::Par	2.533s	Smoke.exe
⊞ AIScene::GetPOI	1.710s	SystemAI.DLL
⊞ TaskManagerTBB::Wa	1.682s	Smoke.exe
Selected 189 row(s):	47.481s	

No filters are applied. Module: [All] Call Stack Mode: Only user functions

VTune™ Amplifier XE

GUI Layout



Line	Source	CPU Time
469	FireObject::checkCollision(V3 pos,V3 pre	
		0.476s
472	#define FMax std::max<float>	
		0.561s
	- prev	6.846s
	evPos	3.593s
477	float param2 = (AABB.zMax - prevPos.	0.830s
478	bool neg = (rz < 0.f);	0.615s
479	minP = FMax(neg? param2 : param1, mi	3.008s
480	maxP = FMin(neg? param1 : param2, ma	1.875s
481	if(maxP > minP) {	0.972s
482	rx = 1.f/(pos.x - prevPos.x);	0.252s
483	param1 = (AABB.xMin - prevPos.x)	0.264s
	Pos.x)	0.040s
		0.047s
	param1	0.274s
	param2	0.164s
		0.612s
	Selected 1 row(s):	0.830s

Address	Assembly	CPU Time
0x388c	fld st0, dword ptr [esp+0xc]	0.004s
0x3890	fld st0, st0	0.993s
0x3892	fmulp st2, st0	0.787s
0x3894	fxch st0, st1	1.465s
0x3896	fstp dword ptr [esp+0x8], st0	0.325s
0x389a	fld st0, dword ptr [esp+0x40]	0.014s
0x389e	fsubrp st2, st0	
0x38a0	fld st0, st0	0.010s
0x38a2	fmulp st2, st0	0.233s
0x38a4	fxch st0, st1	0.247s
0x38a6	fstp dword ptr [esp+0xc], st0	0.326s
0x38aa	fcomp st0, st2	0.032s
0x38ac	fnstsw ax	
0x38ae	test ah, 0x5	
0x38b1	jp 0x100038b	
0x38b3	Block 2:	
0x38b3	mov dl, 0x1	
0x38b5	lea ecx, ptr [esp+0xc]	0.024s
0x38b9	jmp 0x100038c1 <Block 4>	
0x38bb	Block 3:	
0x38bb	xor dl, dl	0.159s
	Highlighted 6 row(s):	0.830s

Time on Source / Asm

Quick Asm navigation:
Select source to highlight Asm

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

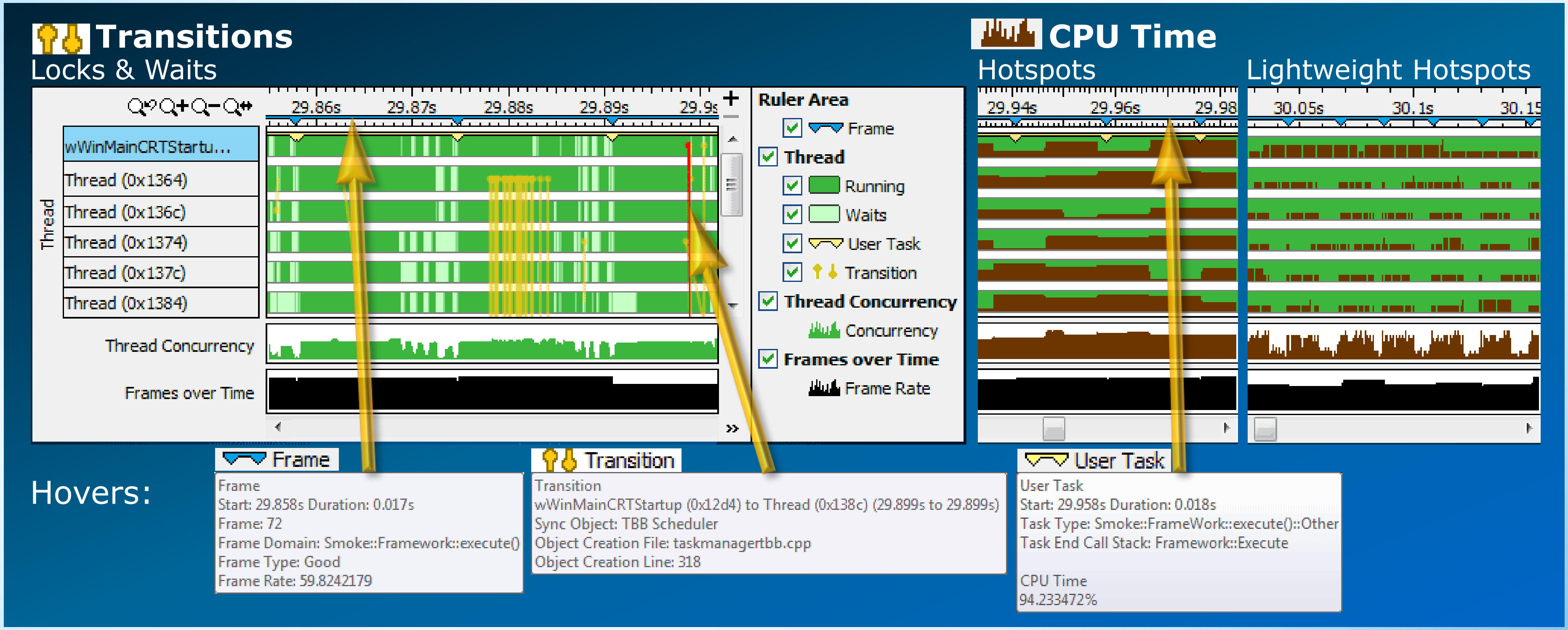
Right click for instruction reference manual

Click jump to scroll Asm

Intel® VTune™ Amplifier XE

VTune™ Amplifier XE

GUI Layout



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

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

User Task
Start: 29.958s Duration: 0.018s
Task Type: Smoke::FrameWork::execute()::Other
Task End Call Stack: Framework::Execute

CPU Time
94.233472%

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

Software & Services Group, Developer Products Division

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

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

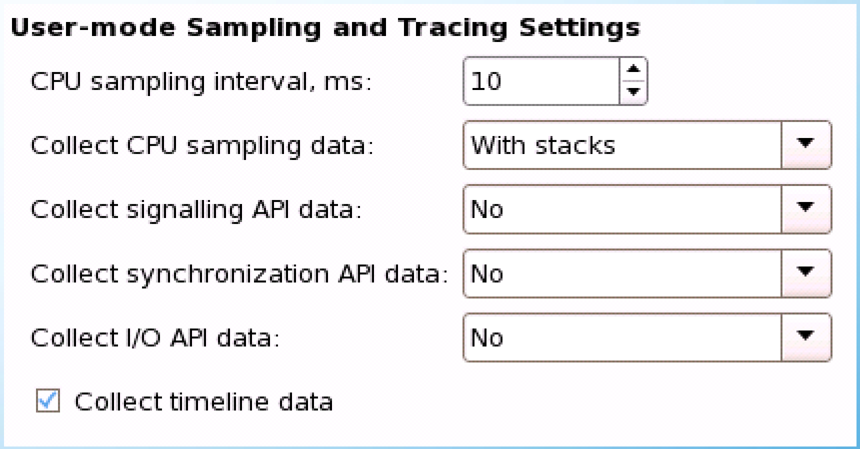
- Intel Threading Runtimes need instrumented runtimes
  - TBB: Define TBB_USE_THREADING_TOOLS
  - OpenMP: Use Intel Dynamic Version of OpenMP

- 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

Basic 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



User-mode Sampling and Tracing Settings

CPU sampling interval, ms:	10
Collect CPU sampling data:	With stacks
Collect signalling API data:	No
Collect synchronization API data:	No
Collect I/O API data:	No

☑ Collect timeline data

Analysis Types
Advanced Hotspots

- Similar to Hotspot Analysis
  - Sampling is performed with the SEP collector
  - Driver is required
- Stack walking can be performed
  - Only hotspots are reported by default
  - More complex data collection is possible, incl. trip counts
- 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

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

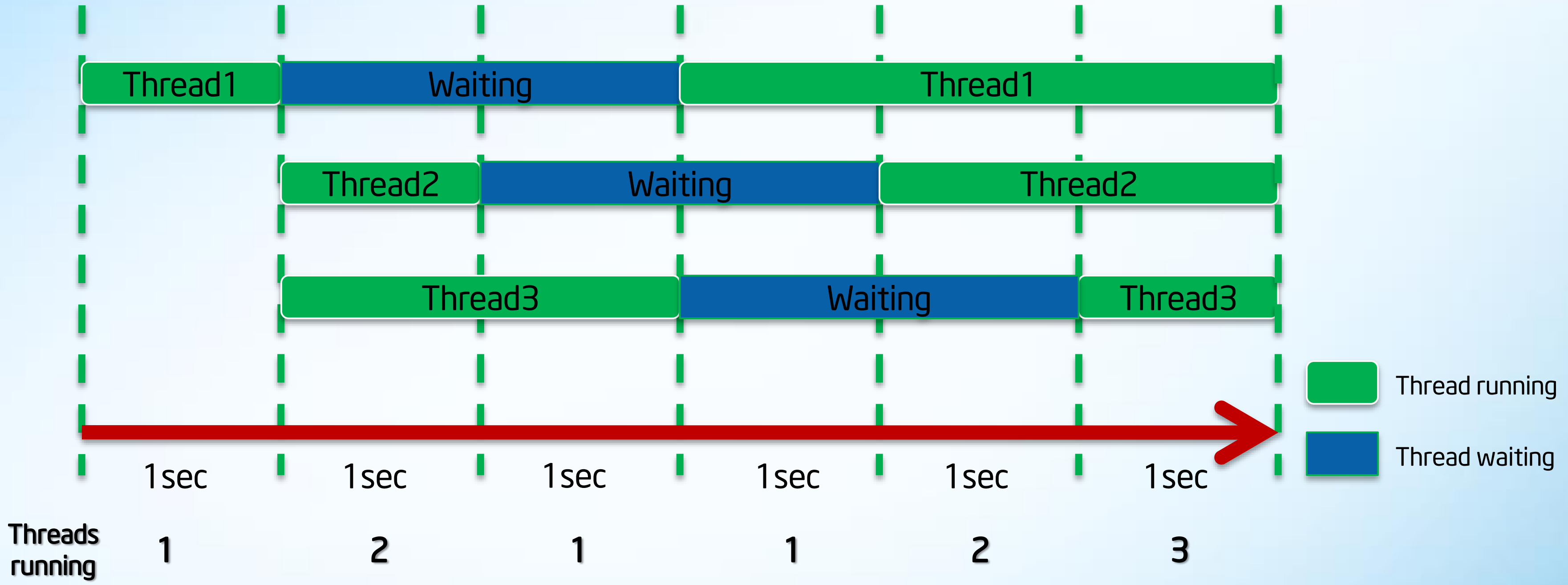
Issues in Parallel Applications

- 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)

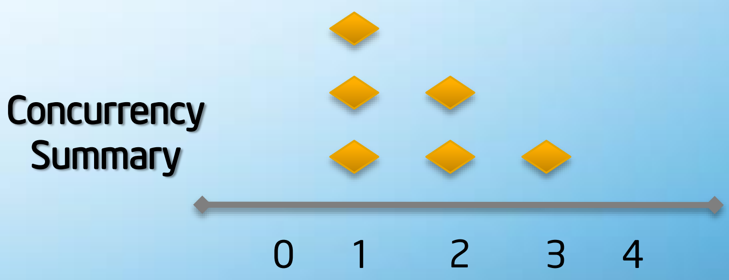
Issues in Parallel Applications

- 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)

