Evaluating program correctness and performance with new software tools from Intel

Andrzej Nowak, CERN openlab

March 18th 2011

CERN IT Technical Forum

Agenda

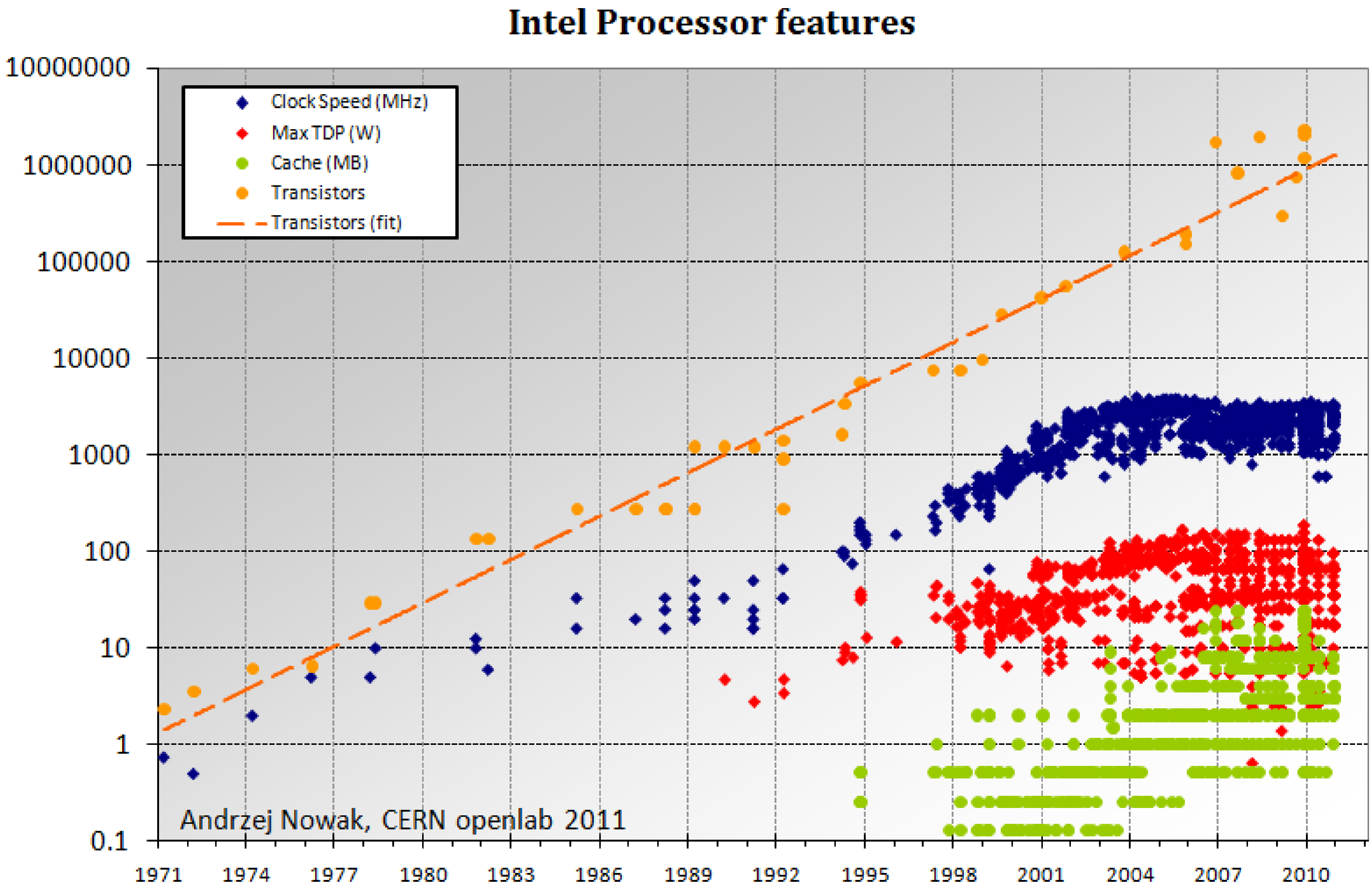
> **An introduction to the new generation of software tools from Intel**

> **Intel VTune Amplifier XE 2011 - overview**
- Description
- Features

> **Intel Inspector XE 2011 - overview**
- Description
- Features

> **API**
- Organizing data

This presentation contains some material from the Intel tools documentation


Intel Processor features
Clock Speed (MHz)
Max TDP (W)
Cache (MB)
Transistors
Transistors (fit)
10000000
1000000
100000
10000
1000
100
10
1
0.1
1971
1974
1977
1980
1983
1986
1989
1992
1995
1998
2001
2004
2007
2010
Andrzej Nowak, CERN openlab 2011

The case for optimization

> Limited scaling in hardware

- Some important CPU features that we used to rely on do not scale or even regress: frequency, cache, bus, internal buffers, ILP
- Other features (that we typically don't exploit, but we should) still scale to an extent: the number of cores, hardware threads, vectors

> Software complexity is growing rapidly

> Hence our interest in performance tuning

- As Intel puts it: "What in the world is happening to my computer?"
- What should be true, but rarely is:
  - Optimization is an integral part of the software development process
  - Performance is a feature

Intel software tools

> **Designed to aid with developing software on Intel x86 processors**

> **Previous generation:**
- Linux undermaintained: a lot of functionality missing from the Linux versions
- Tools:
  - VTune and Thread Profiler – performance tuning
  - Thread Checker – threading correctness
  - PTU 3.x (“Performance tuning utility”)

> **Current (new) generation:**
- Redesigned interfaces, new functionality
  - Unified functionality across Windows and Linux
- Much better software support (that means CERN software too)
- CERN openlab participates intensively in Alpha and Beta programs
- Tools:
  - VTune Amplifier – performance and profiling
  - Inspector – threading and memory correctness
  - PTU 4.x (experimental/expert – not our focus today)

CERN openlab participation

- **CERN openlab participated intensively in the Alpha and Beta phases of the XE tools**
  - Evaluations with CERN software – several “showstopping” bugs discovered and fixed, enabling work and avoiding long delays
  - Enhancement proposals and feature requests (dozens made)
  - Bugreports (dozens filed)
- **Cross-departmental collaborations based on Intel PTU driven by David Levinthal (Intel)**
- **Special workshops held for advanced programmers**
  - Featured lectures by engineers from Intel working on the tools
- **Regular openlab workshops now promote these new tools as well (4 in a year)**
  - Featuring demos and exercises with both open-source and Intel tools

Package components (both tools)

> **Graphical interface**
- Based on wxWidgets
- Works in Linux as well as Windows

> **Command line interface**
- Full collection capabilities
- Limited reporting capabilities

> **Tool API and libraries**
- Available for program instrumentation

VTune Amplifier

Monitoring and tweaking performance

Rationale

> **Performance tuning is increasingly growing in importance**

> **PC tuning was missing a comprehensive product which supported:**
- PMU based monitoring
- Instrumented monitoring
- Multi-threading and multi-core environments
- Graphical interpretation of results

> **Intel VTune was a step in that direction, later with a "Thread Profiler" addon**

> **Amplifier is VTune's spiritual successor, borrowing features from the experimental Intel Performance Tuning Utility (PTU)**

Functionality

> **A performance tuning tool, adapted to multi-threaded programs**

> **Two main modes**

- **User-mode sampling and tracing** – instrumented; may have a heavy impact on runtime, a lot of data collected (including stack data)
- **Hardware event-based sampling** – virtually no impact on runtime, good for hotspots and hardware utilization measurements
  - The widely covered perfmon2 does the same thing, but this tool has much better visualization capabilities

> **Operating systems supported (same functionality):**

- Linux
- Windows

Issue detection capacity

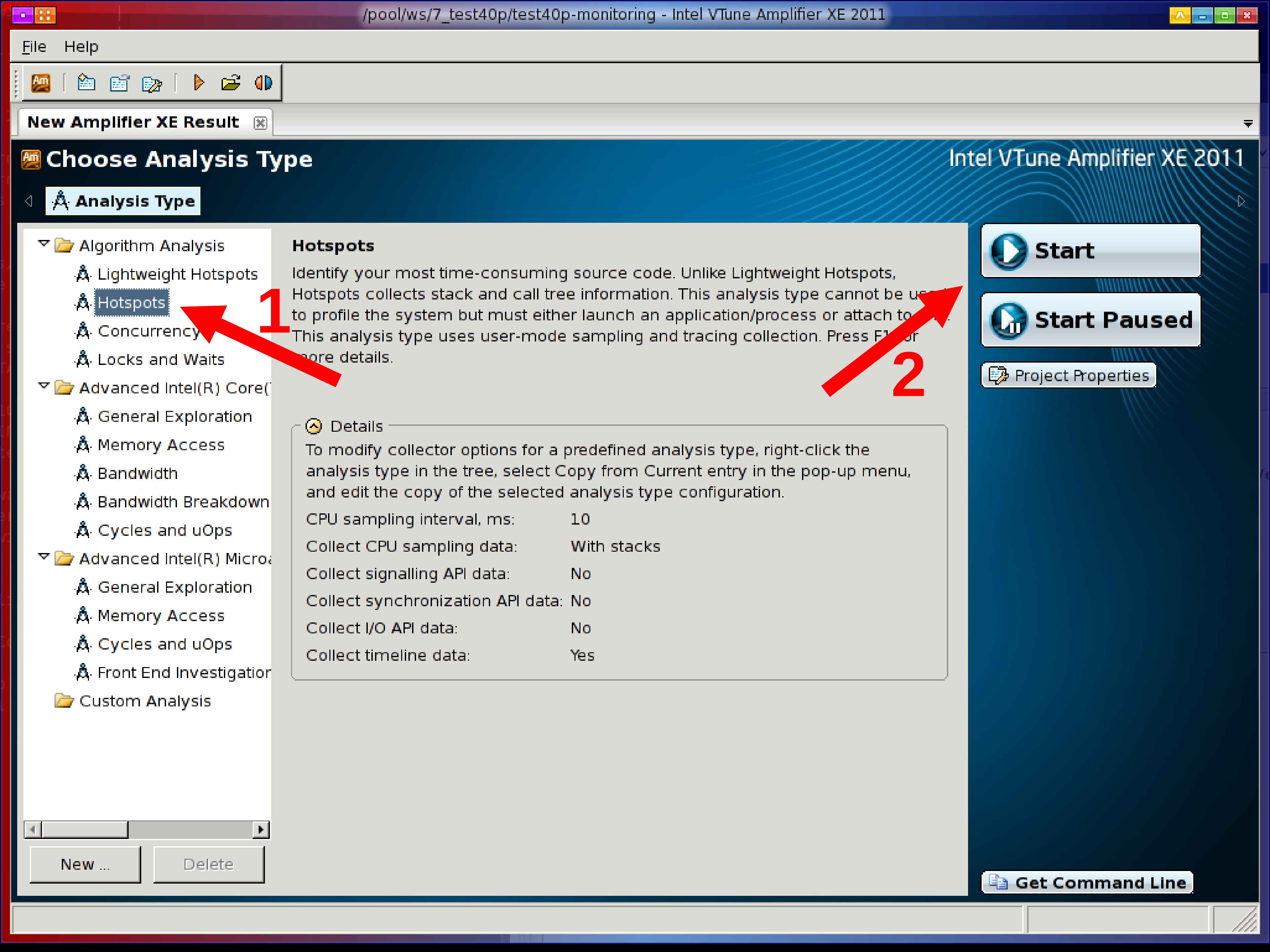
- Identify the most **time-consuming (hot) functions** in your application and/or on the whole system
- Locate sections of code that do not **effectively utilize available processor time**
- Determine the best sections of code to optimize for sequential performance and for threaded performance
- **Locate synchronization objects** that affect the application performance
- Find whether, where, and why your application spends time on **input/output operations**
- Identify and **compare** the performance impact of different synchronization methods, different numbers of threads, or different algorithms
- **Analyze thread activity and transitions**
- Identify **hardware-related bottlenecks** in your code

