



Tools for OpenMP Programming

PPCES 2016

Tim Cramer
Dirk Schmidl

■ Intel Inspector XE

→ Overview

→ Live Demo

■ Intel VTune Amplifier XE

→ Overview

→ Live Demo

Race Condition

- **Data Race: the typical OpenMP programming error, when:**
 - two or more threads access the same memory location, and
 - at least one of these accesses is a write, and
 - the accesses are not protected by locks or critical regions, and
 - the accesses are not synchronized, e.g. by a barrier.
- **Non-deterministic occurrence: e.g. the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run**
- **In many cases *private* clauses, *barriers* or *critical regions* are missing**
- **Data races are hard to find using a traditional debugger**
 - Use the *Intel Inspector XE*

■ Detection of

- Memory Errors
- Dead Locks
- Data Races

■ Support for

- Linux (32bit and 64bit) and Windows (32bit and 64bit)
- WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP

■ Features

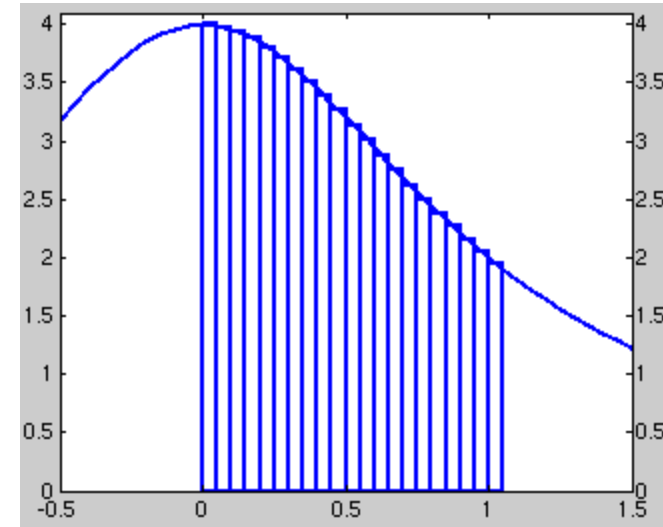
- Binary Instrumentation gives full functionality
- Independent stand-alone GUI for Windows and Linux
- Memory error detection
- ~~Static security analysis (in combination with the Intel compiler)~~