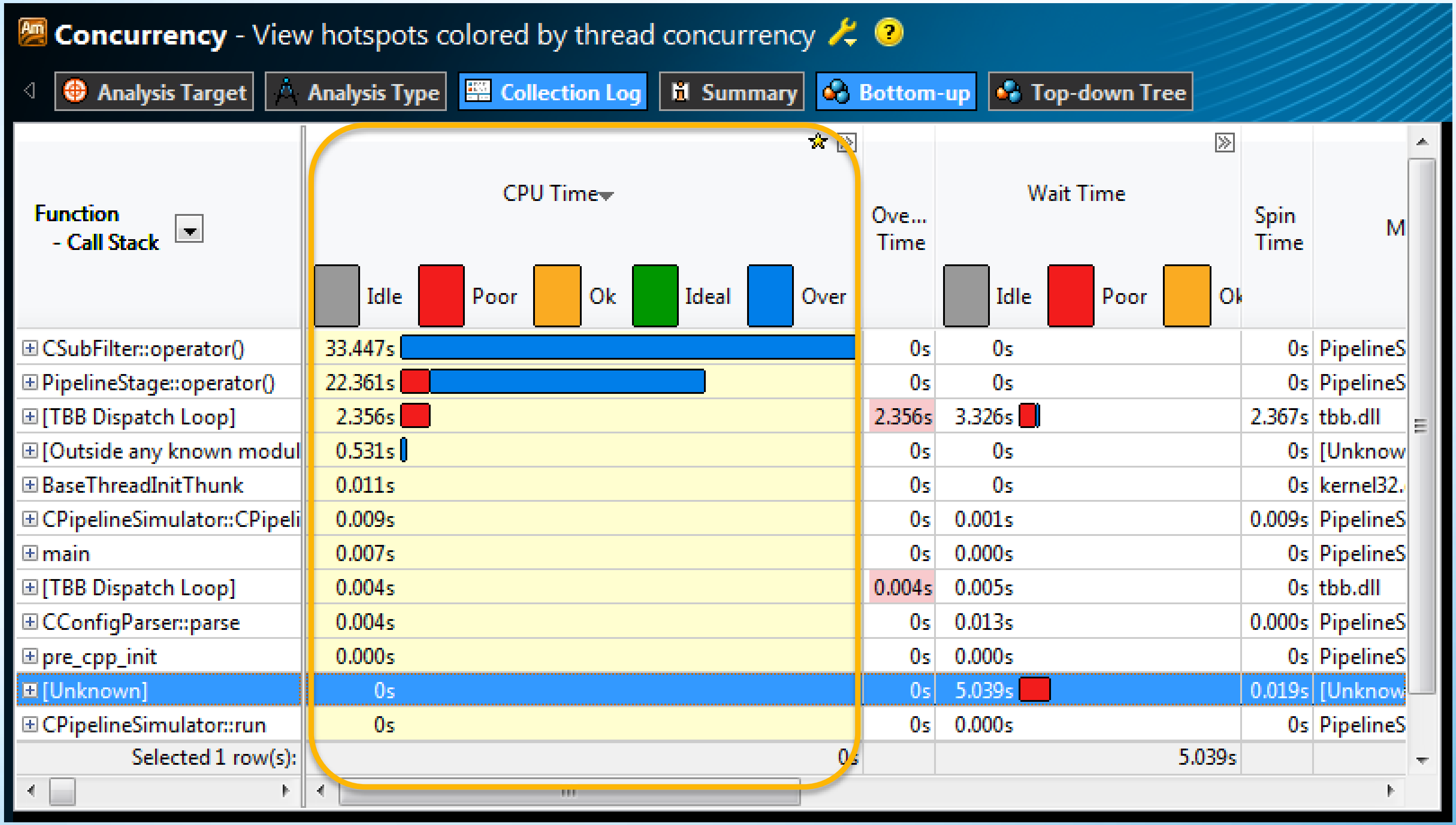
Threading Analysis Terminology



- **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

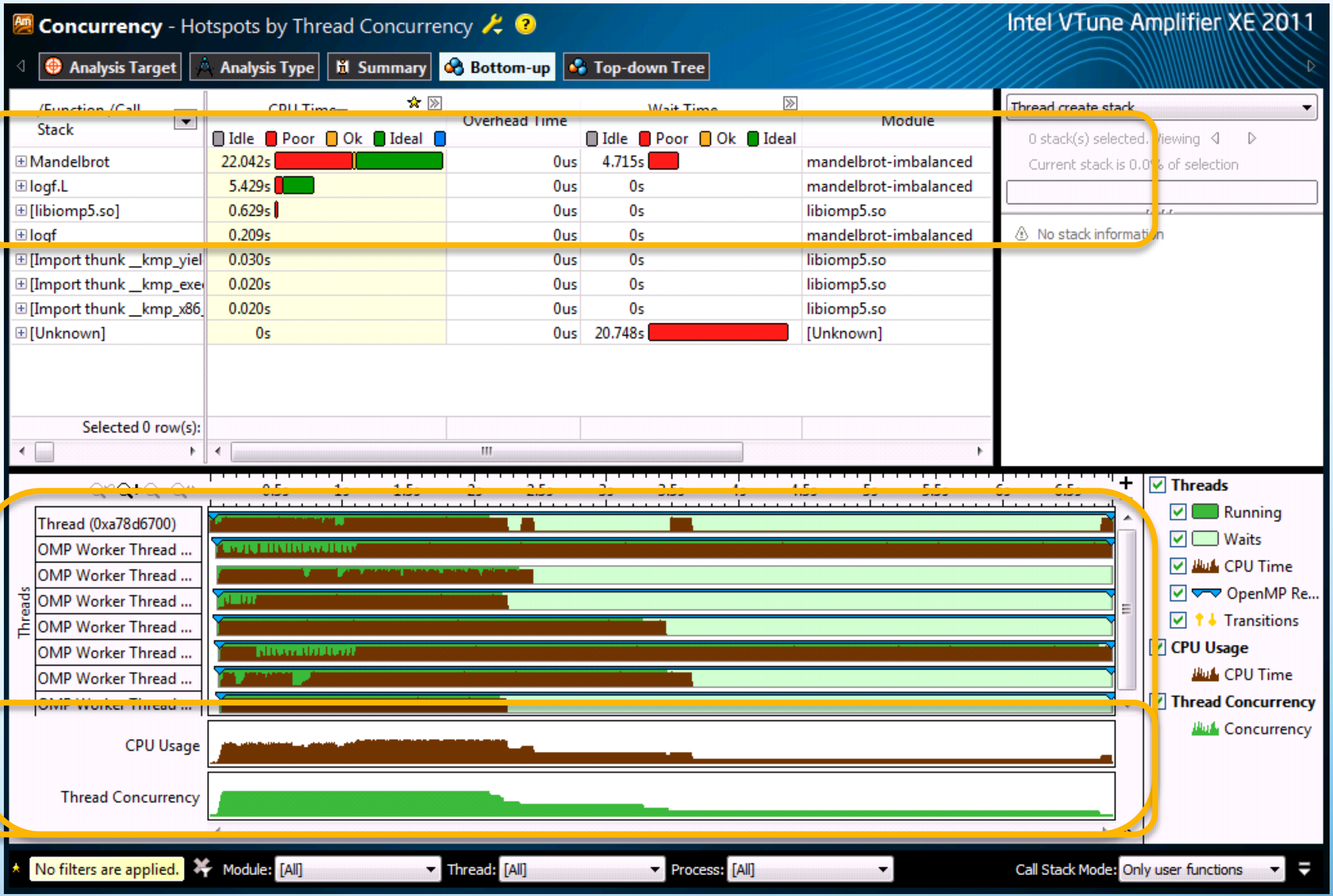


Concurrency - View hotspots colored by thread concurrency

Function - Call Stack	CPU Time (▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Ove... Time	Wait Time (▨ Idle 🟥 Poor 🟧 Ok)	Spin Time	M
⊞ CSubFilter::operator()	33.447s 🟦	0s	0s	0s	PipelineS
⊞ PipelineStage::operator()	22.361s 🟥🟦	0s	0s	0s	PipelineS
⊞ [TBB Dispatch Loop]	2.356s 🟥	2.356s	3.326s 🟥	2.367s	tbb.dll
⊞ [Outside any known modul	0.531s 🟦	0s	0s	0s	[Unknow
⊞ BaseThreadInitThunk	0.011s	0s	0s	0s	kernel32.
⊞ CPipelineSimulator::CPipeli	0.009s	0s	0.001s	0.009s	PipelineS
⊞ main	0.007s	0s	0.000s	0s	PipelineS
⊞ [TBB Dispatch Loop]	0.004s	0.004s	0.005s	0s	tbb.dll
⊞ CConfigParser::parse	0.004s	0s	0.013s	0.000s	PipelineS
⊞ pre_cpp_init	0.000s	0s	0.000s	0s	PipelineS
⊞ [Unknown]	0s	0s	5.039s 🟥	0.019s	[Unknow
⊞ CPipelineSimulator::run	0s	0s	0.000s	0s	PipelineS
Selected 1 row(s):	0s		5.039s		

Hotspots Analysis vs. Concurrency Analysis

- Hotspot Analysis and Concurrency Analysis are similar:



Concurrency - Hotspots by Thread Concurrency

Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Summary | Bottom-up | Top-down Tree

/Function /Call Stack	CPU Time: Idle / Poor / Ok / Ideal	Overhead Time	Wait Time: Idle / Poor / Ok / Ideal	Module
⊞Mandelbrot	22.042s	0us	4.715s	mandelbrot-imbalanced
⊞logf.L	5.429s	0us	0s	mandelbrot-imbalanced
⊞[libiomp5.so]	0.629s	0us	0s	libiomp5.so
⊞logf	0.209s	0us	0s	mandelbrot-imbalanced
⊞[Import thunk __kmp_yiel	0.030s	0us	0s	libiomp5.so
⊞[Import thunk __kmp_exe	0.020s	0us	0s	libiomp5.so
⊞[Import thunk __kmp_x86_	0.020s	0us	0s	libiomp5.so
⊞[Unknown]	0s	0us	20.748s	[Unknown]

Selected 0 row(s):

Thread create stack

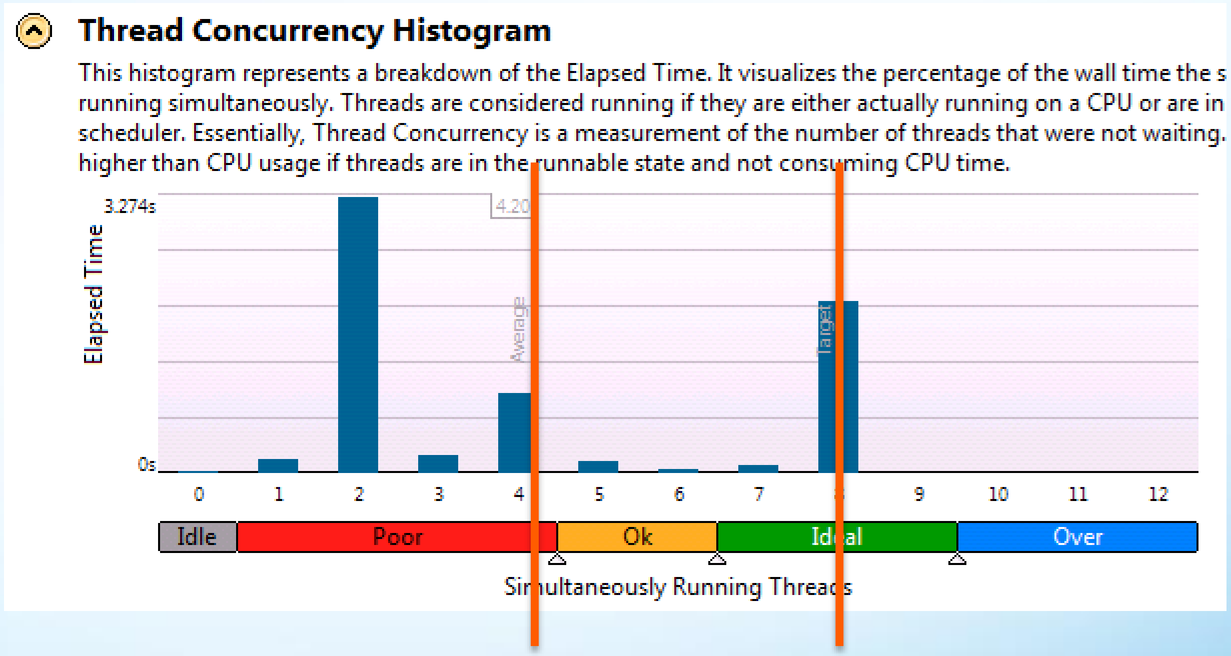
0 stack(s) selected. Viewing

Current stack is 0.0% of selection

No stack information

No filters are applied. Module: [All] Thread: [All] Process: [All] Call Stack Mode: Only user functions