Select features

- **Analysis tree**: Use the performance analysis tree to choose and configure the type of analysis for your target.
- **Start data collection paused**: Click the **Start Paused** button on the command bar to start collecting performance data after a delay.
- **Viewpoints**: Choose among preset configurations of windows and panes available for the analysis result. This helps focus on particular performance problems.
- **Top-down tree**: Use to understand which flow in your application is more performance-critical.
- **Timeline analysis**: Analyze the thread activity and transitions between threads.
- **Grouping**: Group your data in different ways in the Bottom-up window to analyze the problem from different angles.
- **Source analysis**: View source with the performance data attributed to source lines to understand a possible cause of an issue.
- **Comparison analysis**: Compare performance analysis results for several application runs to estimate the performance gain you got after optimization.

An example from the HEP world

- **Based on the multi-threaded Geant 4 prototype with the FullCMS simulation example**
  - A multi-threaded simulation of the passage of particles through the CMS detector
- **Light instrumentation discussed (~10 lines inserted in total)**

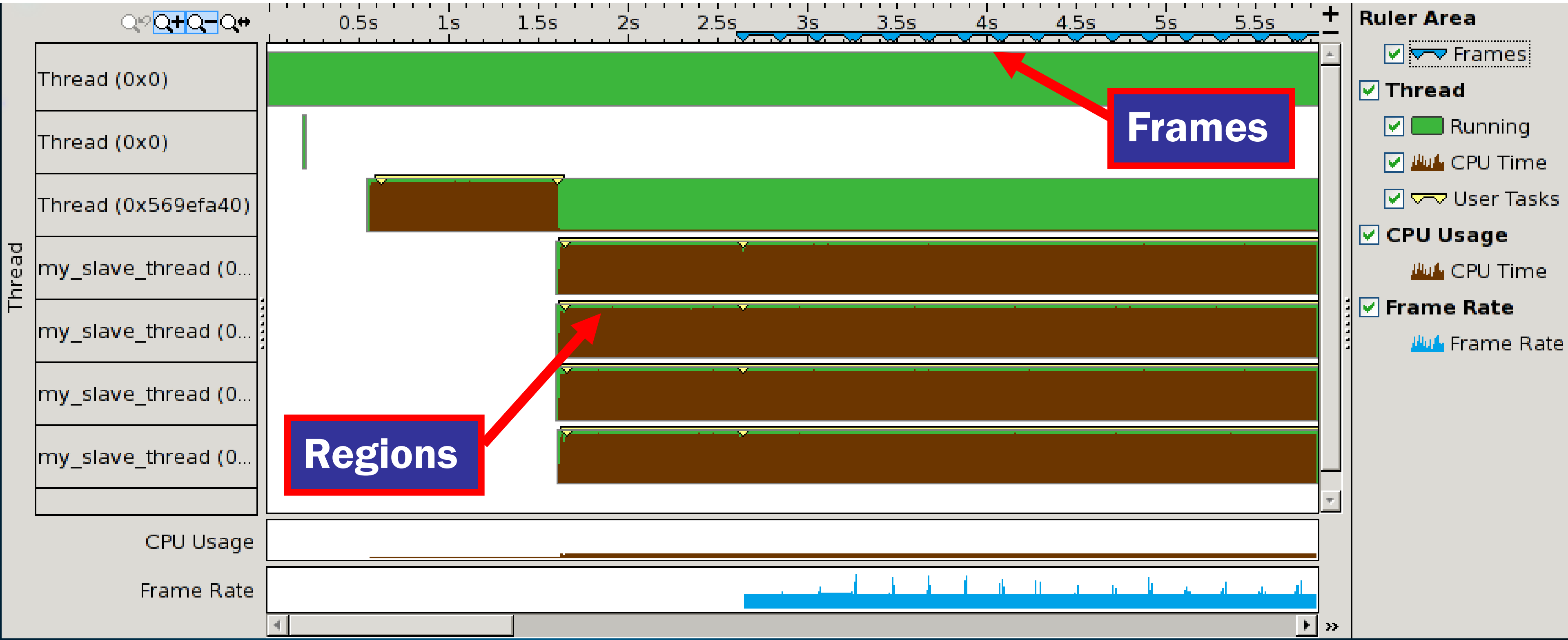

/pool/ws/7_test40p/test40p-monitoring - Intel VTune Amplifier XE 2011
File
Help
New Amplifier XE Result
Choose Analysis Type
Intel VTune Amplifier XE 2011
Analysis Type
Algorithm Analysis
Lightweight Hotspots
Hotspots
Concurrency
Locks and Waits
Advanced Intel(R) Core(
General Exploration
Memory Access
Bandwidth
Bandwidth Breakdown
Cycles and uOps
Advanced Intel(R) Micro
General Exploration
Memory Access
Cycles and uOps
Front End Investigatior
Custom Analysis
New ...
Delete
Hotspots
Identify your most time-consuming source code. Unlike Lightweight Hotspots,
Hotspots collects stack and call tree information. This analysis type cannot be
to profile the system but must either launch an application/process or attach to
This analysis type uses user-mode sampling and tracing collection. Press
details.
Details
To modify collector options for a predefined analysis type, right-click the analysis type in the tree, select Copy from Current entry in the pop-up menu, and edit the copy of the selected analysis type configuration.
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: Yes
Start
Start Paused
Project Properties
Get Command Line
1
2

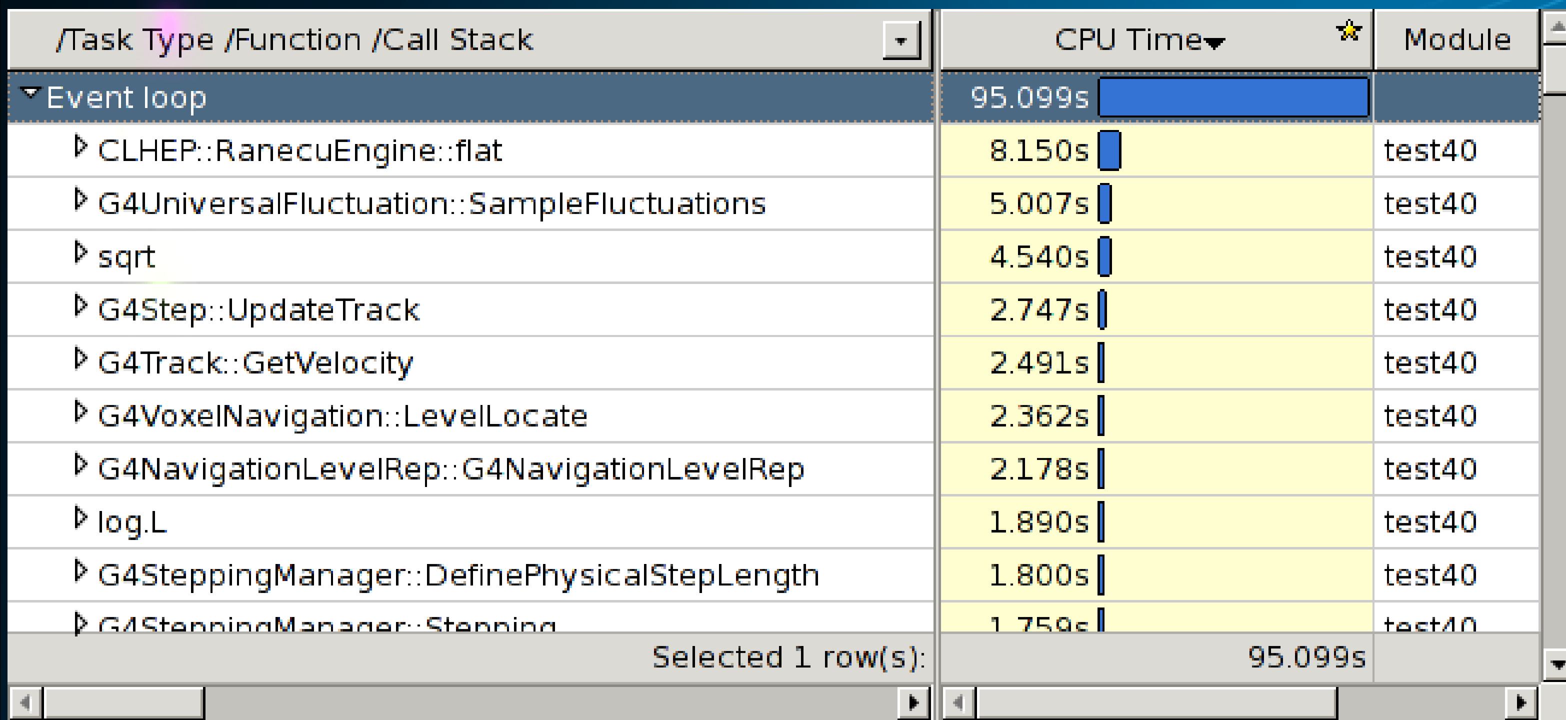
Timeline view

> **Blue elements are frames (events)**
  - as defined by instrumenting the event loop in the simulation

