Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/#/en_US_01

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.


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

Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skool, the skool logo, Sound Mark, The Journey Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Portions Copyright (C) 2001, Hewlett-Packard Development Company, L.P.
Chapter 4: Intel(R) VTune(TM) Amplifier XE Tutorials: Troubleshooting
Troubleshooting..........................................................................................51
Introducing the Intel(R) VTune(TM) Amplifier XE

The Intel(R) VTune(TM) Amplifier XE, an Intel(R) Parallel Studio XE tool, provides information on code performance for users developing serial and multithreaded applications on Windows* and Linux* operating systems. On Windows systems, the VTune Amplifier XE integrates into Microsoft Visual Studio* software and is also available as a standalone GUI client. On Linux systems, VTune Amplifier XE works only as a standalone GUI client. On both Windows and Linux systems, you can benefit from using the command-line interface for collecting data remotely or for performing regression testing.

VTune Amplifier XE helps you analyze the algorithm choices and identify where and how your application can benefit from available hardware resources. Use the VTune Amplifier XE to locate or determine the following:

- The most time-consuming (hot) functions in your application and/or on the whole system
- Sections of code that do not effectively utilize available processor time
- The best sections of code to optimize for sequential performance and for threaded performance
- Synchronization objects that affect the application performance
- Whether, where, and why your application spends time on input/output operations
- The performance impact of different synchronization methods, different numbers of threads, or different algorithms
- Thread activity and transitions
- Hardware-related bottlenecks in your code

Intel VTune Amplifier XE Tutorials

These tutorials tell you how to use the VTune Amplifier XE to analyze the performance of a sample application by identifying software- and hardware-related issues in the code.

- Finding Hotspots
- Analyzing Locks and Waits

Prerequisites

You need the following tools, skills, and knowledge to effectively use these tutorials.

Required Tools

You need the following tools to use these tutorials:

- Intel(R) VTune(TM) Amplifier XE
- Sample code shipped with the VTune Amplifier XE
- VTune Amplifier XE Help

To acquire the VTune Amplifier XE:

If you do not already have access to the VTune Amplifier XE, you can download an evaluation copy from http://software.intel.com/en-us/articles/intel-software-evaluation-center/.

To install the VTune Amplifier XE, follow the instructions in the Release Notes.

To install and set up VTune Amplifier XE sample code:

1. Copy the tachyon_vtune_amp_xe.tar.gz file from the samples/<locale> folder in the Intel VTune Amplifier XE installation folder (the default installation folder is /opt/intel/vtune_amplifier_xe_2011) to a writable directory or share on your system.
2. Extract the sample(s) from the .tar file.

NOTE.

- Samples are non-deterministic. Your screens may vary from the screen shots shown throughout these tutorials.
- Samples are designed only to illustrate VTune Amplifier XE features and do not represent best practices for tuning the code. Results may vary depending on the nature of the analysis.

To run the VTune Amplifier XE:

Launch the amplxe-gui script from the /opt/intel/vtune_amplifier_xe_2011/bin32 directory.

To access VTune Amplifier XE Help:

See the Getting Help topic.
Navigation Quick Start

**Standalone Intel(R) VTune(TM) Amplifier XE**

Use the VTune Amplifier XE menu to control result collection, define and view project properties, and set various options.

Use the VTune Amplifier XE toolbar to configure and control result collection.
Use the VTune Amplifier XE result tabs to manage result data. You can view or change the result file location from the **Project Properties** dialog box.

Use the drop-down menu to select a **viewpoint**, a preset configuration of windows/panes for an analysis result. For each analysis type, you can switch among several preset configurations to focus on particular performance metrics. Click the yellow question mark icon to read the viewpoint description.

Switch between window tabs to explore the analysis type configuration options and collected data provided by the selected viewpoint.
Key Terms and Concepts

Key Terms

**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: \textit{Wall clock time at end of application} – \textit{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.

**target**: A target is an executable file you analyze using the Intel(R) VTune(TM) Amplifier XE.

**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 VTune Amplifier XE shows in the windows/panes of the result tab. To select the required viewpoint, use the drop-down menu 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 Usage

For the user-mode sampling and tracing analysis types, the Intel(R) VTune(TM) Amplifier XE identifies a processor utilization scale, calculates the target CPU usage, 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.

<table>
<thead>
<tr>
<th>Utilization Type</th>
<th>Default Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td></td>
<td>All CPUs are waiting - no threads are running.</td>
</tr>
<tr>
<td>Poor</td>
<td></td>
<td>Poor usage. By default, poor usage is when the number of simultaneously running CPUs is less than or equal to 50% of the target CPU usage.</td>
</tr>
<tr>
<td>OK</td>
<td></td>
<td>Acceptable (OK) usage. By default, OK usage is when the number of simultaneously running CPUs is between 51-85% of the target CPU usage.</td>
</tr>
</tbody>
</table>
Ideal usage. By default, Ideal usage is when the number of simultaneously running CPUs is between 86-100% of the target CPU usage.

<table>
<thead>
<tr>
<th>Utilization Type</th>
<th>Default color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ideal</td>
<td></td>
<td>Ideal usage. By default, Ideal usage is when the number of simultaneously running CPUs is between 86-100% of the target CPU usage.</td>
</tr>
</tbody>
</table>

**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 Data of Interest column by right-clicking the required column and selecting the Set Column as Data of Interest command from the pop-up menu.