Thread Concurrency Histogram



Thread Concurrency Histogram

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

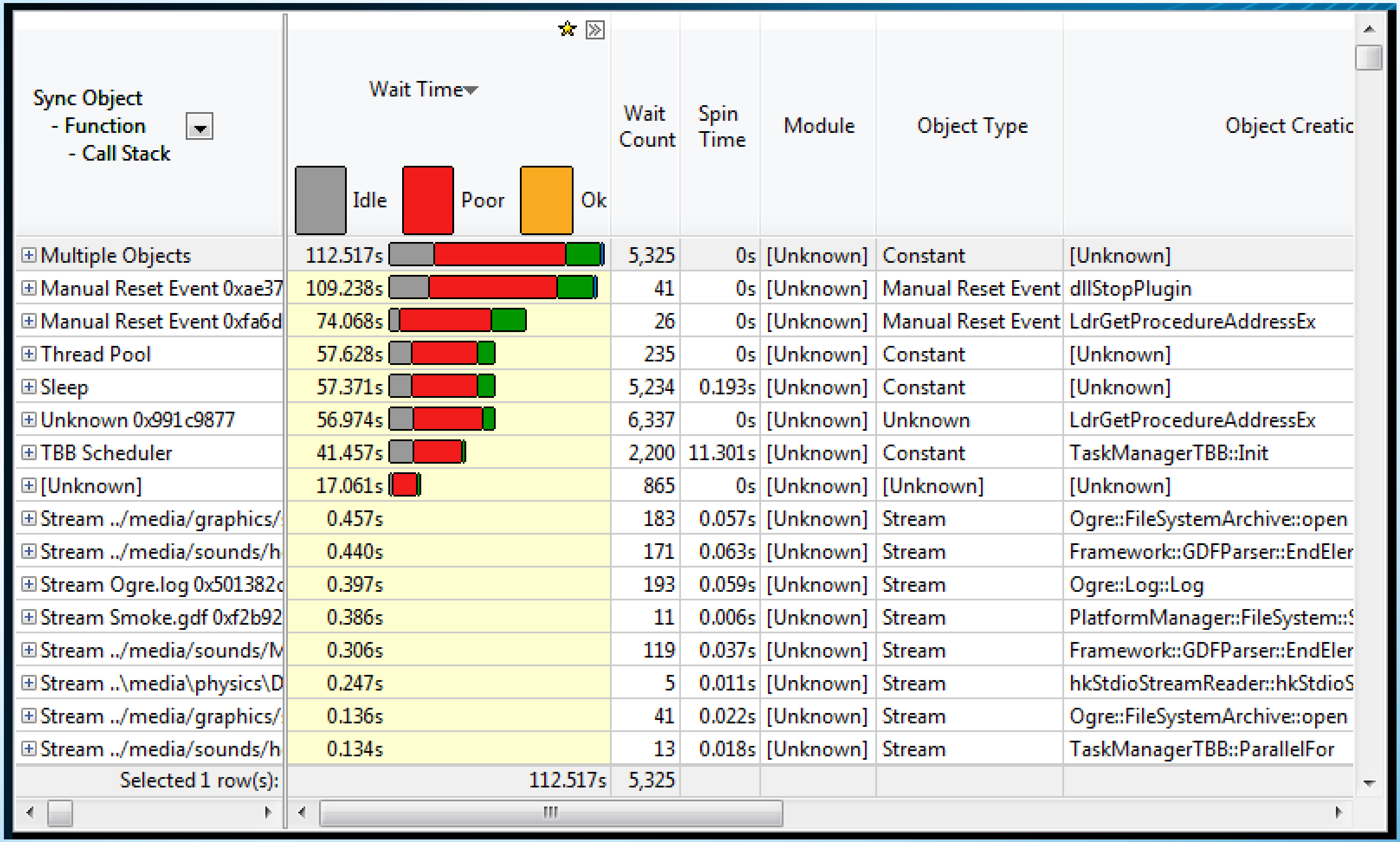
Average Target

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

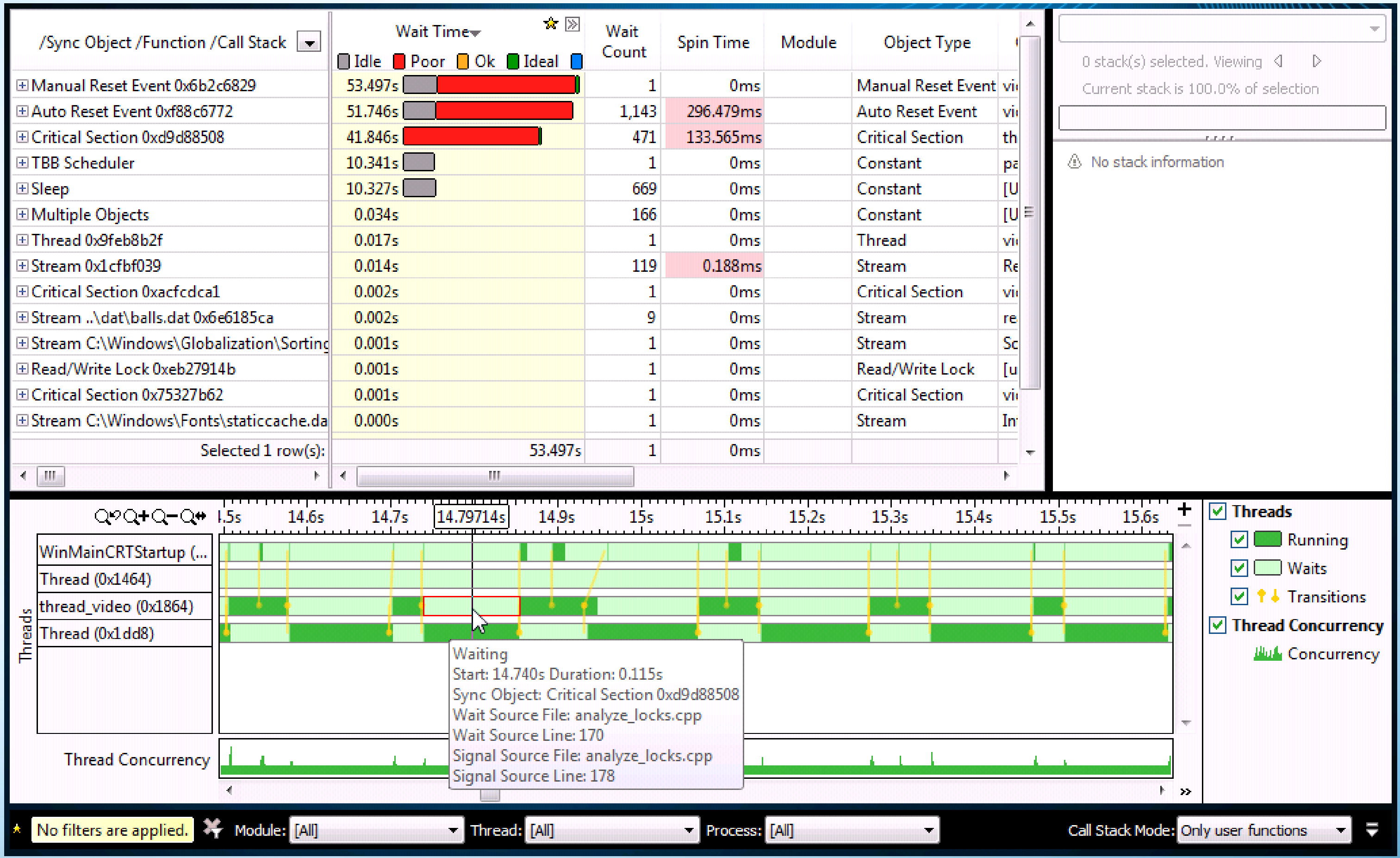
Lock and Waits



Sync Object - Function - Call Stack	Wait Time (▨ Idle 🟥 Poor 🟧 Ok)	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



/Sync Object /Function /Call Stack	Wait Time (Idle / Poor / Ok / Ideal)	Wait Count	Spin Time	Module	Object Type	
⊞Manual Reset Event 0x6b2c6829	53.497s	1	0ms		Manual Reset Event	vi
⊞Auto Reset Event 0xf88c6772	51.746s	1,143	296.479ms		Auto Reset Event	vi
⊞Critical Section 0xd9d88508	41.846s	471	133.565ms		Critical Section	th
⊞TBB Scheduler	10.341s	1	0ms		Constant	pa
⊞Sleep	10.327s	669	0ms		Constant	[U
⊞Multiple Objects	0.034s	166	0ms		Constant	[U
⊞Thread 0x9feb8b2f	0.017s	1	0ms		Thread	vi
⊞Stream 0x1cfbf039	0.014s	119	0.188ms		Stream	Re
⊞Critical Section 0xacfcdca1	0.002s	1	0ms		Critical Section	vi
⊞Stream ..\dat\balls.dat 0x6e6185ca	0.002s	9	0ms		Stream	re
⊞Stream C:\Windows\Globalization\Sorting	0.001s	1	0ms		Stream	Sc
⊞Read/Write Lock 0xeb27914b	0.001s	1	0ms		Read/Write Lock	[u
⊞Critical Section 0x75327b62	0.001s	1	0ms		Critical Section	vi
⊞Stream C:\Windows\Fonts\staticcache.da	0.000s	1	0ms		Stream	In
Selected 1 row(s):	53.497s	1	0ms			

0 stack(s) selected. Viewing
Current stack is 100.0% of selection
No stack information

Timeline: 14.5s, 14.6s, 14.7s, 14.79714s, 14.9s, 15s, 15.1s, 15.2s, 15.3s, 15.4s, 15.5s, 15.6s

Threads: WinMainCRTStartup (..., Thread (0x1464), thread_video (0x1864), Thread (0x1dd8)

Thread Concurrency

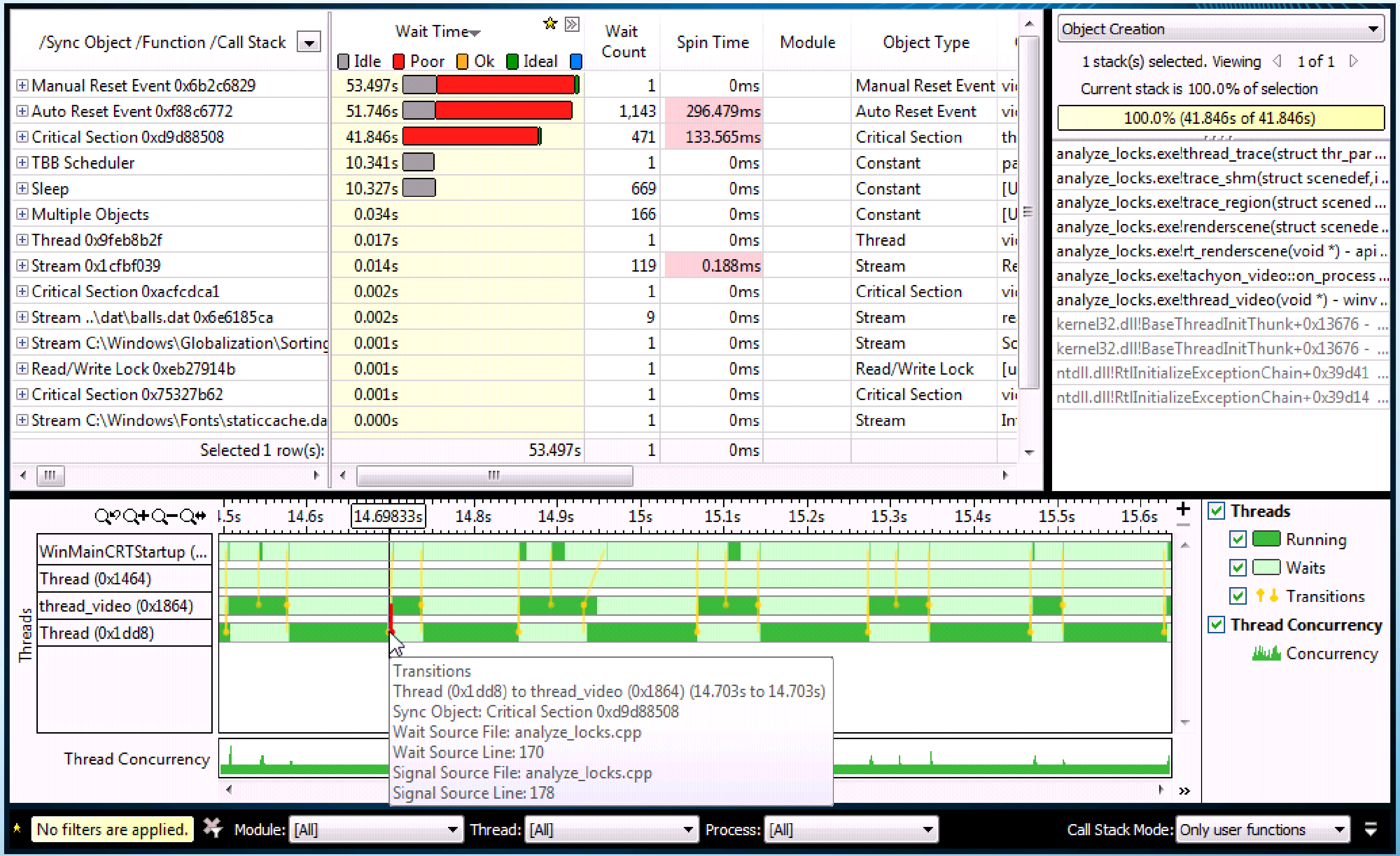
Waiting
Start: 14.740s Duration: 0.115s
Sync Object: Critical Section 0xd9d88508
Wait Source File: analyze_locks.cpp
Wait Source Line: 170
Signal Source File: analyze_locks.cpp
Signal Source Line: 178