PI Example Code

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;
```



```
#pragma omp parallel for private(fX,i) reduction(+:fSum)
for (i = 0; i < n; i++)
{
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
}
return fH * fSum;
}
```

PI Example Code

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;
```

```
#pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

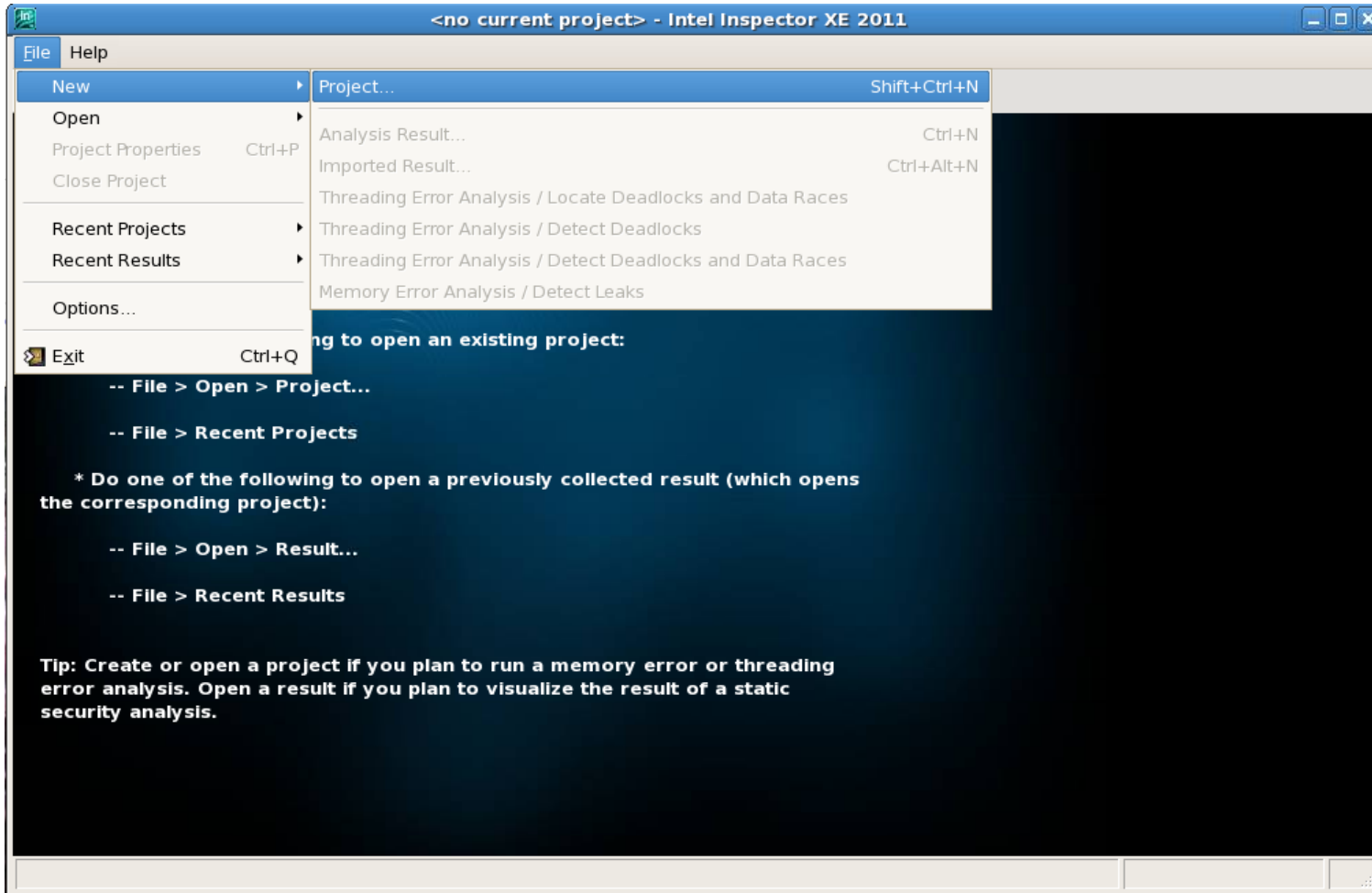
What if we
would have
forgotten
this?

Live Demo

Intel Inspector XE

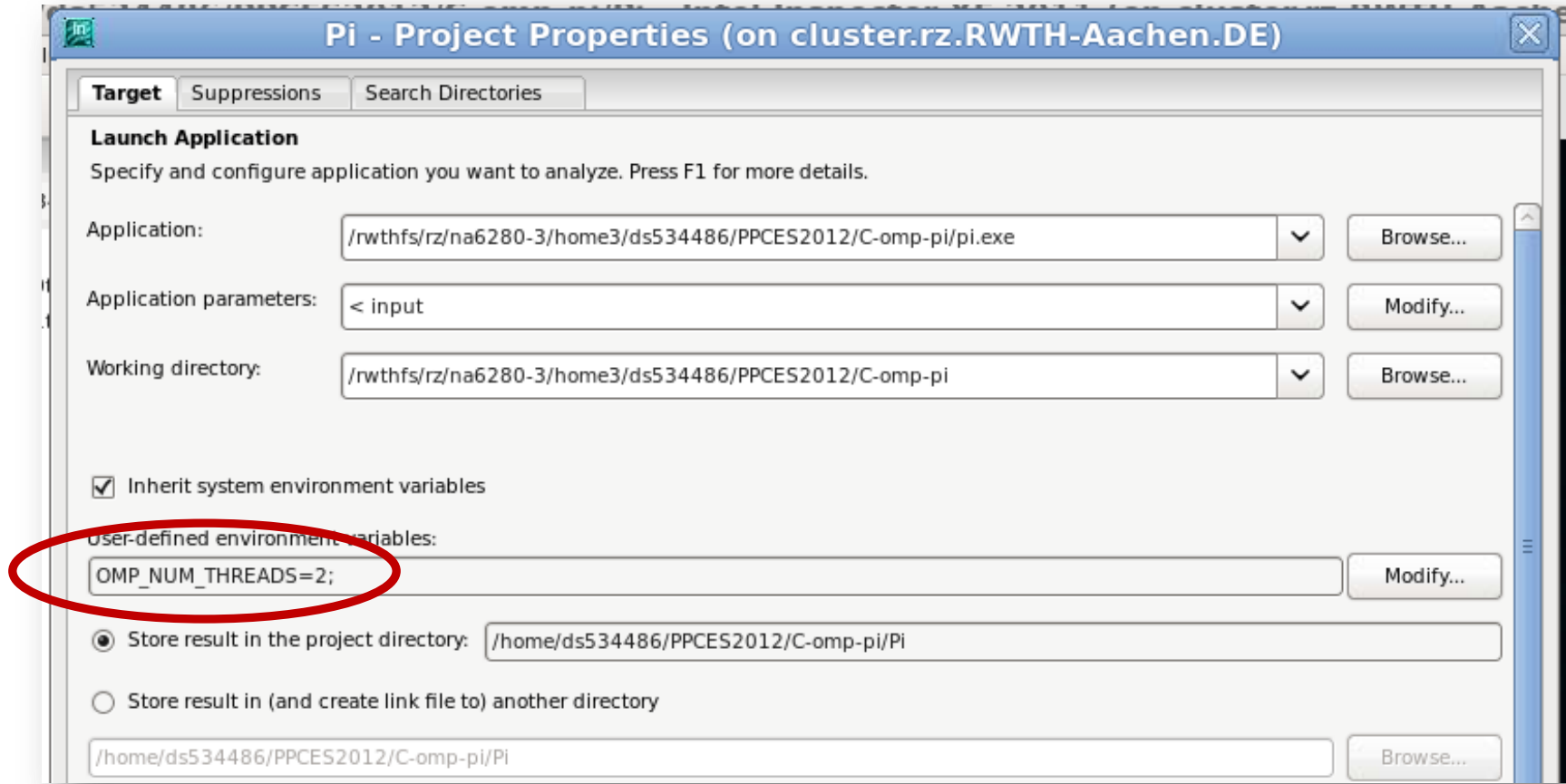
Inspector XE – Create Project

\$ module load intelixe ; inspxe-gui



Inspector XE – Create Project

- ensure that multiple threads are used
- choose a small dataset (really!), execution time can increase 10X – 1000X

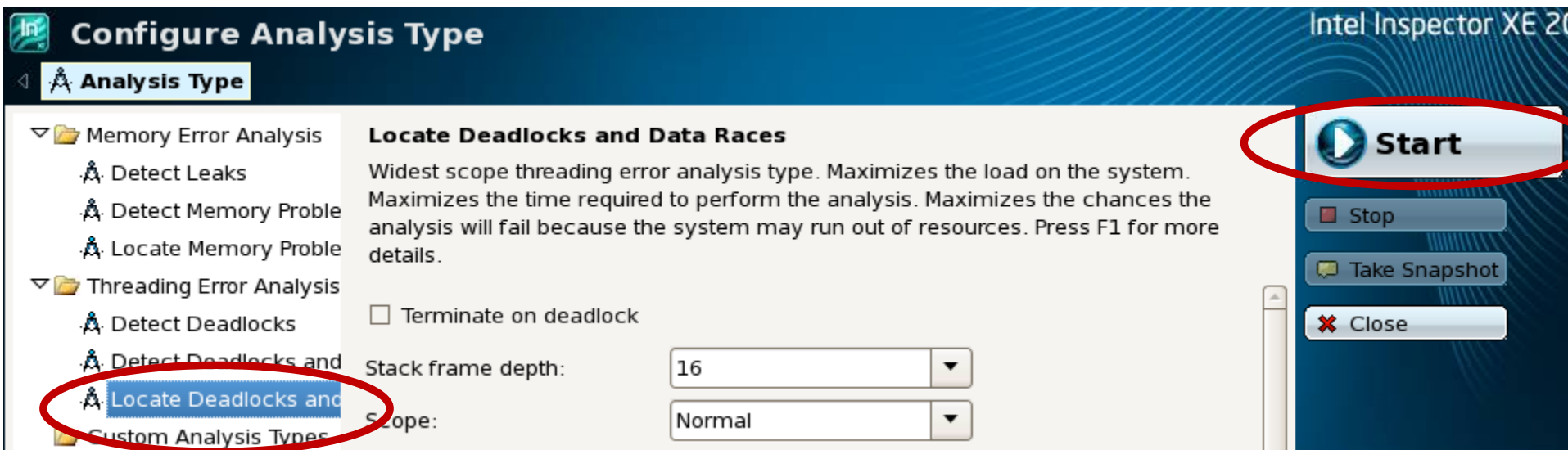


Inspector XE – Configure Analysis

Threading Error Analysis Modes

1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

more details,
more overhead



Inspector XE – Results

- 1 detected problems
- 2 filters
- 3 code location

The screenshot displays the Intel Inspector XE 2011 interface. The main window is titled "Locate Deadlocks and Data Races". The "Problems" pane on the left shows a single problem, P1, which is a "Data race" in the file "pi.c" at module "pi.exe", with a state of "New". A yellow circle with the number "1" is placed next to this problem entry.

The "Code Locations" pane on the left shows the source code for the "Data race" problem. It lists two locations: X1 (Read) and X2 (Write), both in the file "pi.c" at line 71, within the function "CalcPi" in the module "pi.exe". The code snippet for both locations is:

```
69 {  
70     fX = fH * ((double)i + 0.5);  
71     fSum += f(fX);  
72 }  
73 return fH * fSum;
```

A yellow circle with the number "3" is placed next to the code snippet for location X2.

The "Filters" pane on the right shows the following filters:

Severity	Count
Error	1 item(s)

Problem	Count
Data race	1 item(s)

Source	Count
pi.c	1 item(s)

Module	Count
pi.exe	1 item(s)

State	Count
New	1 item(s)

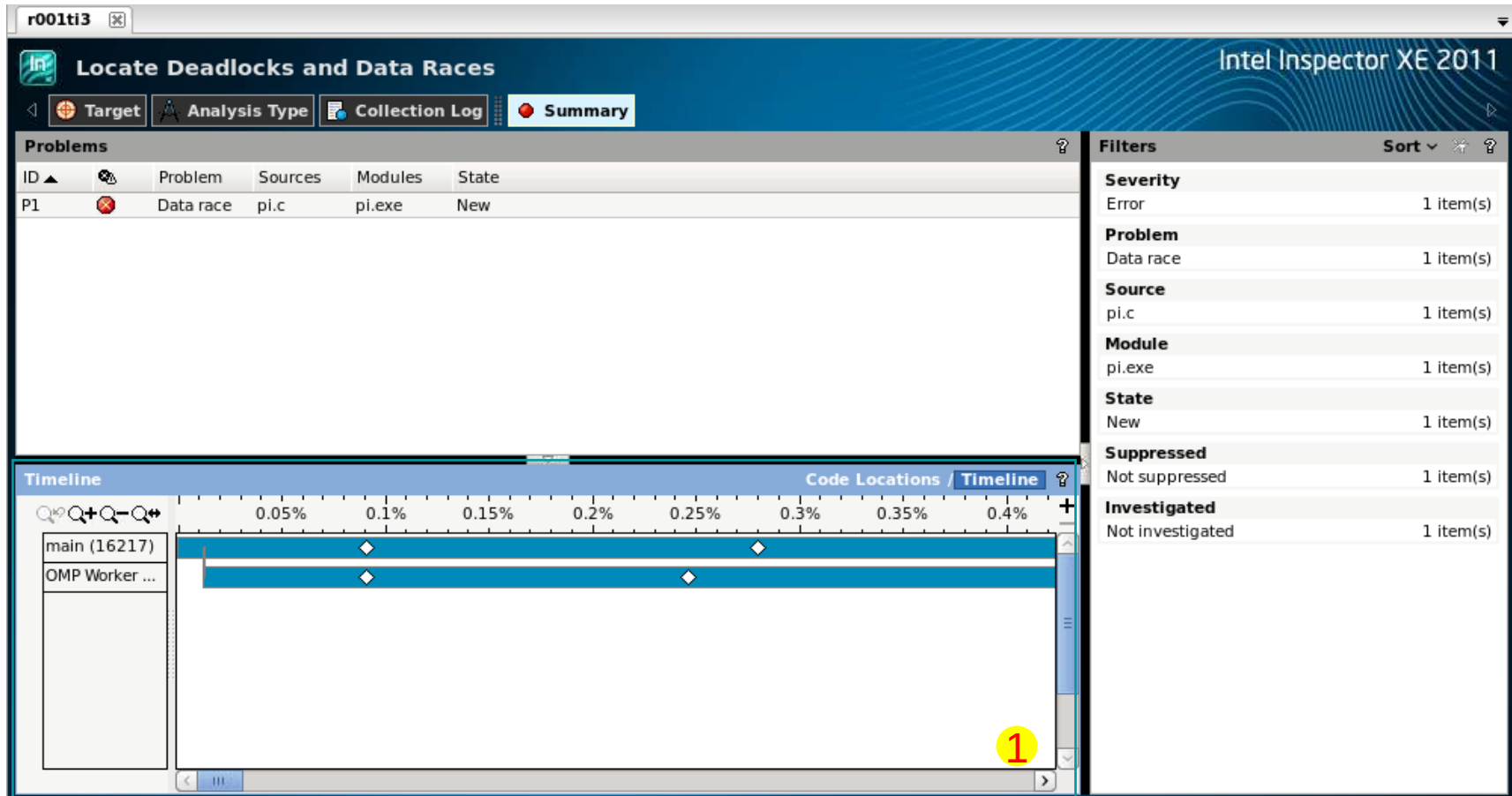
Suppressed	Count
Not suppressed	1 item(s)

Investigated	Count
Not investigated	1 item(s)

A yellow circle with the number "2" is placed next to the "Not investigated" entry in the "Investigated" filter table.

Inspector XE – Results

1 Timeline view



Inspector XE – Results

- 1 Source Code producing the issue – double click opens an editor
- 2 Corresponding Call Stack

The screenshot displays the Intel Inspector XE 2011 interface. The top bar shows the target 'r001ti3' and the title 'Locate Deadlocks and Data Races'. Below this are tabs for 'Target', 'Analysis Type', 'Collection Log', 'Summary', and 'Sources'. The main area is divided into two panels, each showing source code and a call stack.

Focus Code Location: pi.c:71 - Write