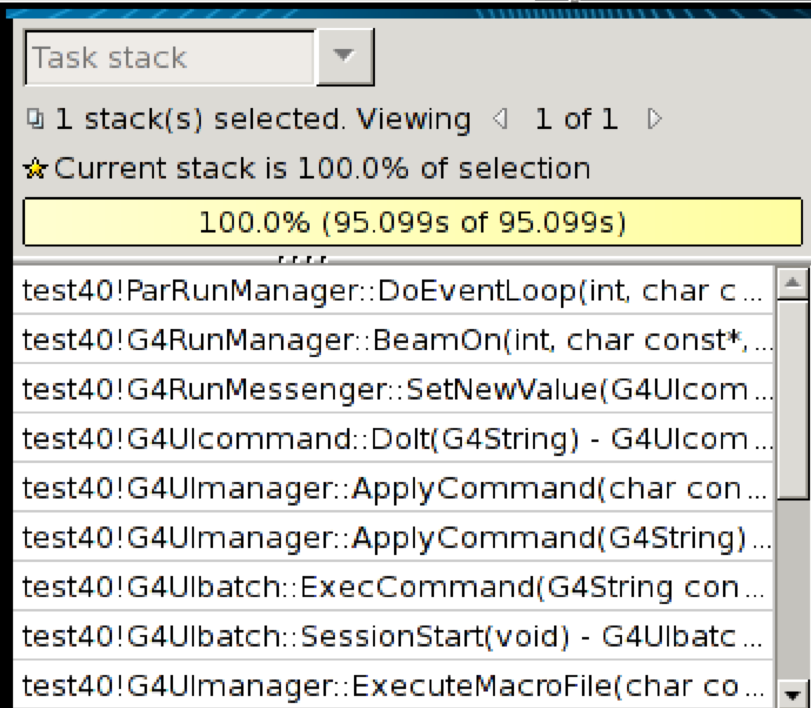
> **Yellow elements are tasks (regions)**
  - As defined by instrumenting the particular regions of the code

> **Green is runtime, brown is CPU usage**
  - Measured by the tool





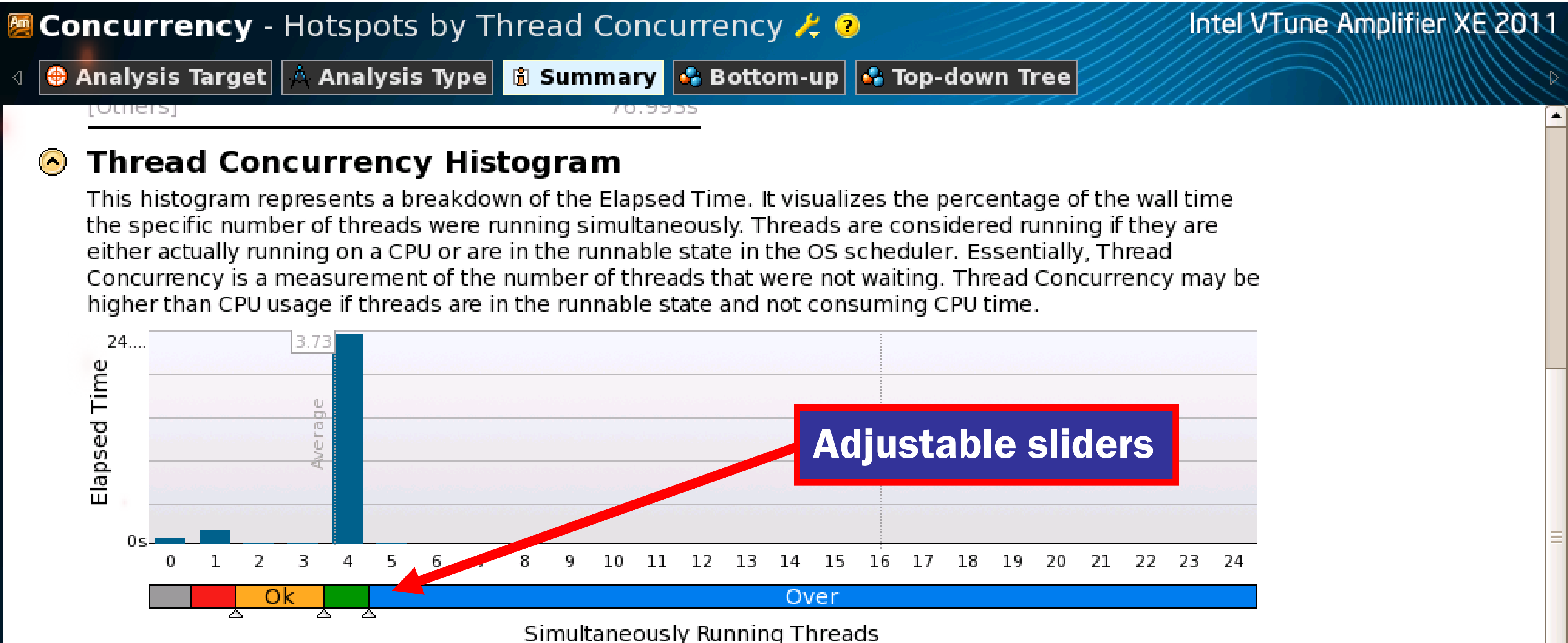
Interactive profile display



Call stack

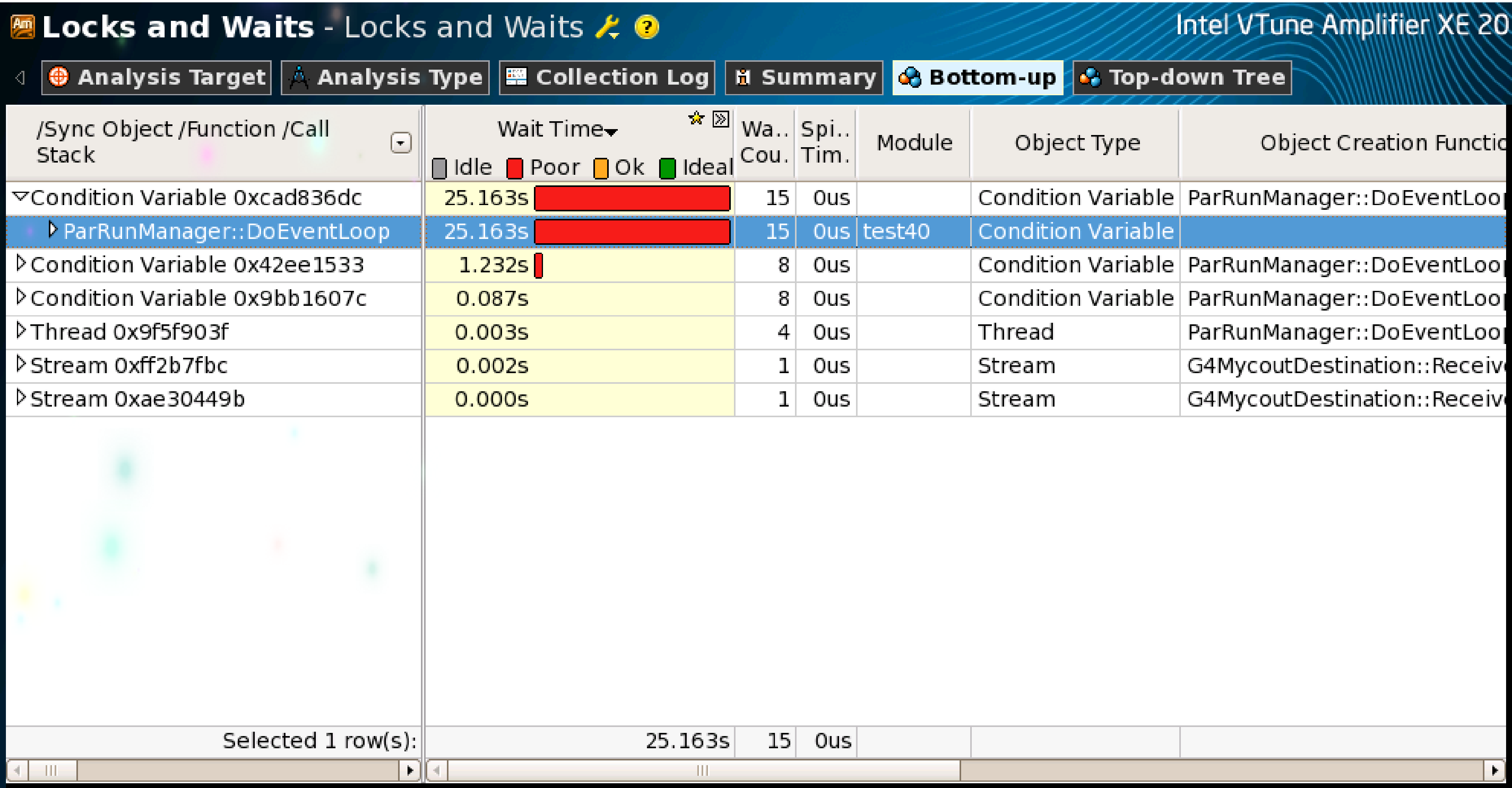
Concurrency histogram

> **Shows a histogram of elapsed time according to thread concurrency**
>
> - The user may adjust the values as he sees fit – other views will adjust the colors accordingly



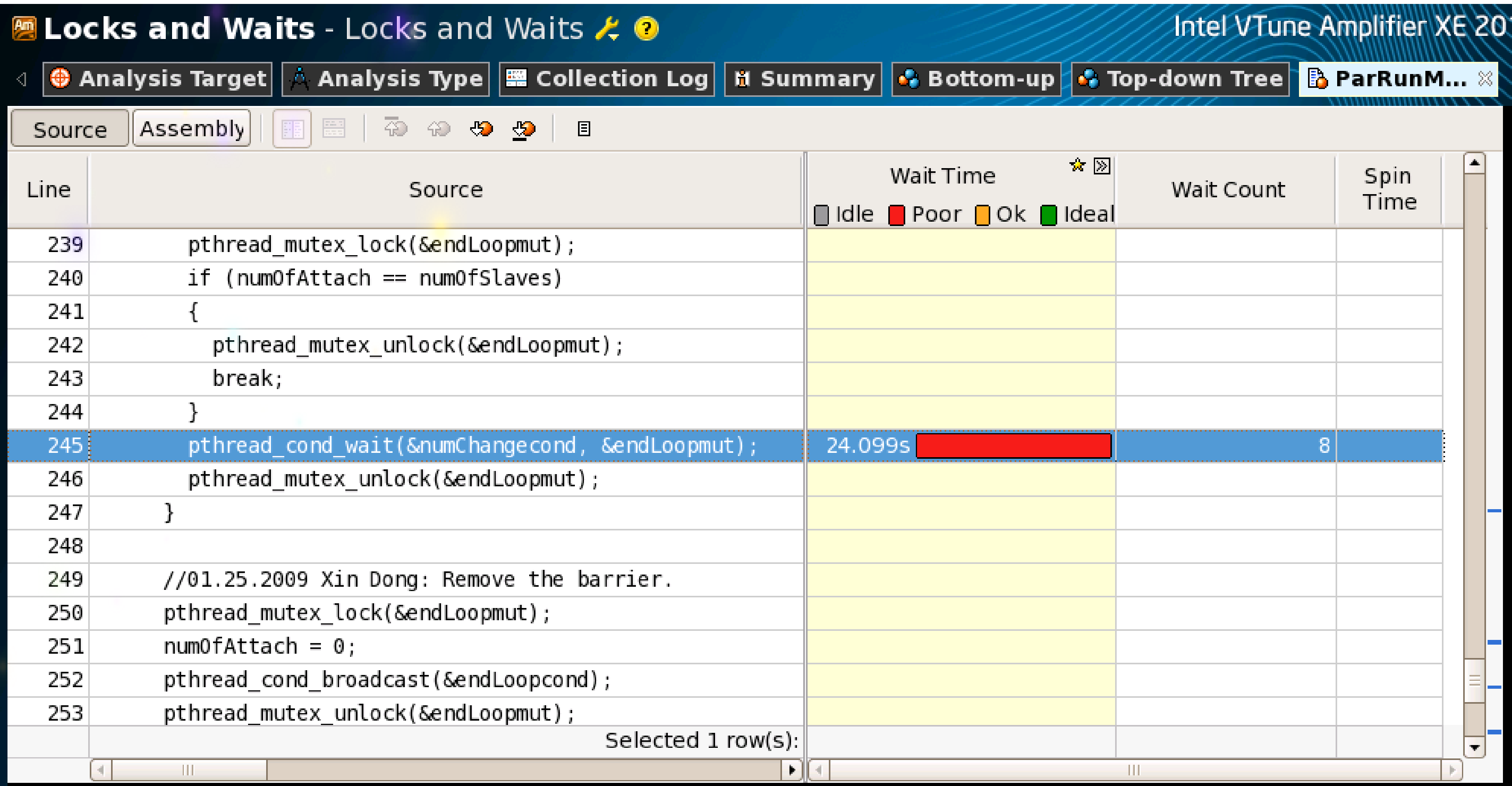
Locks and waits analysis (1)