Legend: ✓ Threads; ✓ 🟩 Running; ✓ Waits; ✓ Transitions; ✓ Thread Concurrency; Concurrency

No filters are applied. Module: [All] Thread: [All] Process: [All] Call Stack Mode: Only user functions

Timeline Visualizes Thread Behavior

- Retrieve additional information on thread transitions



/Sync Object /Function /Call Stack	Wait Time (🟥 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦)	Wait Count	Spin Time	Module	Object Type	
⊞ Manual Reset Event 0x6b2c6829	53.497s	1	0ms		Manual Reset Event	vi
⊞ Auto Reset Event 0xf88c6772	51.746s	1,143	296.479ms		Auto Reset Event	vi
⊞ Critical Section 0xd9d88508	41.846s	471	133.565ms		Critical Section	th
⊞ TBB Scheduler	10.341s	1	0ms		Constant	pa
⊞ Sleep	10.327s	669	0ms		Constant	[U
⊞ Multiple Objects	0.034s	166	0ms		Constant	[U
⊞ Thread 0x9feb8b2f	0.017s	1	0ms		Thread	vi
⊞ Stream 0x1cfbf039	0.014s	119	0.188ms		Stream	Re
⊞ Critical Section 0xacfcdca1	0.002s	1	0ms		Critical Section	vi
⊞ Stream ..\dat\balls.dat 0x6e6185ca	0.002s	9	0ms		Stream	re
⊞ Stream C:\Windows\Globalization\Sorting	0.001s	1	0ms		Stream	Sc
⊞ Read/Write Lock 0xeb27914b	0.001s	1	0ms		Read/Write Lock	[u
⊞ Critical Section 0x75327b62	0.001s	1	0ms		Critical Section	vi
⊞ Stream C:\Windows\Fonts\staticcache.da	0.000s	1	0ms		Stream	In
Selected 1 row(s):	53.497s	1	0ms			

Object Creation

1 stack(s) selected. Viewing 1 of 1

Current stack is 100.0% of selection

100.0% (41.846s of 41.846s)

```
analyze_locks.exe!thread_trace(struct thr_par...
analyze_locks.exe!trace_shm(struct scenedef,i..
analyze_locks.exe!trace_region(struct scened...
analyze_locks.exe!renderscene(struct scenede..
analyze_locks.exe!rt_renderscene(void *) - api..
analyze_locks.exe!tachyon_video::on_process...
analyze_locks.exe!thread_video(void *) - winv..
kernel32.dll!BaseThreadInitThunk+0x13676 - ...
kernel32.dll!BaseThreadInitThunk+0x13676 - ...
ntdll.dll!RtlInitializeExceptionChain+0x39d41 ...
ntdll.dll!RtlInitializeExceptionChain+0x39d14 ...
```

Transitions
Thread (0x1dd8) to thread_video (0x1864) (14.703s to 14.703s)
Sync Object: Critical Section 0xd9d88508
Wait Source File: analyze_locks.cpp
Wait Source Line: 170
Signal Source File: analyze_locks.cpp
Signal Source Line: 178

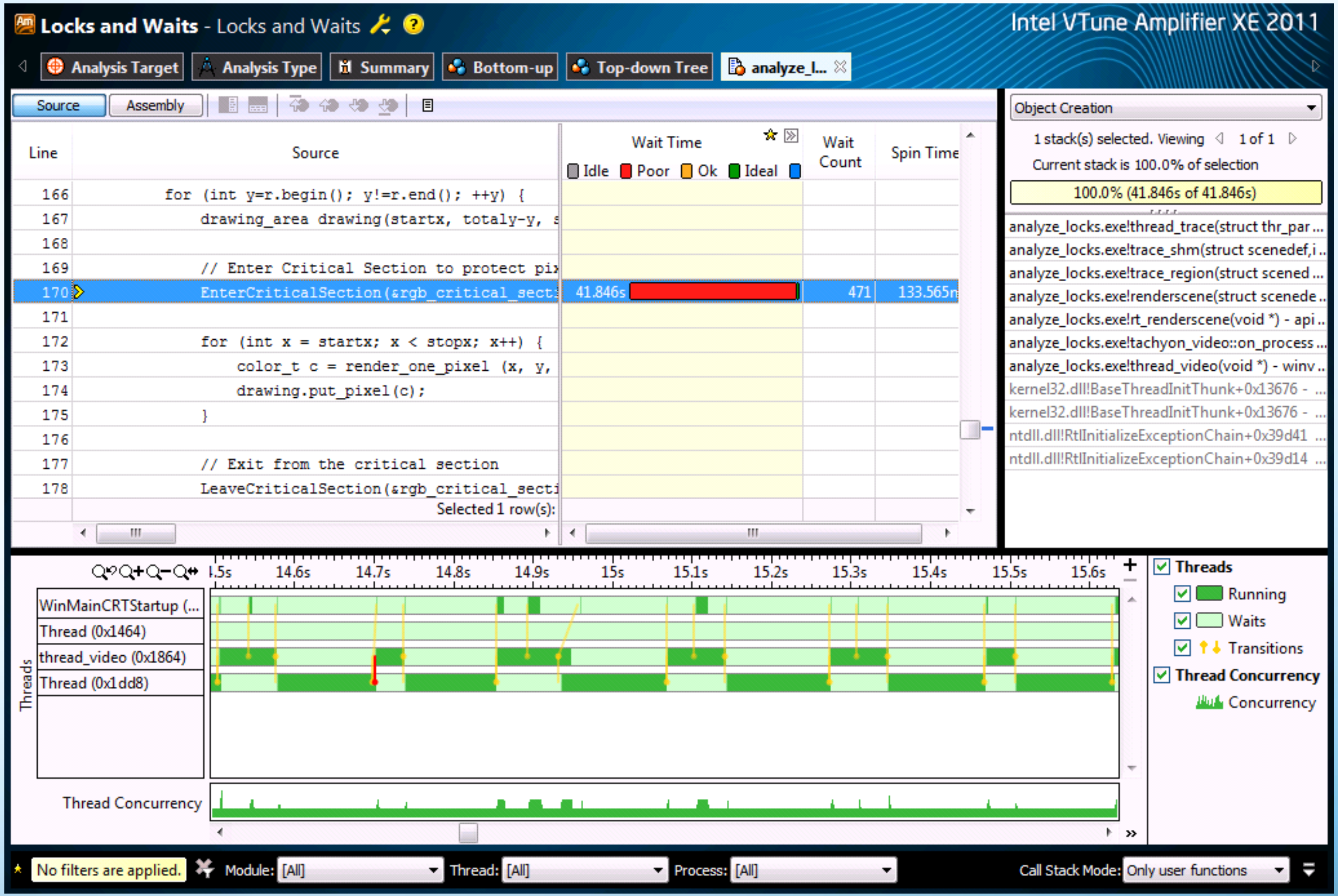
✓ Threads
✓ 🟩 Running
✓ ⬜ Waits
✓ Transitions
✓ Thread Concurrency
Concurrency

No filters are applied. Module: [All] Thread: [All] Process: [All] Call Stack Mode: Only user functions

Software & Services Group, Developer Products Division

Drilling down into Thread Behavior

- Reveal source code that causes thread transitions



Locks and Waits - Locks and Waits

Intel VTune Amplifier XE 2011

Line	Source	Wait Time (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦)	Wait Count	Spin Time
166	`for (int y=r.begin(); y!=r.end(); ++y) {`			
167	`drawing_area drawing(startx, totaly-y, s`			
168				
169	`// Enter Critical Section to protect pix`			
170	`EnterCriticalSection(&rgb_critical_secti`	41.846s	471	133.565m
171				
172	`for (int x = startx; x < stopx; x++) {`			
173	`color_t c = render_one_pixel (x, y,`			
174	`drawing.put_pixel(c);`			
175	`}`			
176				
177	`// Exit from the critical section`			
178	`LeaveCriticalSection(&rgb_critical_secti`			
	Selected 1 row(s):			

Object Creation

1 stack(s) selected. Viewing 1 of 1

Current stack is 100.0% of selection

100.0% (41.846s of 41.846s)

```
analyze_locks.exe!thread_trace(struct thr_par...
analyze_locks.exe!trace_shm(struct scenedef,i...
analyze_locks.exe!trace_region(struct scened...
analyze_locks.exe!renderscene(struct scenede...
analyze_locks.exe!rt_renderscene(void *) - api...
analyze_locks.exe!tachyon_video::on_process...
analyze_locks.exe!thread_video(void *) - winv...
kernel32.dll!BaseThreadInitThunk+0x13676 - ...
kernel32.dll!BaseThreadInitThunk+0x13676 - ...
ntdll.dll!RtlInitializeExceptionChain+0x39d41 ...
ntdll.dll!RtlInitializeExceptionChain+0x39d14 ...
```

No filters are applied. Module: [All] Thread: [All] Process: [All] Call Stack Mode: Only user functions

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

Performance Monitoring Unit (PMU)

- **Per-core PMU**:
  - Each core provides 2 programmable counters and 1 fixed counter.
  - 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