```
68 for (i = iRank; i < n; i += iNumProcs)
69 {
70     fX = fH * ((double)i + 0.5);
71     fSum += f(fX);
72 }
73 return fH * fSum;
74
75
```

Call Stack

- pi.exe!CalcPi - pi.c:71
- pi.exe!CalcPi - pi.c:67
- pi.exe!main - pi.c:40
- pi.exe!_start - pi.exe:2340

Related Code Location: pi.c:71 - Write

```
68 for (i = iRank; i < n; i += iNumProcs)
69 {
70     fX = fH * ((double)i + 0.5);
71     fSum += f(fX);
72 }
73 return fH * fSum;
74
75
```

Call Stack

- pi.exe!CalcPi - pi.c:71
- pi.exe!CalcPi - pi.c:67
- pi.exe!main - pi.c:40
- pi.exe!_start - pi.exe:2340

Code Locations

ID	Description	Source	Function	Module
X1	Read	pi.c:71	CalcPi	pi.exe
X2	Write	pi.c:71	CalcPi	pi.exe
X3	Write	pi.c:71	CalcPi	pi.exe

Inspector XE – Results

- 1 Source Code producing the issue – double click opens an e
- 2 Corresponding Call Stack

The missing reduction is detected.

The screenshot displays the Intel Inspector XE 2011 interface. The top bar shows the target 'r001ti3'. The main window is divided into several panes. The 'Focus Code Location: pi.c:71 - Write' pane shows a C code snippet with a for loop and a write operation on line 71, which is highlighted with a yellow circle and the number 1. The 'Call Stack' pane on the right shows the call stack for the selected location, with the top frame 'pi.exe!CalcPi - pi.c:71' highlighted with a yellow circle and the number 2. Below the focus code pane is the 'Related Code Location: pi.c:71 - Write' pane, which also shows the same code snippet with line 71 highlighted and a yellow circle and the number 1. To the right of this pane is another 'Call Stack' pane, which also shows the same call stack with the top frame highlighted and a yellow circle and the number 2. At the bottom of the interface is the 'Code Locations' pane, which contains a table with three rows of code locations.

ID	Description	Source	Function	Module
X1	Read	pi.c:71	CalcPi	pi.exe
X2	Write	pi.c:71	CalcPi	pi.exe
X3	Write	pi.c:71	CalcPi	pi.exe

Command Line Tool – inspxe-cl

Threading Error Analysis Modes

1. Detect Deadlocks (ti1)
2. Detect Deadlocks and Data Races (ti2)
3. Locate Deadlocks and Data Races (ti3)

\$ inspxe-cl -collect ti3 -- pi.exe < input

Data collection without GUI allows to use batch jobs.

\$ inspxe-cl -report problems ...

Viewing results in text mode is helpful, when remote connections are slow.

- **Performance Analyses for**

- Serial Applications
- Shared Memory Parallel Applications

- **Sampling-based measurements**

- **Features:**

- Hot Spot Analysis
- Concurrency Analysis
- Wait
- Hardware Performance Counter Support

Prerequisites:

- ssh -Y cluster-linux-tuning
- module load intelvtune

Stream

- Standard Benchmark to measure memory performance.
- Version is parallelized with OpenMP.

Measures Memory bandwidth for:

y=x (copy)

y=s*x (scale)

y=x+z (add)

y=x+s*z (triad)