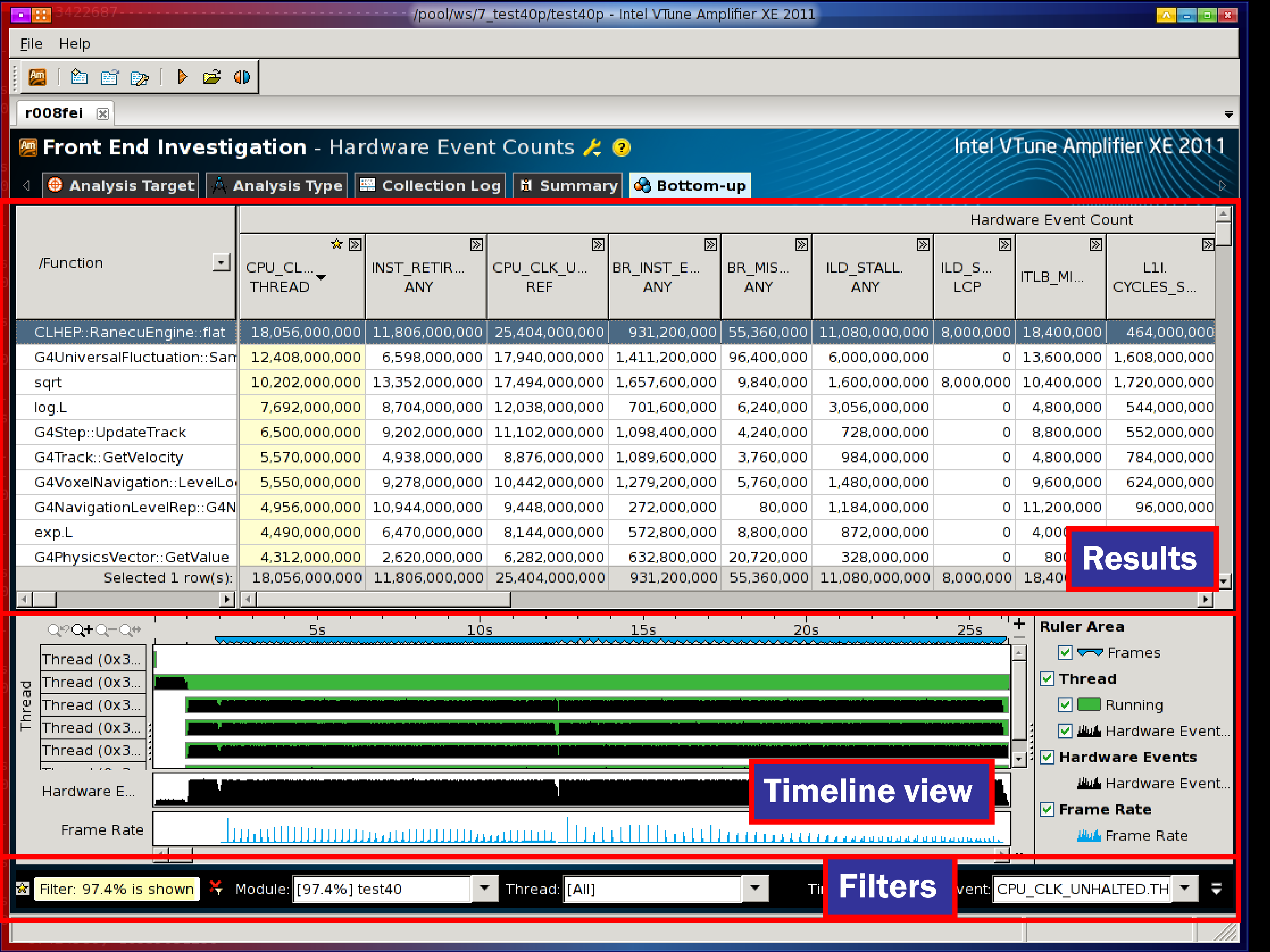
> Shows time spent in locks and synchronization objects

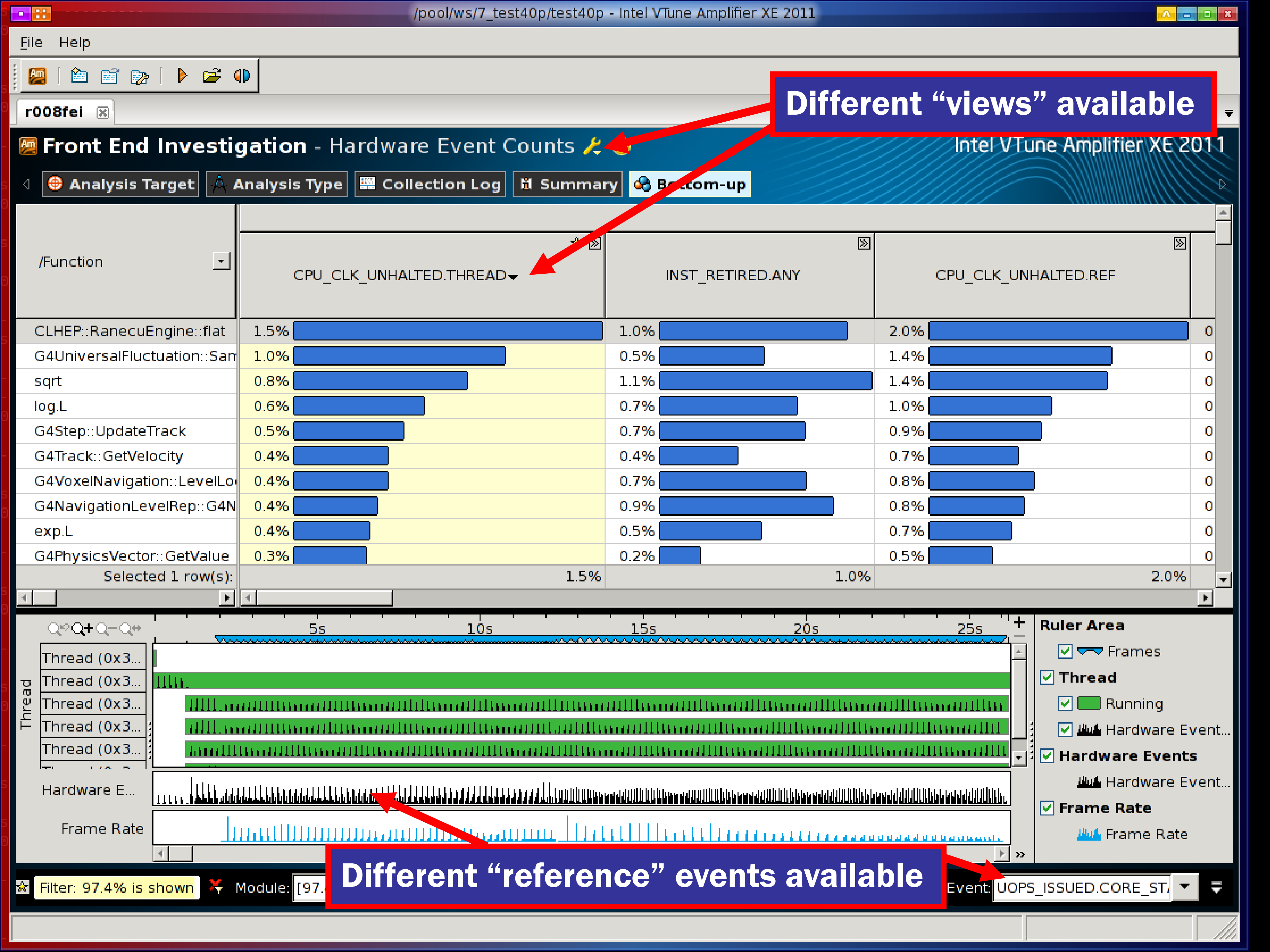


Locks and waits analysis (2)