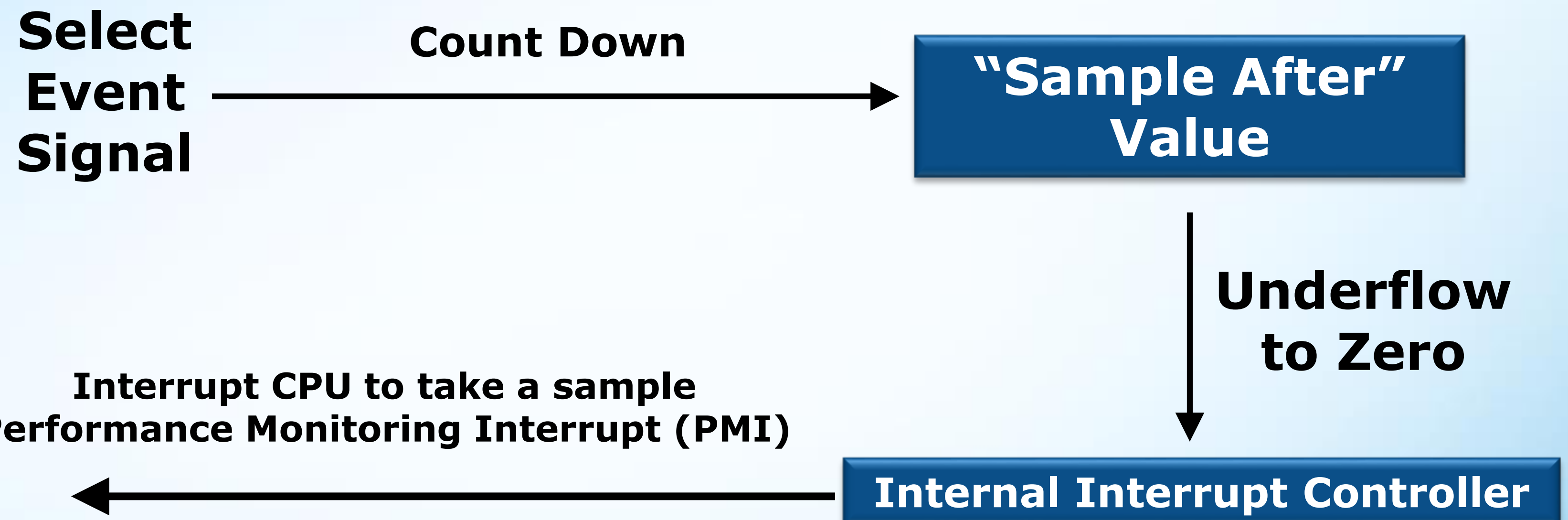
Event Based Performance Analysis

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

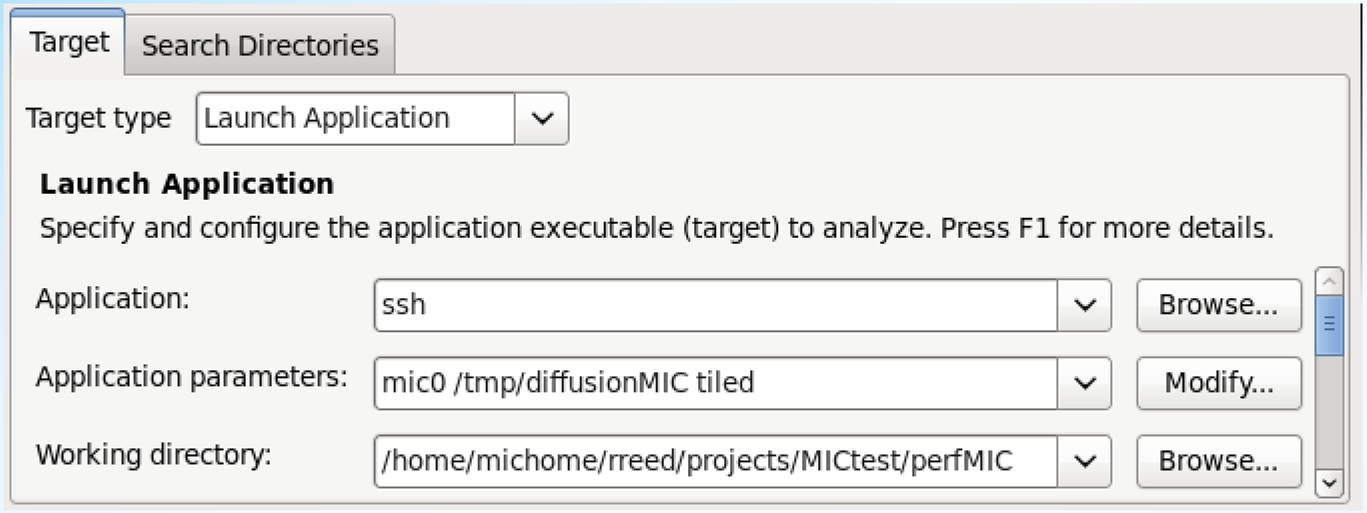
How Event Based Sampling (EBS) Works


Select Event Signal
Count Down
"Sample After" Value
Underflow to Zero
Internal Interrupt Controller
Interrupt CPU to take a sample
Performance Monitoring Interrupt (PMI)

•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



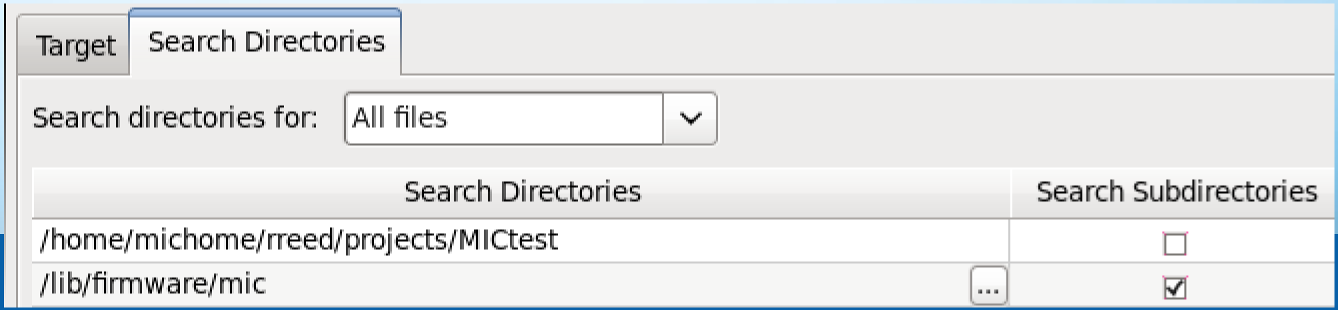
Target type: Launch Application

Launch Application

Specify and configure the application executable (target) to analyze. Press F1 for more details.

Application:	ssh
Application parameters:	mic0 /tmp/diffusionMIC tiled
Working directory:	/home/michome/rreed/projects/MICtest/perfMIC

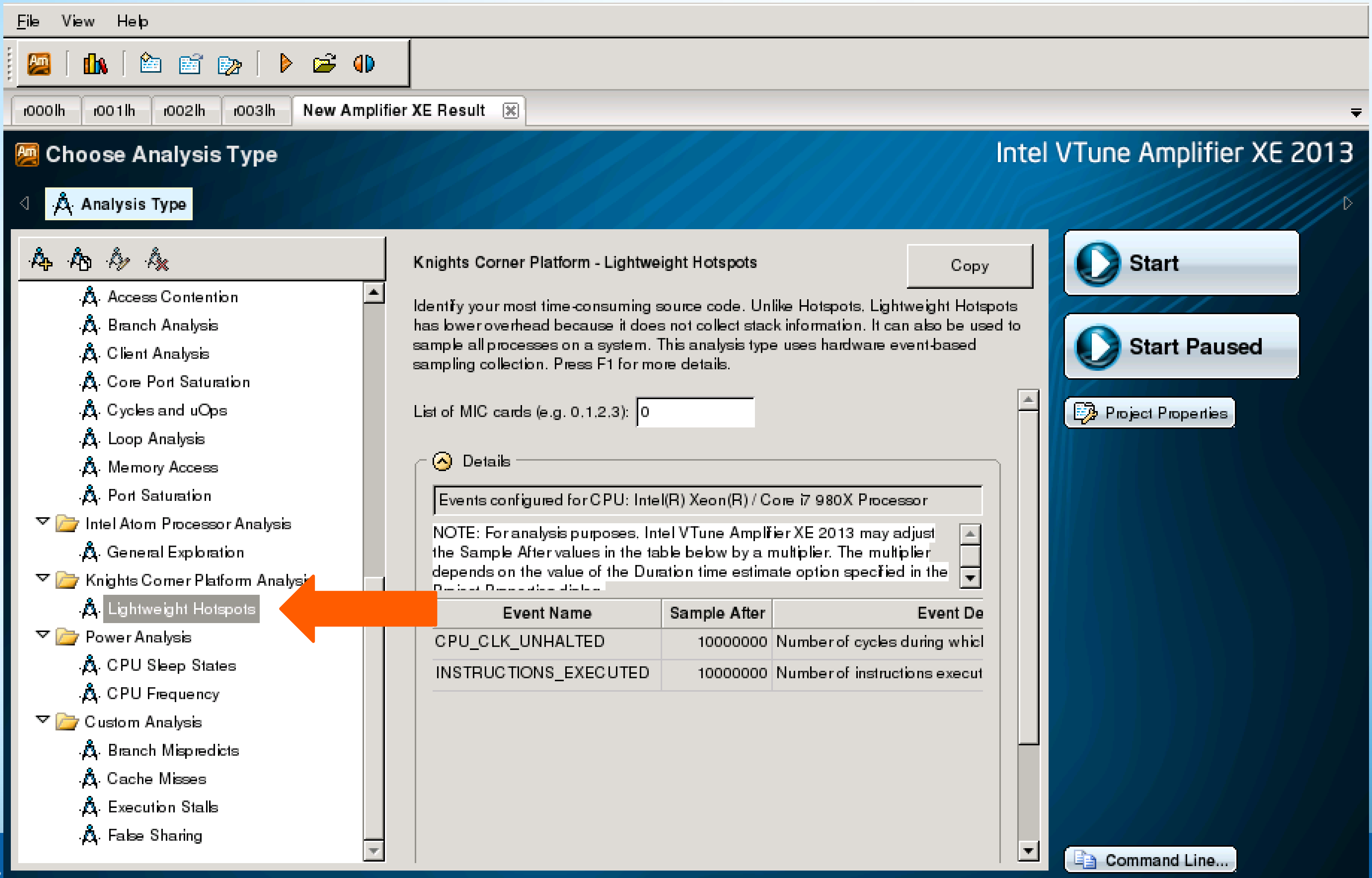
- Application settings:
  - Application: ssh
  - Parameters: mic0 "<app startup>"
  - Working directory: Usually does not matter
  - Don't forget to set search directories under "All files"



Search directories for: All files

Search Directories	Search Subdirectories
/home/michome/rreed/projects/MICtest	☐
/lib/firmware/mic	✓

Application Configuration



Knights Corner Platform - Lightweight Hotspots

Identify your most time-consuming source code. Unlike Hotspots, Lightweight Hotspots has lower overhead because it does not collect stack information. It can also be used to sample all processes on a system. This analysis type uses hardware event-based sampling collection. Press F1 for more details.

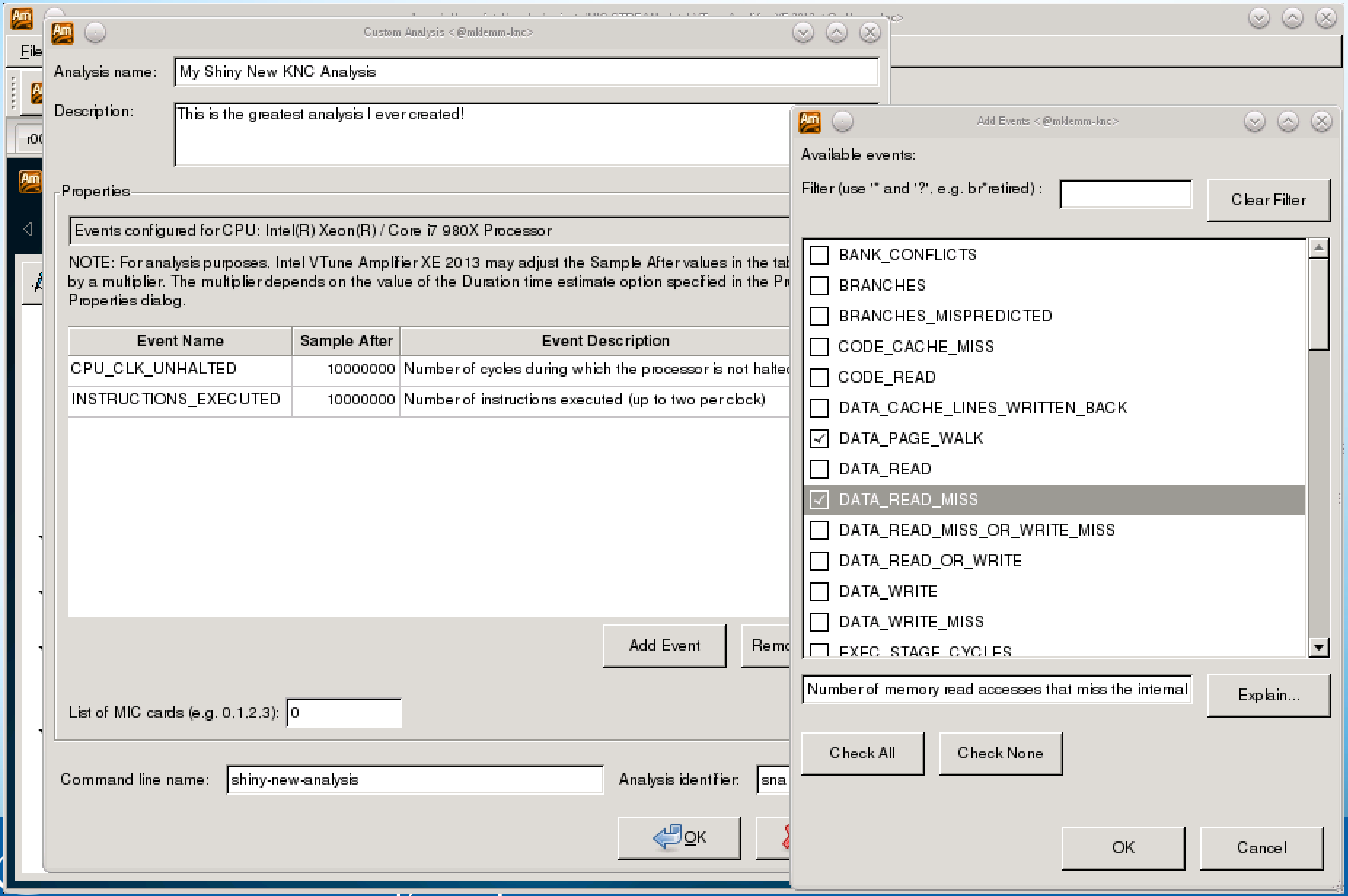
List of MIC cards (e.g. 0,1,2,3): 0

Events configured for CPU: Intel(R) Xeon(R) / Core i7 980X Processor

NOTE: For analysis purposes, Intel VTune Amplifier XE 2013 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

Event Name	Sample After	Event De
CPU_CLK_UNHALTED	10000000	Number of cycles during whicl
INSTRUCTIONS_EXECUTED	10000000	Number of instructions execut

Configuring a User-defined Analysis



Analysis name: My Shiny New KNC Analysis

Description: This is the greatest analysis I ever created!