```
#pragma omp parallel for  
for (j=0; j<N; j++)  
    b[j] = scalar*c[j];
```

for double vectors x,y,z and scalar double value s

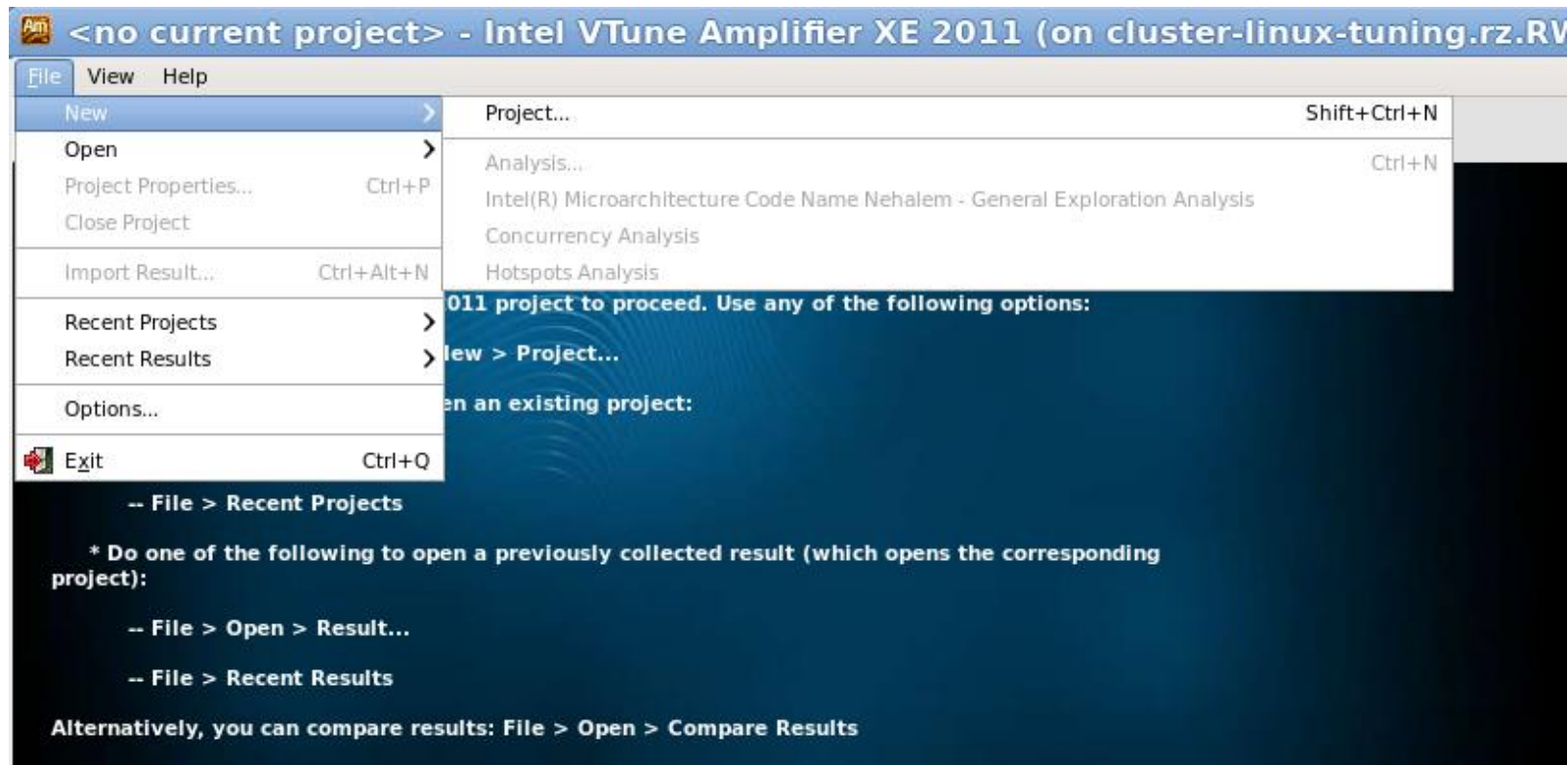
Function	Rate (MB/s)	Avg time	Min time	Max
time				
Copy:	33237.0185	0.0050	0.0048	0.0055
Scale:	33304.6471	0.0049	0.0048	0.0059
Add:	35456.0586	0.0070	0.0068	0.0073
Triad:	36030.9600	0.0069	0.0067	0.0072

Live Demo

Intel VTune Amplifier XE

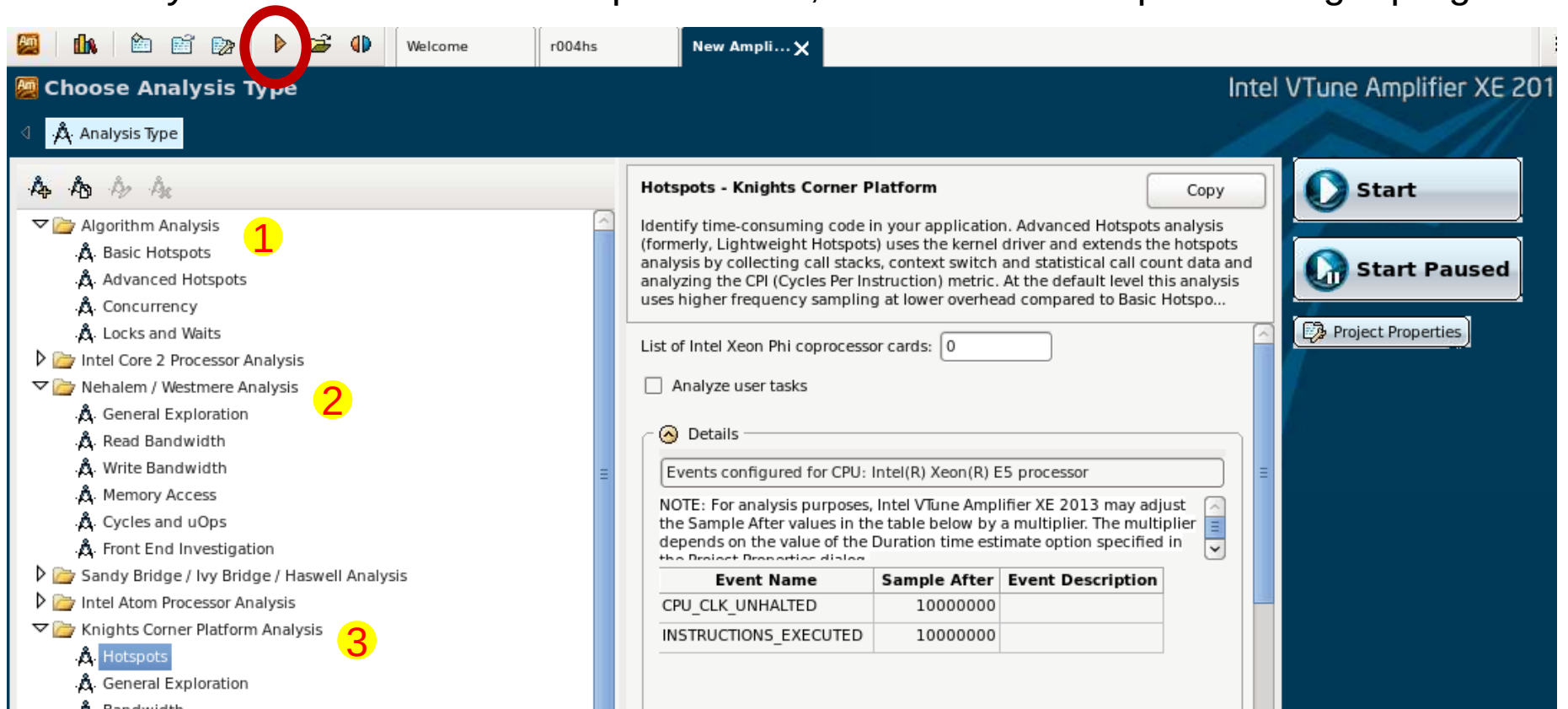
Intel VTune Amplifier XE – Create Project

- Create a Project in the same way as with the inspector.
- Executable should be build with optimization.
- Use a reasonable sized data set.



Amplifier XE – Measurement Runs

- 1 Basic Analysis Types
- 2 Hardware Counter Analysis Types (e.g., Westmere Architecture on cluster-linux-tuning)
- 3 Analysis for Intel Xeon Phi coprocessors, choose this for OpenMP target programs.



Amplifier XE – Hotspot Analysis

The screenshot shows the 'Hotspots - Hotspots' window in Amplifier XE. The window has a menu bar with 'Analysis Target', 'Analysis Type', 'Collection Log', 'Summary', 'Bottom-up', and 'Top-down'. The main content area is divided into three sections: 'Elapsed Time', 'Top Hotspots', and 'Collection and Platform Info'. Each section is marked with a yellow circle containing a number (1, 2, and 3 respectively).