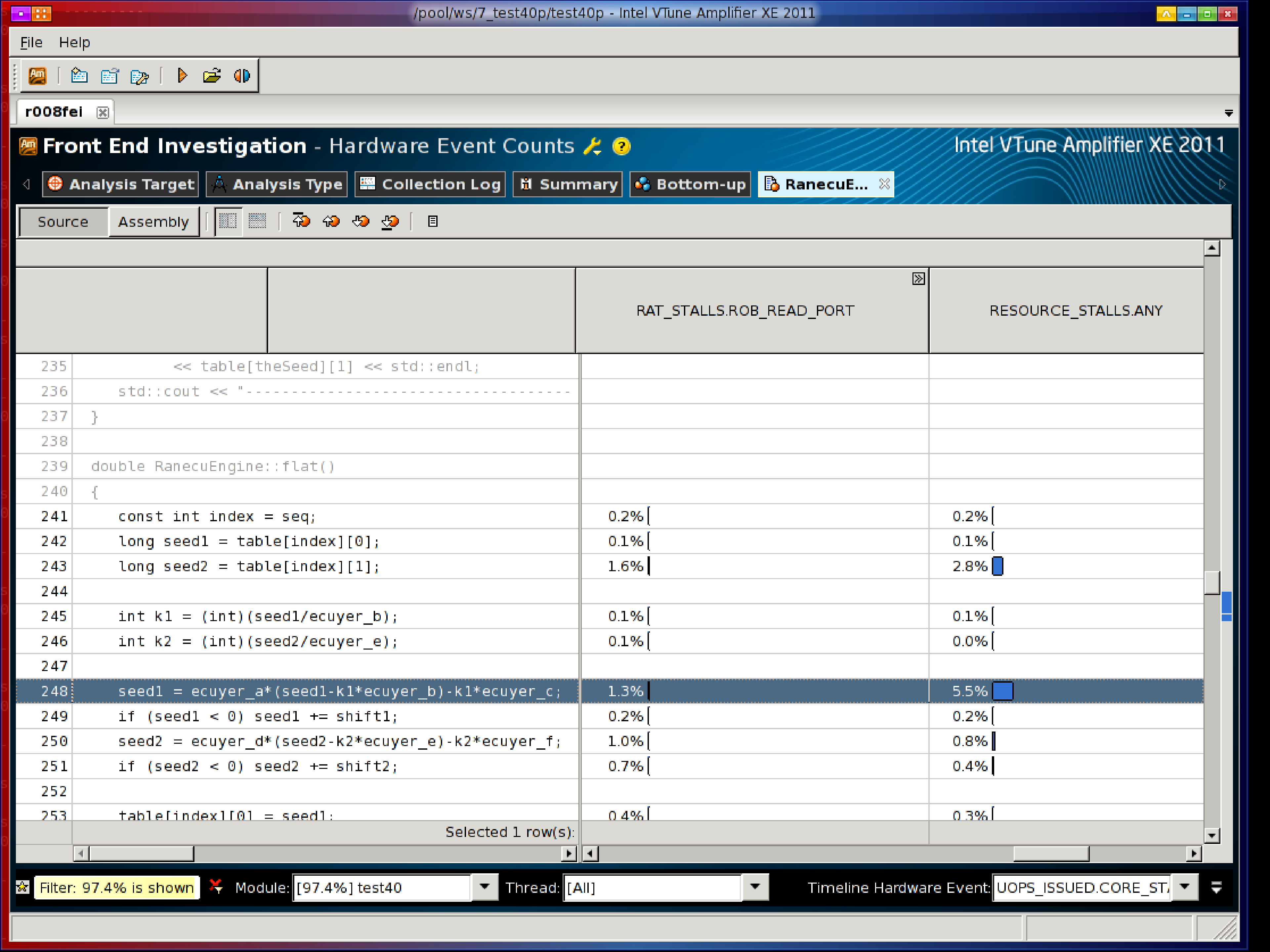
- See the precise lock location and the time spent in locks



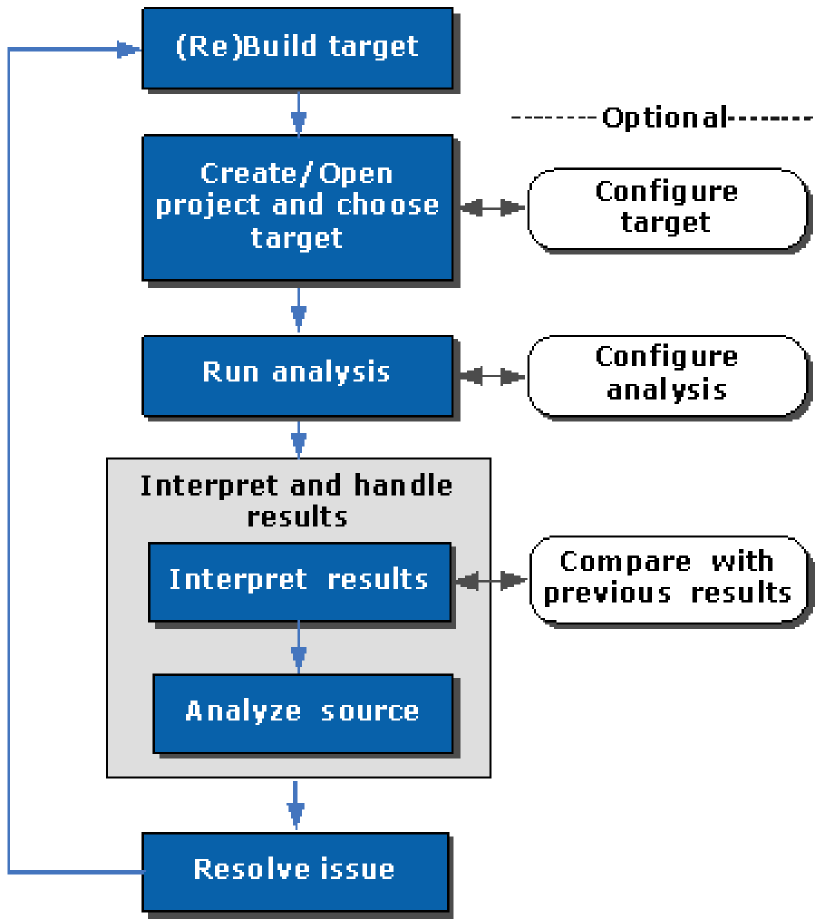

/pool/ws/7_test40p/test40p - Intel VTune Amplifier XE 2011
File
Help
r008fei
Front End Investigation - Hardware Event Counts
Intel VTune Amplifier XE 2011
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Hardware Event Count
/Function
CPU_CL... THREAD
INST_RETIR... ANY
CPU_CLK_U... REF
BR_INST_E... ANY
BR_MIS... ANY
ILD_STALL. ANY
ILD_S... LCP
ITLB_MI...
L1I. CYCLES_S...
CLHEP::RanecuEngine::flat 18,056,000,000 11,806,000,000 25,404,000,000 931,200,000 55,360,000 11,080,000,000 8,000,000 18,400,000 464,000,000
G4UniversalFluctuation::Sam 12,408,000,000 6,598,000,000 17,940,000,000 1,411,200,000 96,400,000 6,000,000,000 0 13,600,000 1,608,000,000
sqrt 10,202,000,000 13,352,000,000 17,494,000,000 1,657,600,000 9,840,000 1,600,000,000 8,000,000 10,400,000 1,720,000,000
log.L 7,692,000,000 8,704,000,000 12,038,000,000 701,600,000 6,240,000 3,056,000,000 0 4,800,000 544,000,000
G4Step::UpdateTrack 6,500,000,000 9,202,000,000 11,102,000,000 1,098,400,000 4,240,000 728,000,000 0 8,800,000 552,000,000
G4Track::GetVelocity 5,570,000,000 4,938,000,000 8,876,000,000 1,089,600,000 3,760,000 984,000,000 0 4,800,000 784,000,000
G4VoxelNavigation::LevelLo 5,550,000,000 9,278,000,000 10,442,000,000 1,279,200,000 5,760,000 1,480,000,000 0 9,600,000 624,000,000
G4NavigationLevelRep::G4N 4,956,000,000 10,944,000,000 9,448,000,000 272,000,000 80,000 1,184,000,000 0 11,200,000 96,000,000
exp.L 4,490,000,000 6,470,000,000 8,144,000,000 572,800,000 8,800,000 872,000,000 0
G4PhysicsVector::GetValue 4,312,000,000 2,620,000,000 6,282,000,000 632,800,000 20,720,000 328,000,000 0
Selected 1 row(s): 18,056,000,000 11,806,000,000 25,404,000,000 931,200,000 55,360,000 11,080,000,000 8,000,000
Results
5s
10s
15s
20s
25s
Thread
Thread (0x3...
Hardware E...
Frame Rate
Ruler Area
Frames
Running
Hardware Event...
Hardware Events
Timeline view
Filter: 97.4% is shown
Module: [97.4%] test40
Thread: [All]
Filters
CPU_CLK_UNHALTED.TH



/pool/ws/7_test40p/test40p - Intel VTune Amplifier XE 2011
File
Help
r008fei
Different "views" available
Front End Investigation - Hardware Event Counts
Intel VTune Amplifier XE 2011
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
/Function
CPU_CLK_UNHALTED.THREAD
INST_RETIRED.ANY
CPU_CLK_UNHALTED.REF
CLHEP::RanecuEngine::flat 1.5% 1.0% 2.0%
G4UniversalFluctuation::Sam 1.0% 0.5% 1.4%
sqrt 0.8% 1.1% 1.4%
log.L 0.6% 0.7% 1.0%
G4Step::UpdateTrack 0.5% 0.7% 0.9%
G4Track::GetVelocity 0.4% 0.4% 0.7%
G4VoxelNavigation::LevelLo 0.4% 0.7% 0.8%
G4NavigationLevelRep::G4N 0.4% 0.9% 0.8%
exp.L 0.4% 0.5% 0.7%
G4PhysicsVector::GetValue 0.3% 0.2% 0.5%
Selected 1 row(s): 1.5% 1.0% 2.0%
5s
10s
15s
20s
25s
Thread
Thread (0x3...
Thread (0x3...
Thread (0x3...
Thread (0x3...
Thread (0x3...
Hardware E...
Frame Rate
Ruler Area
Frames
Thread
Running
Hardware Event...
Hardware Events
Hardware Event...
Frame Rate
Frame Rate
Filter: 97.4% is shown
Module: [97.
Different "reference" events available
Event: UOPS_ISSUED.CORE_ST/


/pool/ws/7_test40p/test40p - Intel VTune Amplifier XE 2011
File Help
r008fei
Front End Investigation - Hardware Event Counts
Intel VTune Amplifier XE 2011
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
RanecuE...
Source
Assembly
RAT_STALLS.ROB_READ_PORT
RESOURCE_STALLS.ANY
235 << table[theSeed][1] << std::endl;
236 std::cout << "-----------------------------------
237 }
238
239 double RanecuEngine::flat()
240 {
241 const int index = seq; 0.2% 0.2%
242 long seed1 = table[index][0]; 0.1% 0.1%
243 long seed2 = table[index][1]; 1.6% 2.8%
244
245 int k1 = (int)(seed1/ecuyer_b); 0.1% 0.1%
246 int k2 = (int)(seed2/ecuyer_e); 0.1% 0.0%
247
248 seed1 = ecuyer_a*(seed1-k1*ecuyer_b)-k1*ecuyer_c; 1.3% 5.5%
249 if (seed1 < 0) seed1 += shift1; 0.2% 0.2%
250 seed2 = ecuyer_d*(seed2-k2*ecuyer_e)-k2*ecuyer_f; 1.0% 0.8%
251 if (seed2 < 0) seed2 += shift2; 0.7% 0.4%
252
253 table[index][0] = seed1; 0.4% 0.3%
Selected 1 row(s):
Filter: 97.4% is shown
Module: [97.4%] test40
Thread: [All]
Timeline Hardware Event: UOPS_ISSUED.CORE_ST/