Properties

Events configured for CPU: Intel(R) Xeon(R) / Core i7 980X Processor

NOTE: For analysis purposes, Intel VTune Amplifier XE 2013 may adjust the Sample After values in the tal by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Pr Properties dialog.

Event Name	Sample After	Event Description
CPU_CLK_UNHALTED	10000000	Number of cycles during which the processor is not halted
INSTRUCTIONS_EXECUTED	10000000	Number of instructions executed (up to two per clock)

List of MIC cards (e.g. 0,1,2,3): 0

Command line name: shiny-new-analysis

Analysis identifier: sna

Available events:

Filter (use '*' and '?', e.g. br*retired) :

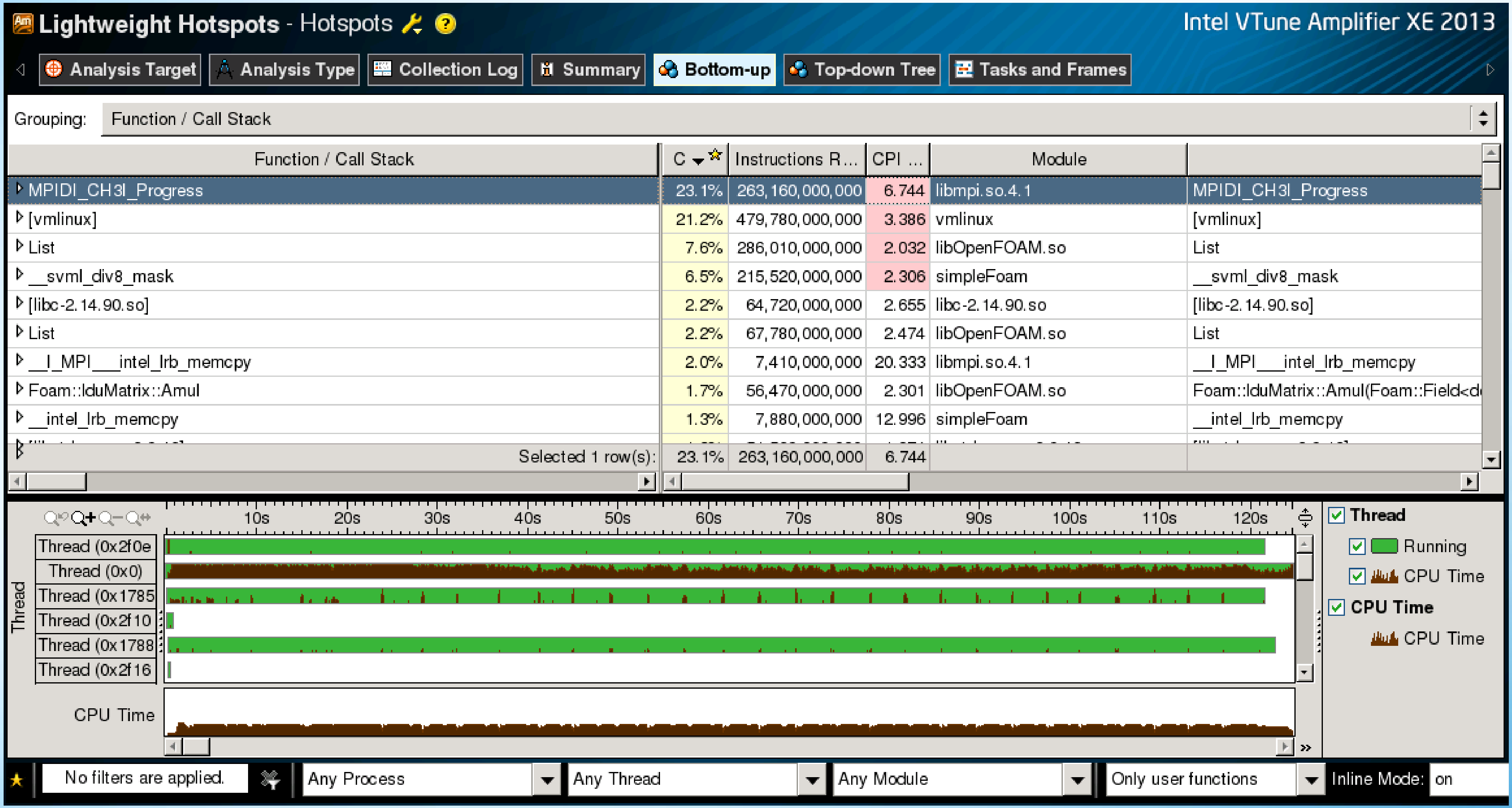
- [ ] BANK_CONFLICTS
- [ ] BRANCHES
- [ ] BRANCHES_MISPREDICTED
- [ ] CODE_CACHE_MISS
- [ ] CODE_READ
- [ ] DATA_CACHE_LINES_WRITTEN_BACK
- [x] DATA_PAGE_WALK
- [ ] DATA_READ
- [x] DATA_READ_MISS
- [ ] DATA_READ_MISS_OR_WRITE_MISS
- [ ] DATA_READ_OR_WRITE
- [ ] DATA_WRITE
- [ ] DATA_WRITE_MISS
- [ ] EXEC_STAGE_CYCLES

Number of memory read accesses that miss the internal

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



Lightweight Hotspots - Hotspots

Intel VTune Amplifier XE 2013

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

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
Selected 1 row(s):	23.1%	263,160,000,000	6.744		

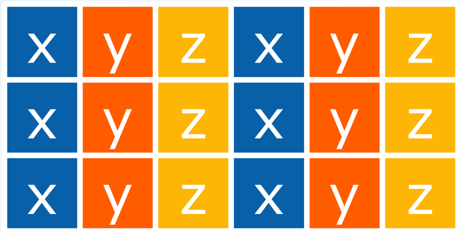
No filters are applied. | Any Process | Any Thread | Any Module | Only user functions | Inline Mode: on

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

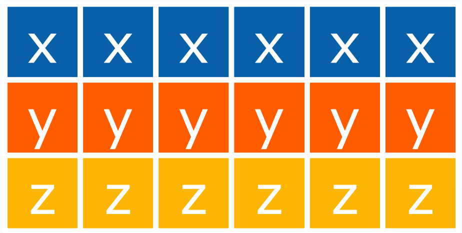
Data Layout – Common Layouts

Array-of-Structs (AoS)



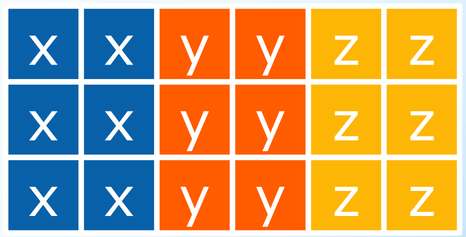
- Pros:
  Good locality of {x, y, z}.
  1 memory stream.
- Cons:
  Potential for gather/scatter.

Struct-of-Arrays (SoA)



- Pros:
  Contiguous load/store.
- Cons:
  Poor locality of {x, y, z}.
  3 memory streams.

Hybrid (AoSoA)

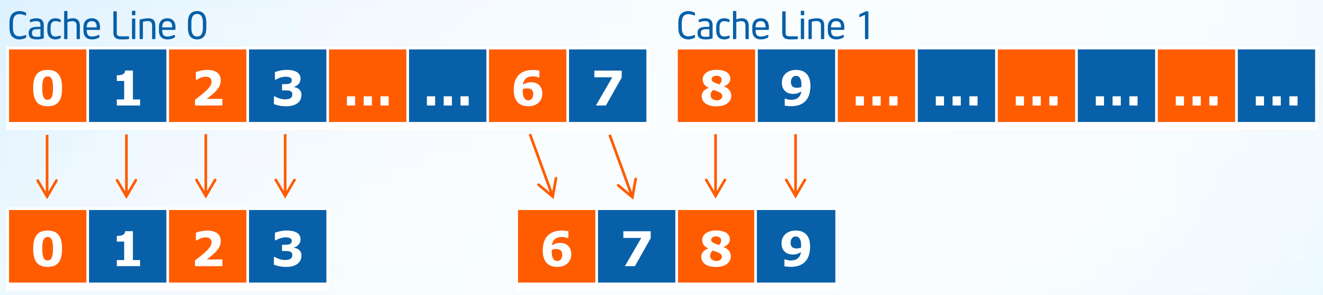


- Pros:
  Contiguous load/store.
  1 memory stream.
- Cons:
  Not a "normal" layout.

Data Layout – Why It's Important

- Instruction-Level
  - Hardware is optimized for contiguous loads/stores.
  - Support for non-contiguous accesses differs with hardware. (e.g. AVX2/KNC gather)
- Memory-Level
  - Contiguous memory accesses are cache-friendly.
  - Number of memory streams can place pressure on prefetchers.

Data Alignment – Why It's Important



Aligned Load

- Address is aligned.
- One cache line.
- One instruction.

Unaligned Load

- Address is not aligned.
- Potentially multiple cache lines.
- Potentially multiple instructions.

Data Alignment – Why It's Important

- Cache Associativity
  - L1 and L2 on Knights Corner are 8-way associative.
    - L1: 8 cache lines that are 32KB/8 = 4KB apart.
    - L2: 8 cache lines that are 512KB/8 = 64KB apart.
- Set Conflicts
  - Occur when references are a multiple of 4K (L1) or 64K (L2) apart.
  - Look for high cache miss rate, even though working set < cache capacity.
  - Solution is to pad arrays appropriately.