Elapsed Time: 1.294s

Total Thread Count:	12
CPU Time:	14.840s
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically re performance.

Function	CPU Time
main	10.672s
__kmp_wait_sleep	2.438s
__kmp_x86_pause	1.100s
__kmp_execute_tasks	0.400s
__kmp_yield	0.120s
[Others]	0.110s

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Command Line: /rwthfs/rz/cluster/home/ds534486/PPCES2012/stream/stream.exe

Environment Variables: OMP_NUM_THREADS=12;

Frequency: 3.07 GHz

Logical CPU Count: 24

Operating System: Linux

Computer Name: cluster-linux-tuning.rz.RWTH-Aachen.DE

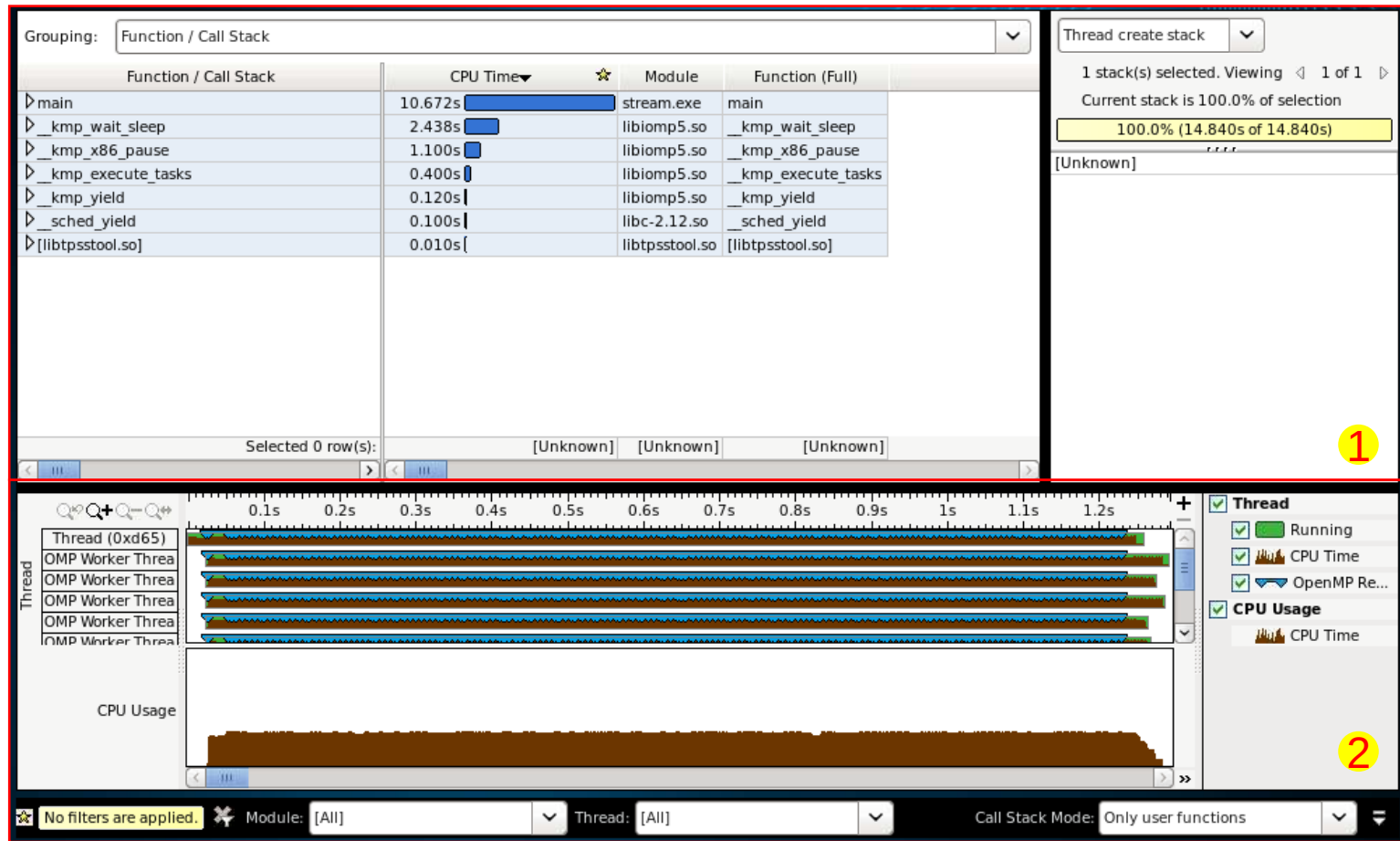
Result Size: 71 KB

Summary:

- 1 General Timing Information
- 2 Top Hotspots
- 3 Platform Information

Amplifier XE – Hotspot Analysis

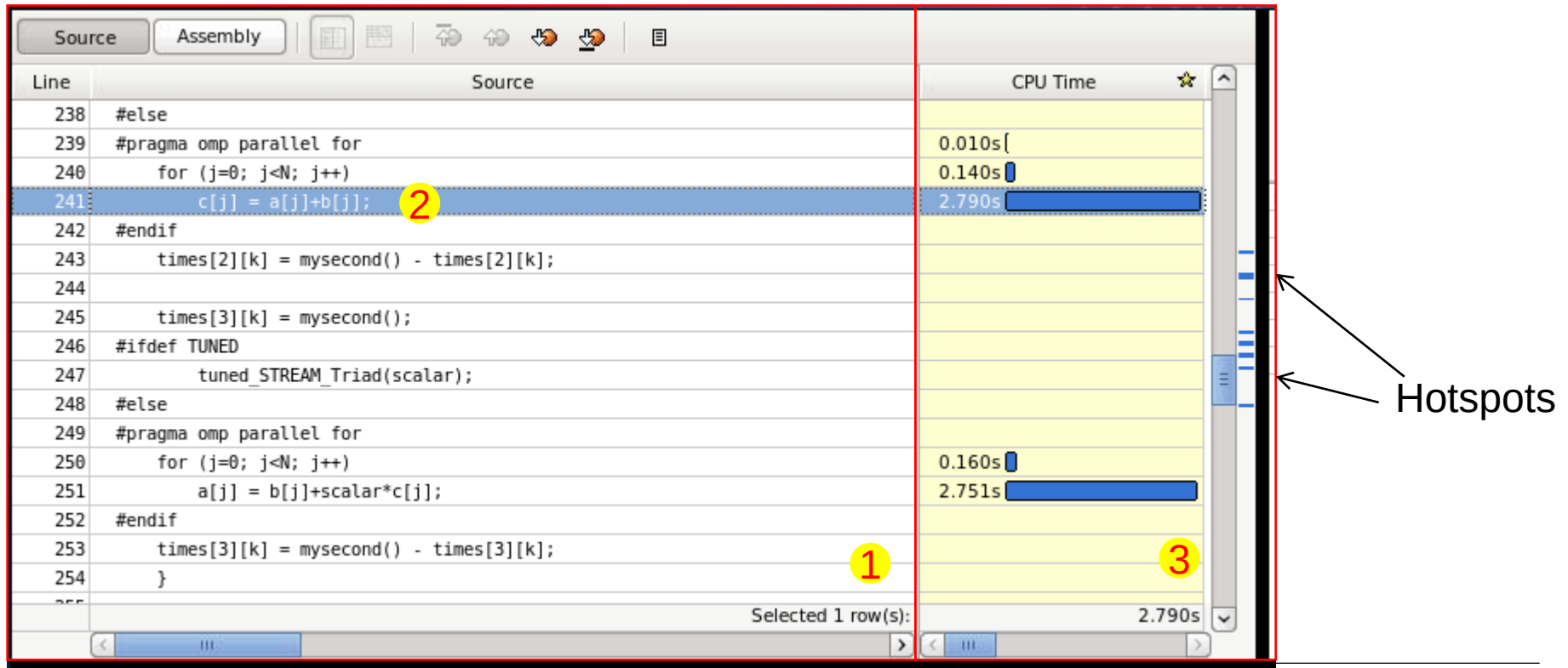
- 1 Function Summary
- 2 Timeline View



Amplifier XE – Hotspot Analysis

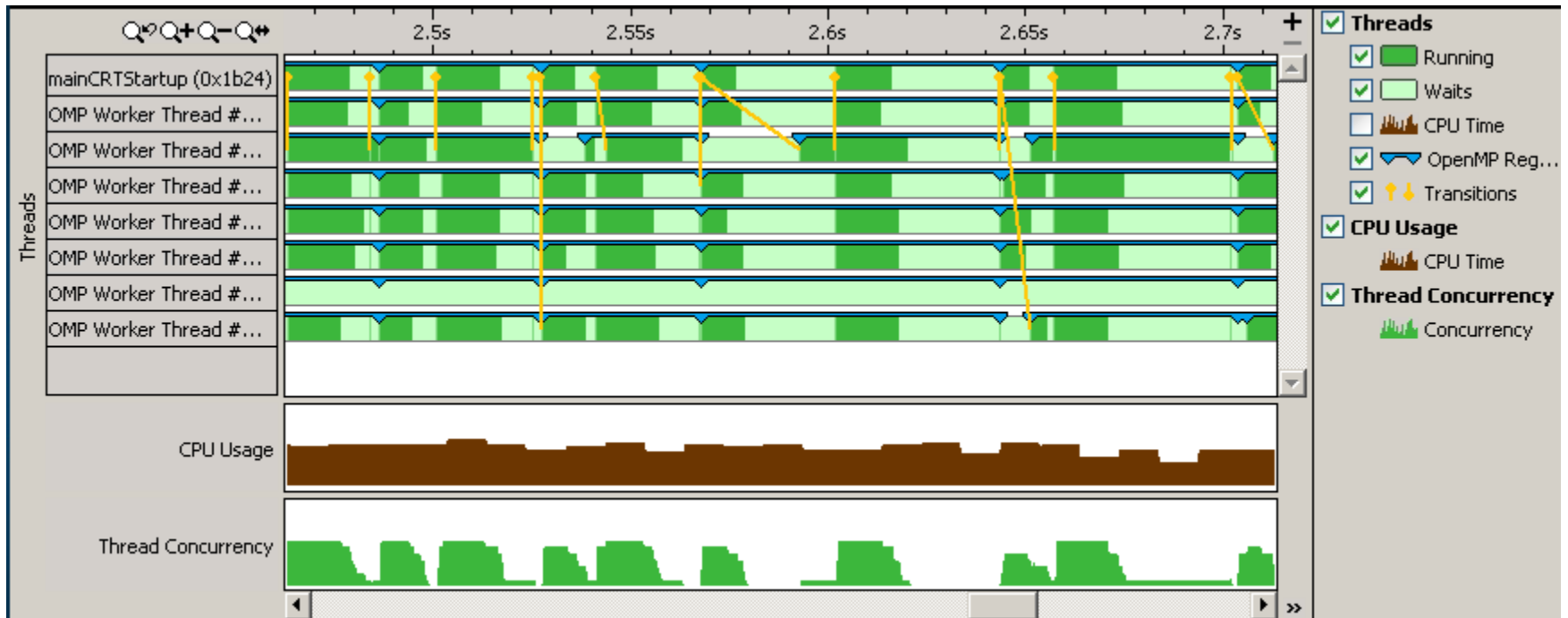
Double clicking on a function opens source code view.

- 1 Source Code View (only if compiled with -g)
- 2 Hotspot: Add Operation of Stream