Workflow



- The basic steps to get going are identical to those in “Inspector”
- The custom workflow for this application is also similar to “Inspector’s” and is shown on the right

Inspector

Threading and memory correctness

Introduction

- **A dynamic memory and threading error checking tool**
- **Languages supported:**
  - C, C++, C#, Fortran
- **Technologies supported:**
  - TBB, Cilk+, pthreads, Windows threads, OpenMP
- **Operating systems supported (same functionality):**
  - Linux
  - Windows
- **Replacement tool for Thread Checker**

Features – instrumented analysis

> Memory error detection and location

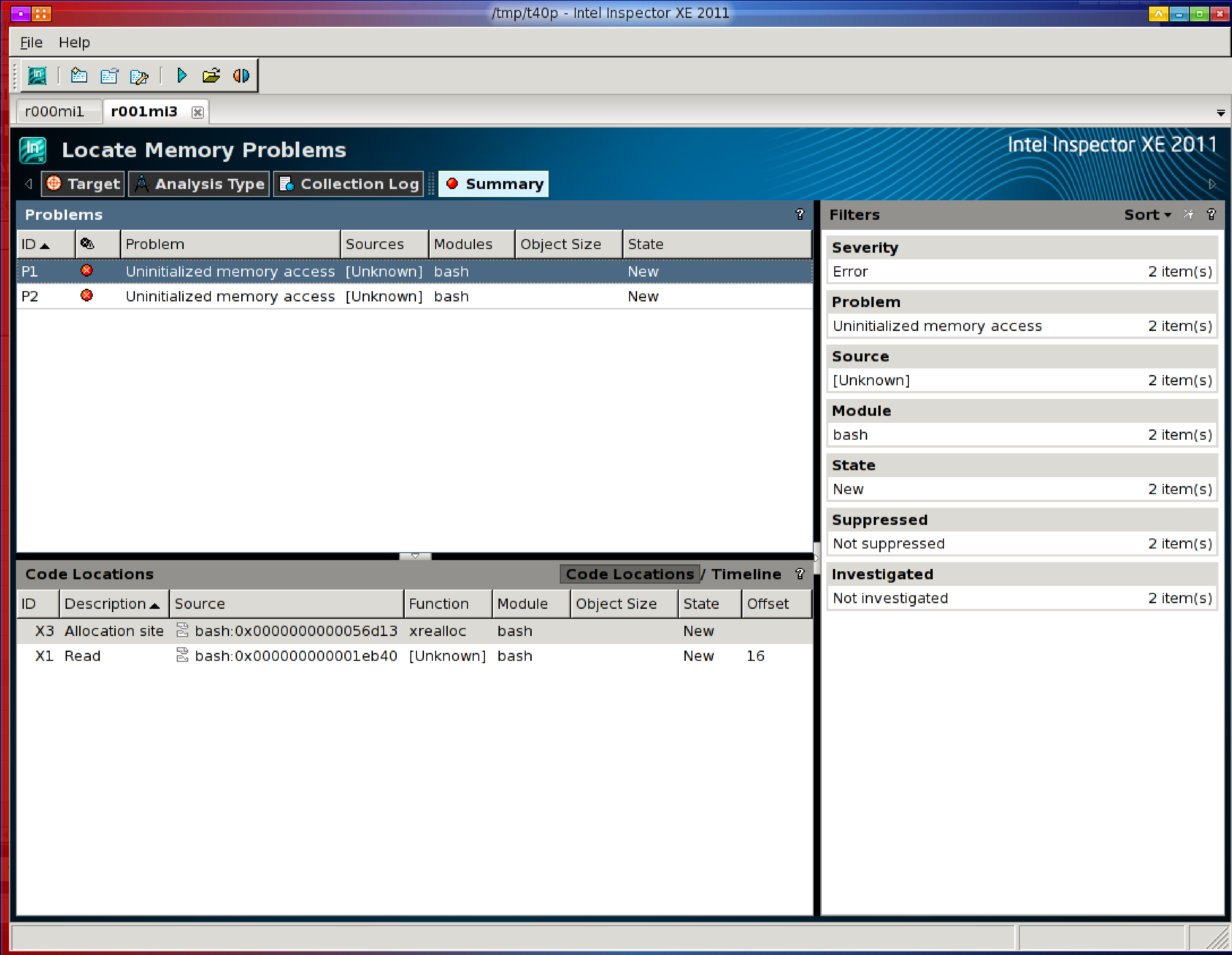
- Detect leaks
  - Detects memory leaks
- Detect memory problems
  - In addition to the above: detects uninitialized accesses
- Locate memory problems
  - In addition to the above: detects dangling pointers, enables guard zones, deep stack analysis

> Threading error detection and location

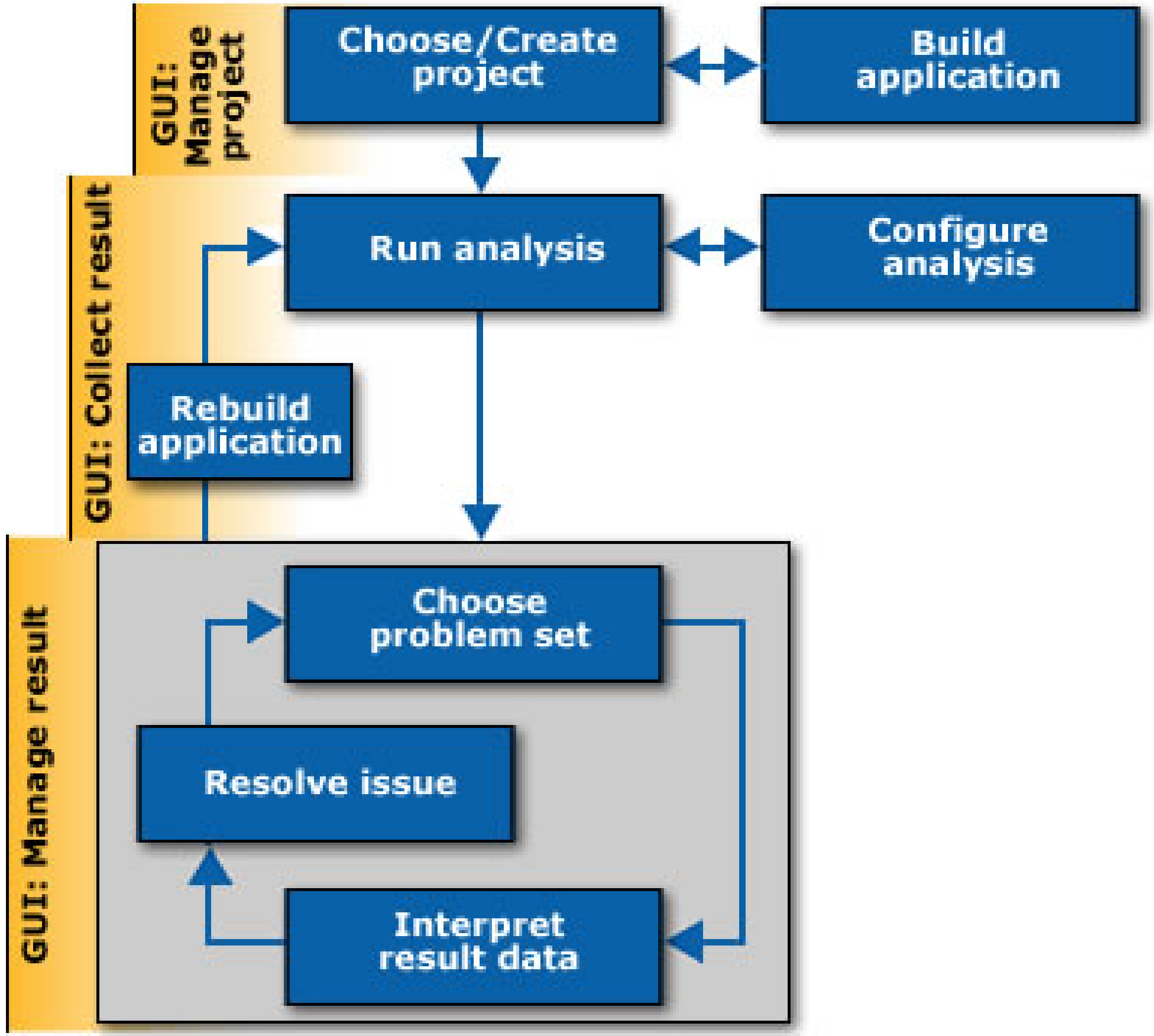
- Detect deadlocks
  - Detects lock hierarchy and deadlocks
- Detect data races
  - In addition to the above: detects cross-thread stack accesses, data races
- Locate deadlocks and data races
  - In addition to the above: collects stack, finer memory access granularity