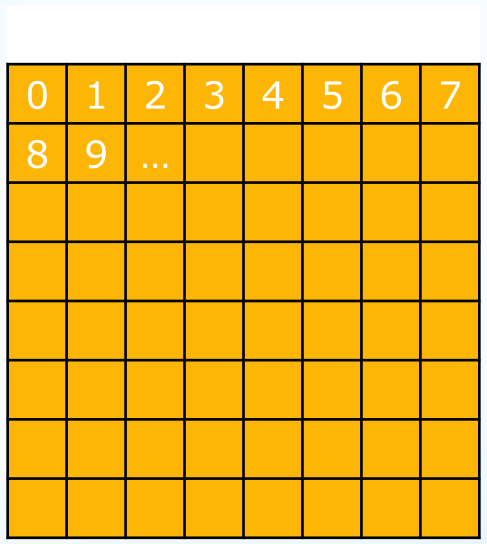
Data Alignment – Sample Applications

1) Align memory
   - _mm_malloc(bytes, 64) / !dir$ attributes align:64

2) Access Memory in an aligned way
   - for (i = 0; i < N; i++) { array[i] … }

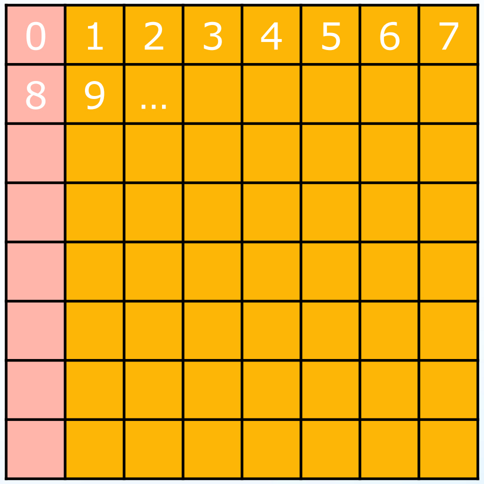
3) Tell the compiler!
   - #pragma vector aligned / !dir$ vector aligned
   - __assume_aligned(p, 16) / !dir$ assume_aligned (p, 16)
   - __assume(i % 16 == 0) / !dir$ assume (mod(i, 16) .eq. 0)

Data Alignment – Real-life Applications



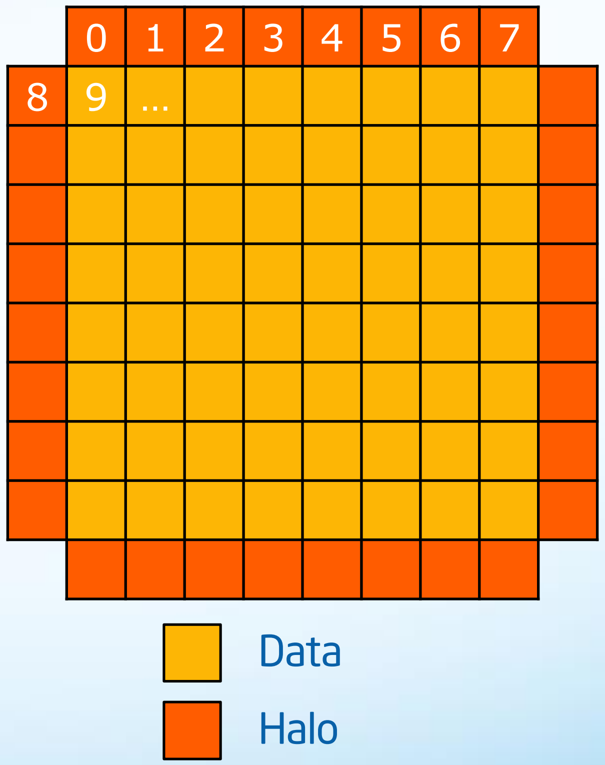
🟨 Data

Data Alignment – Real-life Applications

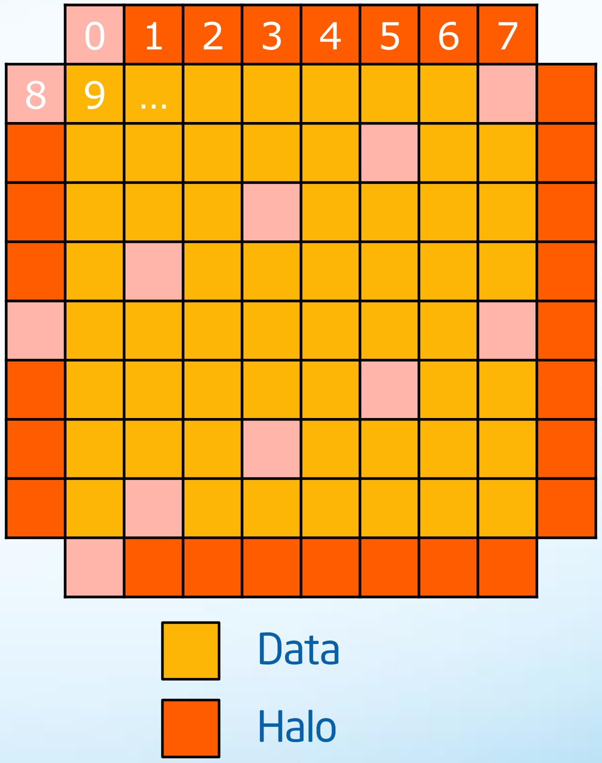


🟨 Data

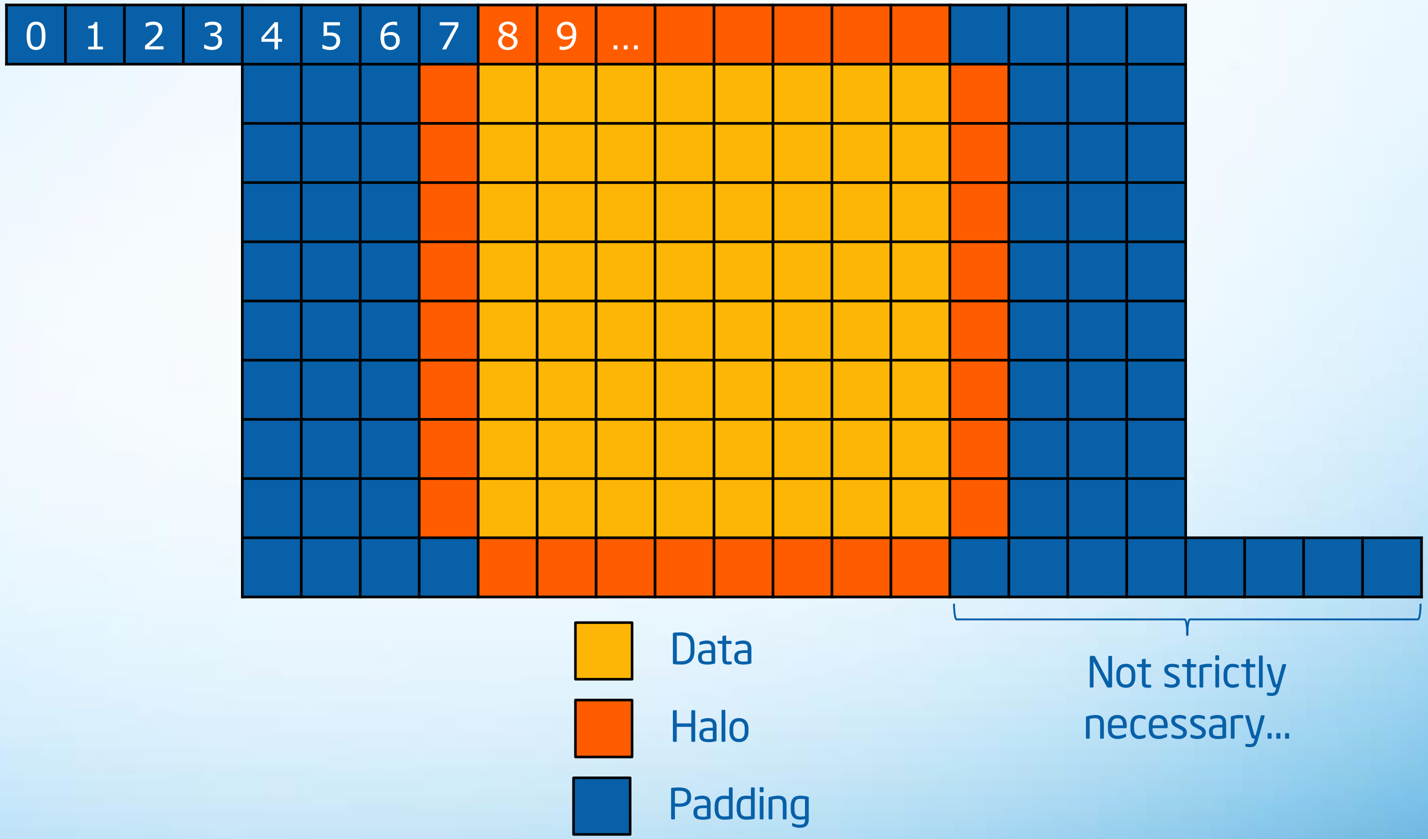
Data Alignment – Real-life Applications



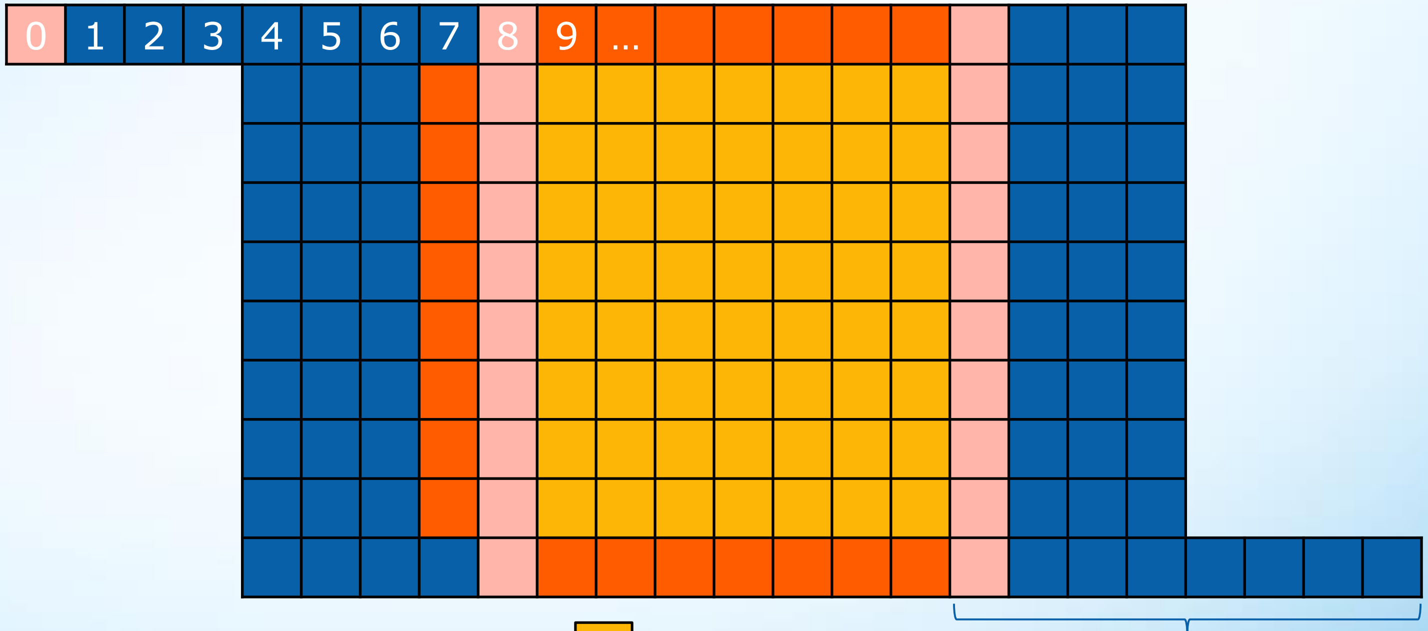
Data Alignment – Real-life Applications



Data Alignment – Real-life Applications



Data Alignment – Real-life Applications



🟨 Data
🟧 Halo
🟦 Padding

Not strictly necessary…

Example: saxpy

```
void saxpy(float *y, float *x, float a, int n) {
    int i;
    for (i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
```

The Effect of Data Alignment on saxpy

Unaligned

```
vloadunpackld (%rsi,%r14,4),%zmm1
vprefetch1 0x400(%rsi,%r14,4)
vloadunpackld 0x40(%rsi,%r14,4),%zmm2
vprefetch0 0x200(%rsi,%r14,4)
vloadunpackhd 0x40(%rsi,%r14,4),%zmm1
vprefetche1 0x400(%rdi,%r14,4)
vloadunpackhd 0x80(%rsi,%r14,4),%zmm2
vprefetch0 0x200(%rdi,%r14,4)
vfmadd213ps (%rdi,%r14,4),%zmm0,%zmm1
vprefetch1 0x440(%rsi,%r14,4)
vfmadd213ps 0x40(%rdi,%r14,4),%zmm0,%zmm2
vprefetch0 0x240(%rsi,%r14,4)
vmovaps %zmm1,(%rdi,%r14,4)
vprefetche1 0x440(%rdi,%r14,4)
vmovaps %zmm2,0x40(%rdi,%r14,4)
vprefetch0 0x240(%rdi,%r14,4)
```