- 3 Metrics View



Amplifier XE – Locks and Waits Analysis

- Waiting time is shown in light green.
- Execution time is shown in dark green.
- CPU Usage and Thread Concurrency differ because waiting threads utilize a CPU.



Derived Metrics

■ **Clock cycles per Instructions (CPI)**

- CPI indicates if the application is utilizing the CPU or not
- Take care: Doing “something” does not always mean doing “something useful”.

■ **Floating Point Operations per second (FLOPS)**

- How many arithmetic operations are done per second?
- Floating Point operations are normally really computing and for some algorithms the number of floating point operations needed can be determined.

Amplifier XE – Hardware Counter

1 CPI rate (Clock cycles per instruction): In theory modern processors can finish 4 instructions in 1 cycle, so a CPI rate of 0.25 is possible. A value between 0.25 and 1 is often considered as good for HPC applications.

 **Elapsed Time:** 1.872s 

Hardware Event Count: 125,574,000,000

CPU_CLK_UNHALTED.THREAD: 6.3462e+10

INST_RETIRED.ANY: 6.2112e+10

1 CPI Rate: 1.022

The CPI may be too high. This could be caused by issues such as memory access stalls, cache misses, or branch mispredicts. Explore the other hardware-related metrics to identify what is causing the high CPI.

Retire Stalls: 0.570s

A high number of retire stalls is detected. This may result from branch mispredicts or other instructions that cause the processor to stall. Use this metric to find where you have stalled instructions. Once you have identified the source of the stalls, you can optimize your code to reduce the number of stalls.

LLC Miss: 0.013s

LLC Load Misses Serviced By Remote DRAM: 0.001s

Instruction Starvation: 0.098s

Branch Mispredict: 0.001s

Execution Stalls: 0.288s

Summary

Correctness:

- Data Races are very hard to find, since they do not show up every program run.
- Intel Inspector XE helps a lot in finding these errors.
- Use really small datasets, since the runtime explores.
- If possible use non optimized code.

Performance:

- Start with simple performance measurements like hotspots analyses and then focus on these hot spots.
- In OpenMP applications analyze the waiting time of threads. Is the waiting time balanced?
- Hardware counters might help for a better understanding of an application, but they are hard to interpret.