> Static security analysis

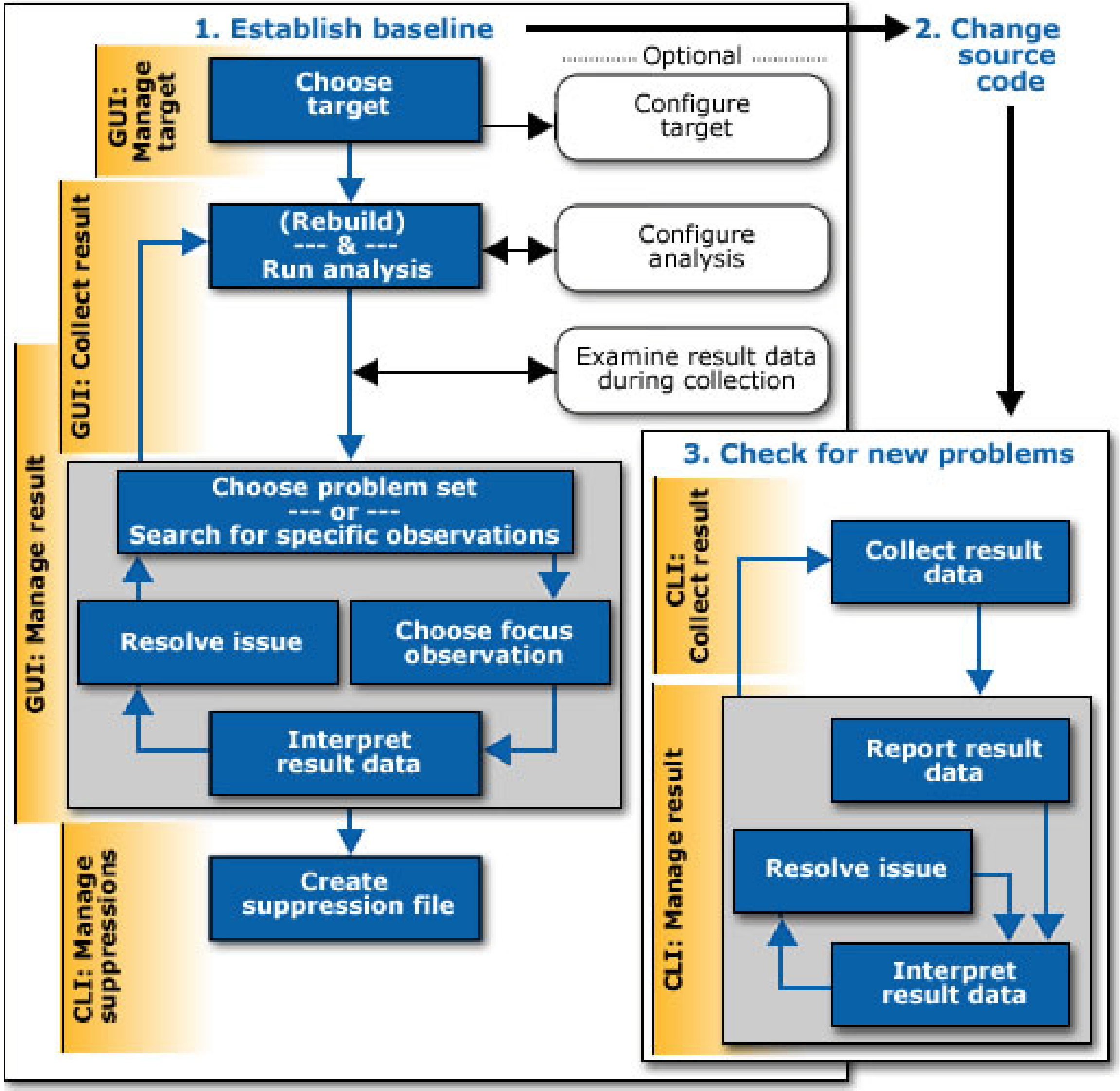
- Visualizes output from analysis performed with Intel compilers


/tmp/t40p - Intel Inspector XE 2011
File
Help
r000mi1
r001mi3
Locate Memory Problems
Intel Inspector XE 2011
Target
Analysis Type
Collection Log
Summary
Problems
ID
Problem
Sources
Modules
Object Size
State
P1
Uninitialized memory access
[Unknown]
bash
New
P2
Uninitialized memory access
[Unknown]
bash
New
Filters
Sort
Severity
Error
2 item(s)
Problem
Uninitialized memory access
2 item(s)
Source
[Unknown]
2 item(s)
Module
bash
2 item(s)
State
New
2 item(s)
Suppressed
Not suppressed
2 item(s)
Investigated
Not investigated
2 item(s)
Code Locations
Code Locations / Timeline
ID
Description
Source
Function
Module
Object Size
State
Offset
X3
Allocation site
bash:0x0000000000056d13
xrealloc
bash
New
X1
Read
bash:0x000000000001eb40
[Unknown]
bash
New
16

Basic workflow - overview



Advanced workflow with regression testing



API

Instrumenting your programs for a streamlined optimization process

API

- **You can use "Intel Threading Tools" calls in your software in order to specify certain actions**
  - Start and stop monitoring (data collection)
  - Describe regions of your code
  - Rename threads
  - Describe synchronization objects
  - Define loop limits
- **Usage:**
  - Include ittnotify.h
  - Link with ittnotify.a

API – examples (Pause/Resume)

```
// code, work – collection was started paused
//                so no profiling data is gathered

__itt_resume(); // switch on profiling

// code, work (profiled)

__itt_pause(); // switch off profiling
```

> **Example usage:**
- Monitoring restricted to a certain routine
- Monitoring enabled only past a certain point

API – examples (Frames)

```
__itt_frame frame = __itt_frame_create("G4 Events");

for ( ... ) {
   __itt_frame_begin(frame);

   // ... loop code

   __itt_frame_end(frame);
}
```

- **Example usage:**
  - Designation of cyclic occurrences – such as events in a physics simulation (for display/grouping purposes)
  - Frame groups (“domains”) available
  - Different frame groupings available

Frame grouping - example

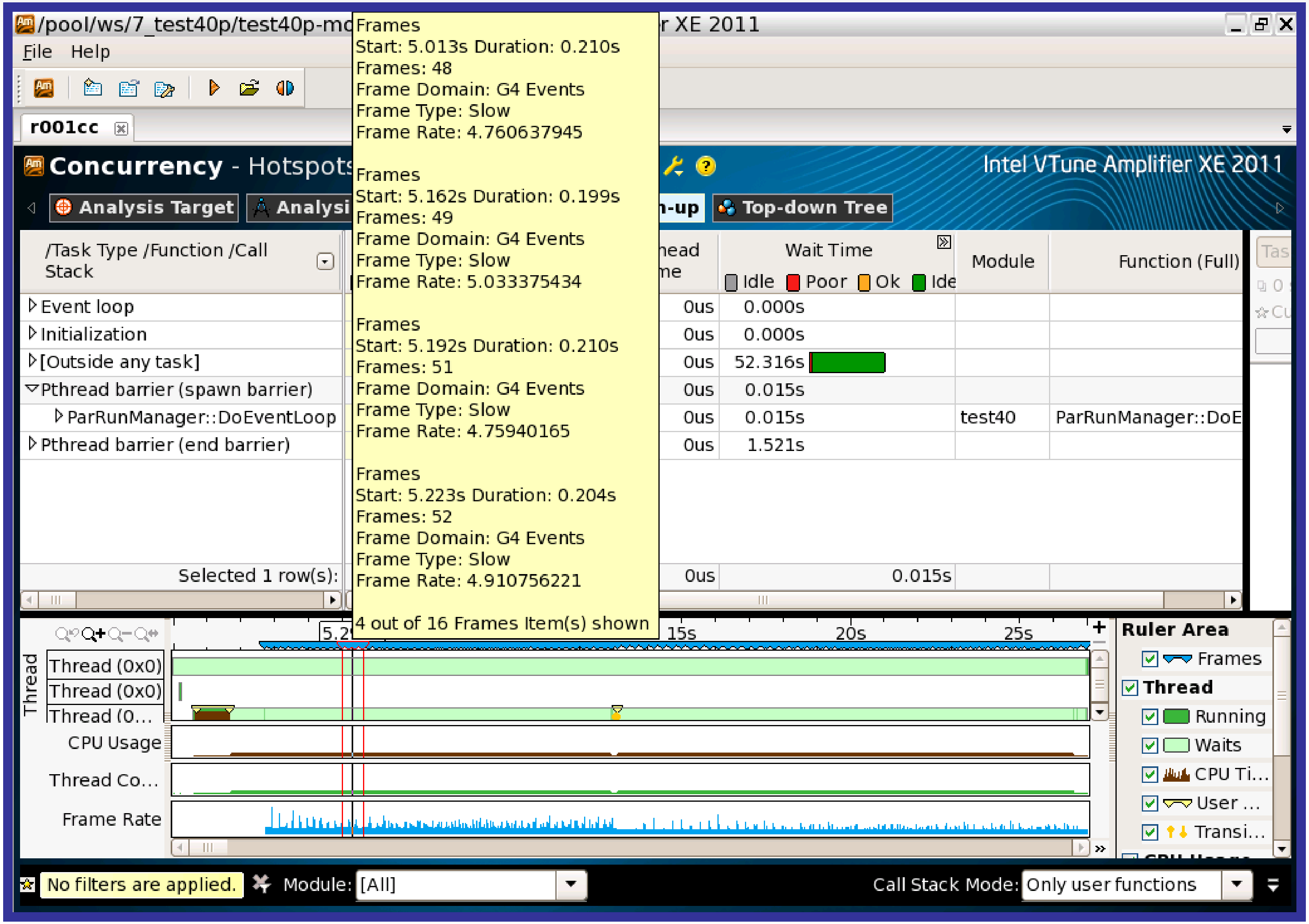
/Frame Domain /Frames /Function /Call Stack	CPU Time	Module	Function (Full)
G4 Events	95.099s		
213	1.072s		
217	1.069s		
237	1.051s		
285	1.050s		
sqrt	0.120s	test40	sqrt
G4SteppingManager::DefinePhysicalStepLength	0.061s	test40	G4SteppingMa...
CLHEP::RanecuEngine::flat	0.060s	test40	CLHEP::Ranec...
G4Mag_UsualEqRhs::EvaluateRhsGivenB	0.040s	test40	G4Mag_UsualE...
G4NavigationLevelRep::G4NavigationLevelRep	0.040s	test40	G4NavigationL...
G4UniversalFluctuation::SampleFluctuations	0.040s	test40	G4UniversalFlu...
exp.L	0.030s	test40	exp.L
G4PEEffectModel::ComputeCrossSectionPerAtom	0.030s	test40	G4PEEffectMod...
G4Track::GetVelocity	0.030s	test40	G4Track::GetV...
G4SteppingManager::Stepping	0.030s	test40	G4SteppingMa...
Selected 1 row(s):	1.050s		

API – examples (Regions/events)

```
// “10” refers to the length of the description string
__itt_event ev_loop = __itt_event_create(“Event loop”, 10);

__itt_event_start(ev_loop);

// ...  Work ...

__itt_event_end(ev_loop) ;
```

- **Example usage:**
  - Designation of code regions (for display/grouping purposes), e.g. “Initialization”, “Detector construction”, “Simulation”, “Finalization”

Regions ("Task") grouping - example



Takeaway advice

> **Instrumented analysis might take quite a while**
> - Whenever possible, always try to choose a representative data set for monitoring
> - Reduce the detail level of the analysis; for example, in “Locks and waits”, uncheck “Spin time data” and “Collect signals” whenever you don’t need that data

> **Hardware-level analysis is as fast as the application itself**
> - No need to reduce your data set!

> **The tools come with APIs which you can use to instrument your source code**

> **Results on non-Intel CPUs should generally be fine, but may be offset or incorrect**

> **Take a look at the documentation, it’s worth it!**

Practical information

- **Intel tools are available pre-installed CERN-wide in the standard AFS folder**
  - /afs/cern.ch/sw/IntelSoftware
  - Ideally: source `all-setup.sh` and you're set up
- **For more information, read the openlab TWiki or the openlab webpages**
  - http://twiki.cern.ch/ -> openlab web
  - http://cern.ch/openlab
- **Graphical version: `amplxe-gui`**
- **Command line: `amplxe-cl`**

Q & A

Other questions? andrzej.nowak@cern.ch

BACKUP

With material from the Intel tools documentation

Key terms (1)

- **analysis**: A process during which the tool performs collection and finalization.
- **code location**: A fact the tool observes at a source code location, such as a write code location. Sometimes called an observation. A focus code location is a source code location with relationships you choose to explore. A related code location is a source code location with a relationship to a focus code location and possibly other code locations.
- **collection**: A process during which the tool executes an application, identifies issues that may need handling, and collects those issues in a result.
- **false positive**: The tool detects something that is not an error.
- **false negative**: The tool does not detect an error because the problem may be too complex/big or involve too much runtime/memory cost.