Aligned

```
vmovaps (%rsi,%r9,4),%zmm2
vprefetch1 0x400(%rsi,%r9,4)
vmovaps 0x40(%rsi,%r9,4),%zmm3
vprefetch0 0x200(%rsi,%r9,4)
vfmadd213ps (%rdi,%r9,4),%zmm0,%zmm2
vprefetche1 0x400(%rdi,%r9,4)
vfmadd213ps 0x40(%rdi,%r9,4),%zmm0,%zmm3
vprefetch0 0x200(%rdi,%r9,4)
vmovaps %zmm2,(%rdi,%r9,4)
vprefetch1 0x440(%rsi,%r9,4)
vmovaps %zmm3,0x40(%rdi,%r9,4)
vprefetch0 0x240(%rsi,%r9,4)
vprefetche1 0x440(%rdi,%r9,4)
mov %al,%al
vprefetch0 0x240(%rdi,%r9,4)
```

Data Alignment – Host

- AVX instructions are less sensitive to unaligned accesses.
- Still makes sense to align data on the host
  - Cache aligned
  - 4K page aligned
- Data alignment ensures that DMA transfers work best when offloading data to the target
  - Ideal alignment: 4K pages

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

Loop Optimizations

- SIMD works best, if
  - data is properly aligned
  - data is stored in large portions of stride 1
  - loops have a high trip count
- Vectorization potential
  - Average utilization of SIMD register for a loop
  - VP = (N/vp) / ceil(N/vl)
- Vectorization potential can be measured
  - % vector instructions: VPU_INSTRUCTIONS_EXECUTED/INSTRUCTIONS_EXECUTED
  - Avg. % elements used per vector: VPU_ELEMENTS_ACTIVE/INSTRUCTIONS_EXECUTED

Example Loop: NWChem

```
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h7=1,h7d
do h2=1,h2d
do h3=1,h3d
triplesx(h3,h2,h1,p6,p5,p4) =
  triplesx(h3,h2,h1,p6,p5,p4) -
    t2sub(h7,p4,p5,h1)*
      v2sub(h3,h2,p6,h7)
end do
end do
end do
end do
end do
end do
end do
```

- Each loop runs on the order of 10 to 25 iterations
- Example: trip count 19
- Vectorization potential
  - SIMD length: 8
  - 2 registers to cover 16 iterations
  - remainder loop: 3 trips

80%

Example Loop: NWChem - Optimized

```
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h7=1,h7d
do h3h2=1,h2d*h3d
triplesx(h3h2,h1,p6,p5,p4) =
  triplesx(h3h2,h1,p6,p5,p4) -
    t2sub(h7,p4,p5,h1)*
      v2sub(h3h2,p6,h7)
end do
end do
end do
end do
end do
end do
```

- Fused loop has about 1 or 2 orders of magnitudes more trips
- Example: trip count 19
- Vectorization potential
  - SIMD length: 8
  - 45 registers to cover 360 iterations
  - remainder loop: 1 trips

98%

Agenda

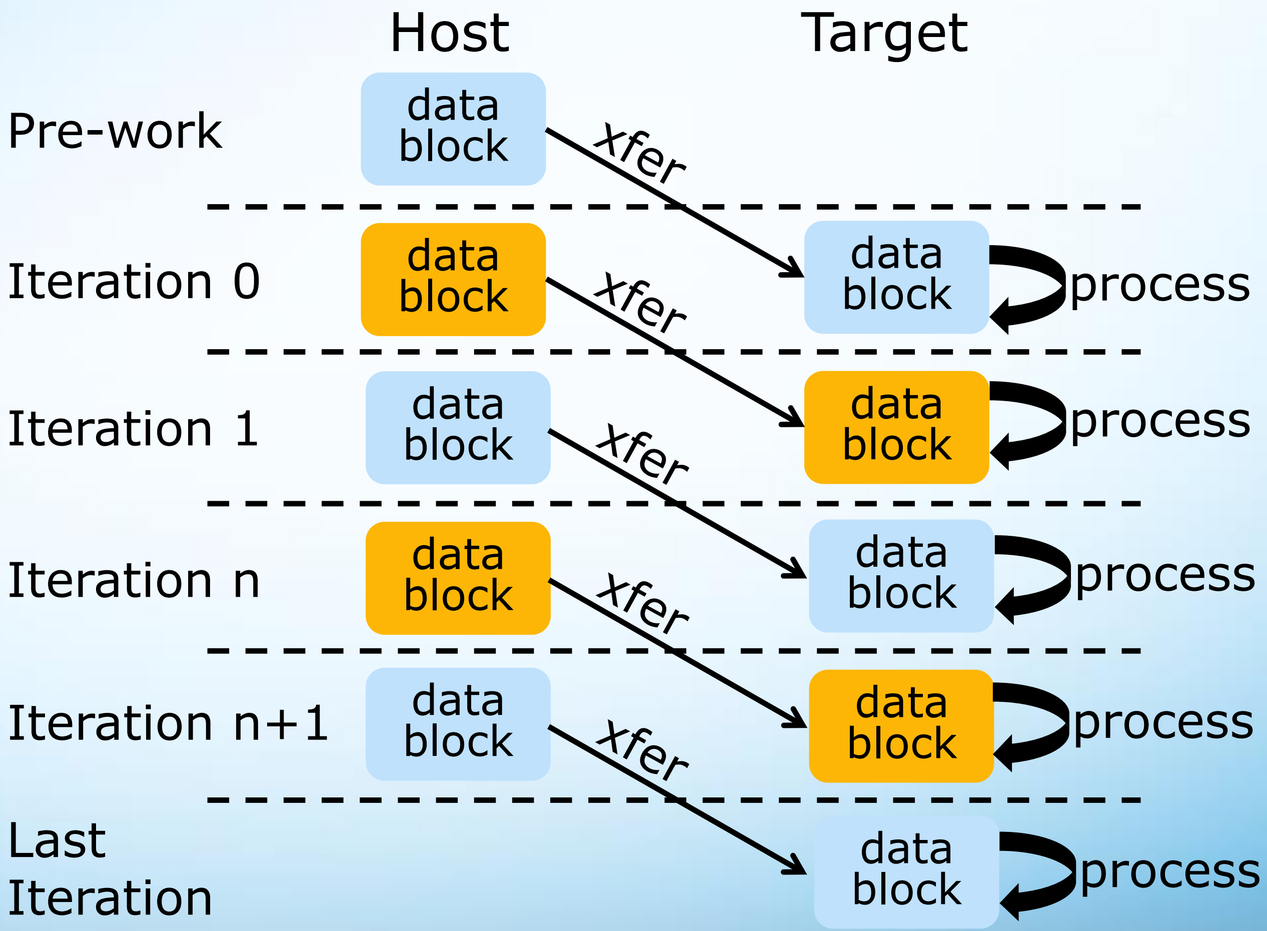
- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

Optimizing Data Transfers

- PCIe is a bottleneck compare to memory
  - High latency (on the order of microseconds)
  - Low bandwidth (1 to 2 orders of magnitude less than GDDR)
- Avoid data transfers as much as possible
- Overlap data transfers and computation where possible
- Stay on the target device as long as possible

Double Buffering Example

- Overlap computation and communication
- Generalizes to data domain decomposition



Double Buffering Example - Main

```
int main(int argc, char* argv[])   {
  int i;
  double st_time, end_time;
  double sync_tm, async_in_tm;
  //  Allocate & initialize in1, res1,
  // in2, res2 on the host

 #pragma offload_transfer target(mic:0) in(cnt)  \
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(1) free_if(0) )

  do_async_in();

  // Validate results and print timings

 #pragma offload_transfer target(mic:0) \
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(0) free_if(1) )
```

Allocate arrays on target: alloc_if(1)

Retain for duration of sample: free_if(0)

in1 and in2 represent 2 separate buffers.

Free target allocations: free_if(1)

Double Buffering – do_asynch_in, evens

```
void do_async_in() {
  float lsum;
  int i;
  lsum = 0.0f;

 #pragma offload_transfer target(mic:0)   \
        in(in1 : length(cnt) alloc_if(0) free_if(0) ) signal(in1)

  for (i=0; i < iter; i++) {
    if (i%2 == 0) {

     #pragma offload_transfer target(mic:0) if(i!=iter-1)  \
            in(in2 : length(cnt) alloc_if(0) free_if(0) ) \
            signal(in2)

     #pragma offload target(mic:0) nocopy(in1) wait(in1)  \
            out(res1 : length(cnt) alloc_if(0) free_if(0) )
     {
       compute(in1, res1);
     }

     lsum = lsum + sum_array(res1);
    }
```

Begin an initial transfer of the first dataset for the target to work on. Transfer begins immediately, is non-blocking, and will signal when complete.

For even loop iterations (except the final), first start another non-blocking transfer of the next dataset.

While that transfer progresses, process the previous dataset through the compute() function, first waiting for its transfer to complete, then return a result. Execution on the host waits for the function to return.

Double Buffering – do_asynch_in, odds

```
      else {
        #pragma offload_transfer target(mic:0) if(i!=iter-1) \
                in(in1 : length(cnt) alloc_if(0) free_if(0) ) \
                signal(in1)


         #pragma offload target(mic:0) nocopy(in2) wait(in2) \
                out(res2 : length(cnt) alloc_if(0) free_if(0) )
         {
           compute13(in2, res2);
         }

         lsum = lsum + sum_array(res2);
      }
    }
    async_in_sum = lsum / (float) iter;
}
```

For odd iterations (except the final), work on the other buffer. Start another non-blocking transfer.

Offload the compute function, and host waits for the result. Repeat, alternating between the buffers (in1 & in2).

Agenda

- Performance Tuning Methodology
- Intel® VTune™ Amplifier XE
  - Fundamental Analysis: Hotspots
  - Finding Issues in Parallel Applications
  - Using the Performance Monitoring Unit
- Optimization Strategies
  - Data Layout and Alignment
  - Vectorization Potential
  - Optimizing Data Transfers
  - Miscellaneous Tuning Hints

Miscellaneous Tuning Hints

- Check if data transfer is necessary
  - Keep data as long on the device as possible
  - Avoid transferring scratch pad memory

- Use 2MB pages for offload data transfers
  - (May) result in less page faults
  - (May) improve utilization of the TLB cache
  - (May) increase time needed to initialize pages

- Controlled by environment variable:

```
export MIC_USE_2MB_BUFFERS=16k
```

Miscellaneous Tuning Hints

- Use 2MB pages for dynamic memory
- Reserve 2MB pages in the kernel:
  - Remember this leaves less memory available for other programs!
  - Example (reserves 128 2MB pages):

```
echo “128” > /proc/sys/vm/nr_hugepages
```

- Use mmap to request allocation of 2MB pages:

```
#include <sys/mman.h>
void *p;
size_t s = 64*2*1024*1024; // 128MB
…
p = mmap(0, s, PROT_READ|PROT_WRITE,
      MAP_ANONYMOUS|MAP_PRIVATE|MAP_HUGETLB, -1, 0);
if (p == MAP_FAILED)
   perror(“mmap failed”);
```

Miscellaneous Tuning Hints

- Always use thread affinity on the coprocessor
  - Avoid expensive thread migration between cores
  - Example:

    ```
    export MIC_ENV_PREFIX=MIC
    export MIC_KMP_AFFINITY=compact / scatter / balanced
    ```

  - Affinity policies:

– compact:	keep threads close
– scatter:	distribute threads as far apart as poissible
– balanced:	create *compact* groups and *scatter* these

  - Simpler approach:

    ```
    export KMP_PLACE_THREADS=60c,3t,0o
    ```

Miscellaneous Tuning Hints

- Do not overload the coprocessor
  - Multiple offloads from different host MPI ranks
  - Multiple offloads from different host threads
- Partition the device:
  - Rank 0: export MIC_PLACE_THREADS=30c,3t,0o
  - Rank 1: export MIC_PLACE_THREADS=30c,3t,30o
- May help to improve relative performance
  - Two offloads will half the compute load each might be more beneficial
  - Less synchronization / threading overhead

Summary

- Use profiling tools to find hotspots and offload candidates
- Use a performance analysis tool to find threading issues
- Optimize offloaded code
  - Data layout and alignment
  - Vectorization potential