Key terms (2)

- **finalization**: A process during which the the tool uses debug information from binary files to convert symbol information into filenames and line numbers, perform duplicate elimination, and form problem sets.
- **problem**: A small group of closely related code locations that indicate an error in an application, such as a data race problem.
- **problem set**: A larger group of more loosely related code locations that could share a common solution, such as a problem set resulting from deallocating an object too early during program execution. You can view problem sets only after analysis is complete.
- **project**: A compiled application, collection of configurable attributes for the compiled application, and a container for results and private suppression rules.
- **result**: A collection of issues that may need handling.
- **target**: An application you inspect for errors

Key terms (3)

- **baseline**: A performance metric used as a basis for comparison of the application versions before and after optimization. Baseline should be measurable and reproducible.
- **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. The application CPU time is the sum of the CPU time of all the threads that run the application.
- **elapsed time**: The total time your target ran, calculated as follows: Wall clock time at end of application – Wall clock time at start of application.
- **hotspot**: A section of code that took a long time to execute. Some hotspots may indicate bottlenecks and can be removed, while other hotspots inevitably take a long time to execute due to their nature.
- **viewpoint**: A preset result tab configuration that filters out the data collected during a performance analysis and enables you to focus on specific performance problems. When you select a viewpoint, you select a set of performance metrics the tool shows in the windows/panes of the result tab. To select the required viewpoint, use the drop-down menu ("wrench") at the top of the result tab.
- **wait time**: The amount of time that a given thread waited for some event to occur, such as: synchronization waits and I/O waits.

Key Concept: CPU Utilization

> For the Concurrency and the Locks and Waits analyses, the Intel(R) VTune(TM) Amplifier XE identifies a processor utilization scale, calculates the target concurrency, and defines default utilization ranges depending on the number of processor cores. You can change the utilization ranges by dragging the slider in the Summary window.

Utilization Type	Default color	Description
Idle		All threads in the program are waiting - no threads are running. There can be only one node in the **Summary** chart indicating idle utilization.
Poor		Poor utilization. By default, poor utilization is when the number of threads is up to 50% of the target concurrency.
OK		Acceptable (OK) utilization. By default, OK utilization is when the number of threads is between 51-85% of the target concurrency.
Ideal		Ideal utilization. By default, ideal utilization is when the number of threads is between 86-115% of the target concurrency.
Over		Over-utilization. By default, over-utilization is when the number of threads is more than 115% of the target concurrency.

Key Concept: Hardware-level Analysis

> The VTune Amplifier XE introduces a set of advanced hardware analysis types based on the event-based sampling data collection and targeted for the Intel(R) Core(TM) 2 processor family and processors based on the Intel(R) microarchitecture codename Nehalem. Depending on the analysis type, the VTune Amplifier XE monitors a set of hardware events and, as a result, provides collected data per, so-called, *hardware performance metrics* defined by Intel architects (for example, Clockticks per Instructions Retired, Contested Accesses, and so on). Each metric is an event ratio with its own threshold values. As soon as the performance of a program unit per metric exceeds the threshold, the VTune Amplifier XE marks this value as a performance issue and provides recommendations how to fix it.

> Typically, you are recommended to start with the General Exploration analysis type that collects the maximum number of events and provides the widest picture of the hardware issues that affected the performance of your application.

Key Concept: Hotspots Analysis

- The Hotspots analysis helps understand the application flow and identify sections of code that took a long time to execute (*hotspots*). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hotspots can be removed, while other hotspots are fundamental to the application functionality and cannot be removed.
- The Intel(R) VTune(TM) Amplifier XE creates a list of functions in your application ordered by the amount of time spent in a function. It also detects the call stacks for each of these functions so you can see how the hot functions are called.
- The VTune Amplifier XE uses a low overhead (about 5%) statistical sampling algorithm that gets you the information you need without a significant slowing of application execution.

Key Concept: Locks and Waits Analysis

- While the Concurrency analysis helps identify where your application is not parallel, the Locks and Waits analysis helps identify the cause of the ineffective processor utilization. One of the most common problems is threads waiting too long on synchronization objects (locks). Performance suffers when waits occur while cores are under-utilized.
- During the Locks and Waits analysis you can estimate the impact each synchronization object introduces to the application and understand how long the application was required to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O.

Key Concept: Choosing Small, Representative Data Sets

> When you run a dynamic analysis, the tool executes an application against a data set. Data set size has a direct impact on application execution time and analysis speed.

> You can control analysis *cost* without sacrificing completeness by removing redundancies from your data set (e.g. redundant iterations).

> Instead of choosing large, repetitive data sets, choose small, representative data sets. Data sets with runs in the seconds time range are ideal. You can always create additional data sets to ensure all your code is inspected.

Key Concept: Data of Interest

- **The VTune Amplifier XE maintains a special column called Data of Interest. This column is highlighted with yellow background and a yellow star in the column header .**
- **The data in the Data of Interest column is used by various windows as follows:**
  - The Call Stack pane calculates the contribution, shown in the contribution bar, using the Data of Interest column values.
  - The Filter bar uses the data of interest values to calculate the percentage indicated in the filtered option.
  - The Source/Assembly window uses this column for hotspot navigation.
- **If a viewpoint has more than one column with numeric data or bars, you can change the default <u>Data of Interest</u> column by right-clicking the required column and selecting the <u>Set Column as Data of Interest</u> command from the pop-up menu.**

Key Concept: Finalization

> *Finalization* is a process when the VTune Amplifier XE converts the collected data to a database, resolves symbol information, and pre-computes data to make further analysis more efficient and responsive. The VTune Amplifier XE finalizes data automatically when generating results.

> You may want to re-finalize a result to:
  - update symbol information after changes in the search directories settings
  - resolve the number of [Unknown]-s in the results

“Amplifier”: Algorithm analysis

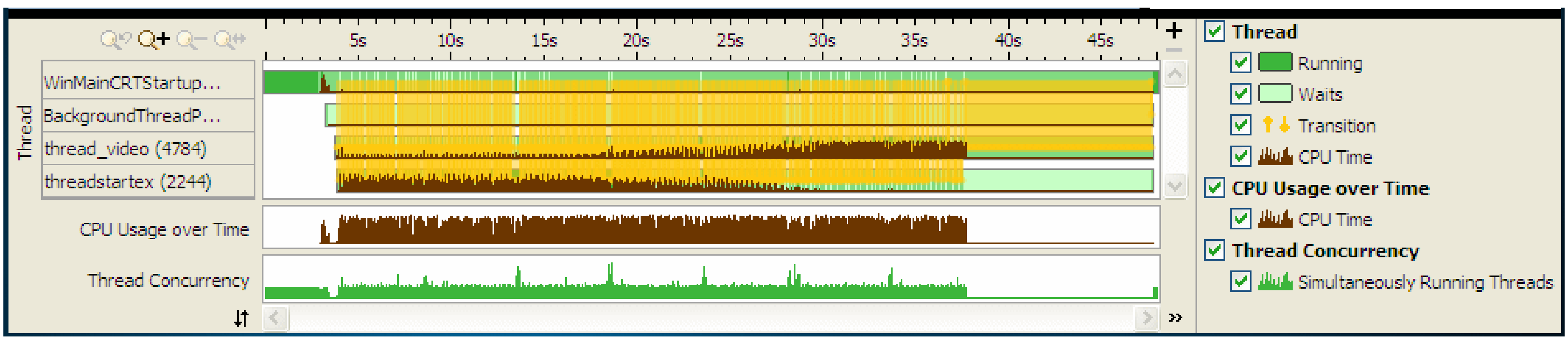
- Algorithm analysis branch introduces analysis types targeted for software tuning. You run the analysis and use the collected data to understand where you could choose a better algorithm, and improve the application performance. Algorithm analysis includes the following analysis types:
- Lightweight Hotspots: Event-based sampling analysis that monitors all the software executing on your system including the operating system modules. The collector interrupts the processor at the specified sampling interval and collects samples of instruction addresses.
- Hotspots: Performance analysis based on the user-mode sampling and tracing collection. It focuses on a particular target, identifies functions that took the most CPU time to execute, restores the call tree for each function, and shows thread activity.
- Concurrency: Performance analysis based on the user-mode sampling and tracing collection. It focuses on a particular target, identifies functions that took the most CPU time to execute, and shows how well your application is threaded for the existing number of logical CPUs.
- Locks and Waits: Performance analysis based on the user-mode sampling and tracing collection that helps identify the synchronization objects that caused ineffective CPU usage.

“Amplifier”: Hardware-level analysis

- The Advanced hardware-level analysis introduces a set of analysis types based on the event-based sampling data collection and targeted for the Intel(R) Core(TM) 2 processor family and Intel(R) microarchitecture codename Nehalem.
- General Exploration: Event-based analysis that helps identify the most significant hardware issues affect the performance of your application. Consider this analysis type as a starting point when you make the hardware-level analysis on Intel microarchitecture codename Nehalem.
- Cycles and uOps: Event-based analysis that helps understand where the cycles and uOps issues affect the performance of your application.
- Front End Investigation: Event-based analysis that helps understand where the front end issues affect the performance of your application.
- Memory Access: Event-based analysis that helps understand where the memory access issues affect the performance of your application.

Amplifier: Timeline view

Thread State	Description	Viewpoint
Running	The time threads are active.	Hotspots, Hotspots by CPU Usage, Hotspots by Thread Concurrency, Locks and Waits
Waits	The time threads are spending waiting for a particular object.	Hotspots by Thread Concurrency, Locks and Waits
Transition	The execution flow between threads where one thread signals to another thread waiting to receive that signal. For example, one thread attempts to acquire a lock held by another thread, which then releases it. The release acts like a signal to the waiting thread.	Hotspots by Thread Concurrency, Locks and Waits
CPU Time	The CPU time utilization by a thread during the application run.	Hotspots, Hotspots by CPU Usage, Lightweight Hotspots, Hotspots by Thread Concurrency
Analysis Metrics	**Description**	**Viewpoint**
CPU Usage	The CPU time utilization over time for the whole application.	Hotspots, Hotspots by CPU Usage
Thread Concurrency	The concurrency level for the whole application.	Hotspots, Hotspots by Thread Concurrency, Locks and Waits
Hardware Events Sample Count	Distribution of the application performance per metric over time. This data is available for the event-based analysis sampling collection types where each metric is based on a set of processor events.	Hardware Event Counts, Hardware Event Sample Counts, Hardware Issues



Amplifier: working with performance events