**Key Concept: Finalization**

Finalization is the process of the Intel(R) VTune(TM) Amplifier XE converting the collected data to a database, resolving symbol information, and pre-computing data to make further analysis more efficient and responsive. The VTune Amplifier XE finalizes data automatically when data collection completes.

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

**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%) user-mode sampling and tracing collection that gets you the information you need without slowing down the application execution significantly.

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: Thread Concurrency

The number of active threads corresponds to the concurrency level of an application. By comparing the concurrency level with the number of processors, Intel(R) VTune(TM) Amplifier XE classifies how an application utilizes the processors in the system. It defines default utilization ranges depending on the number of processor cores and displays the thread concurrency in the Summary and Bottom-up window. You can change the utilization ranges by dragging the slider in the Summary window.

Thread concurrency may be higher than CPU Usage if threads are in the runnable state and not consuming CPU time. VTune Amplifier XE defines the Target Concurrency level for your application that is, by default, equal to the number of physical cores.

<table>
<thead>
<tr>
<th>Utilization Type</th>
<th>Default color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td></td>
<td>All threads in the application are waiting - no threads are running. There can be only one node in the Thread Concurrency chart indicating Idle utilization.</td>
</tr>
<tr>
<td>Poor</td>
<td></td>
<td>Poor utilization. By default, poor utilization is when the number of threads is up to 50% of the target concurrency.</td>
</tr>
<tr>
<td>OK</td>
<td></td>
<td>Acceptable (OK) utilization. By default, OK utilization is when the number of threads is between 51-85% of the target concurrency.</td>
</tr>
<tr>
<td>Ideal</td>
<td></td>
<td>Ideal utilization. By default, ideal utilization is when the number of threads is between 86-115% of the target concurrency.</td>
</tr>
<tr>
<td>Utilization Type</td>
<td>Default color</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Over</td>
<td>☢</td>
<td>Over-utilization. By default, over-utilization is when the number of threads is more than 115% of the target concurrency.</td>
</tr>
</tbody>
</table>
Tutorial: Finding Hotspots

Learning Objectives

This tutorial shows how to use the Hotspots analysis of the Intel(R) VTune(TM) Amplifier XE to understand where the sample application is spending time, identify hotspots - the most time-consuming program units, and detect how they were called. Some hotspots may indicate bottlenecks that can be removed, while other hotspots are inevitable and take a long time to execute due to their nature. Typically, the hotspot functions identified during the Hotspots analysis use the most time-consuming algorithms and are good candidates for parallelization. The Hotspots analysis is useful to analyze the performance of both serial and parallel applications.

Estimated completion time: 15 minutes.

After you complete this tutorial, you should be able to:

- Choose an analysis target.
- Choose the Hotspots analysis type.
- Run the Hotspots analysis to locate most time-consuming functions in an application.
- Analyze the function call flow and threads.
- Analyze the source code to locate the most time-critical code lines.
- Compare results before and after optimization.

Workflow Steps to Identify and Analyze Hotspots

You can use the Intel(R) VTune(TM) Amplifier XE to identify and analyze hotspot functions in your serial or parallel application by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample ray-tracer application named tachyon.
1. Build your target.
2. Specify the target to analyze for hotspots in the VTune Amplifier XE project.
3. Choose and run the Hotspots analysis.
4. Interpret the result data.
5. View and analyze code of the performance-critical function.
6. Modify the code to tune the algorithms.
7. Re-build the target, re-run the Hotspots analysis, and compare the result data before and after optimization.

**Build Target**

**Amplifier Tips**
Before you start, you need to do the following to ensure the Intel(R) VTune(TM) Amplifier XE provides the most accurate information on the performance of your application:

- **Build the target in the release mode**, which is recommended for performance analysis.
- **Run the application without debugging to create a performance baseline.**

**Choose a Build Mode and Build a Target**

1. Browse to the directory where you extracted the sample code (for example, `/home/intel/samples/tachyon_vtune_amp_xe`). Make sure this directory contains Makefile.
2. Clean up all the previous builds as follows:
   
   
   "$ make clean"

3. Build your target in the release mode as follows:
   
   "$ make release"
The tachyon_find_hotspots application is built.

Create a Performance Baseline

1. Run tachyon_find_hotspots with dat/balls.dat as an input parameter. For example:
   
   $ /home/intel/samples/tachyon_vtune_amp_xe/tachyon_find_hotspots dat/balls.dat

   The tachyon_find_hotspots application starts running.

   NOTE. Before you start the application, minimize the amount of other software running on your computer to get more accurate results.

2. Note the execution time displayed in the window caption or in the shell window. For the tachyon_find_hotspots executable in the figure above, the execution time is 83.539 seconds. The total execution time is the baseline against which you will compare subsequent runs of the application.
NOTE. Run the application several times, note the execution time for each run, and use the average number. This helps to minimize skewed results due to transient system activity.

Recap

You built the target for the Hotspots analysis in the Release mode, and created the performance baseline. Your application is ready for analysis.

Key Terms and Concepts

- Term: target, baseline

Choose Target

After you built the target, launch the Intel(R) VTune(TM) Amplifier XE GUI client and specify the analysis target in the VTune Amplifier XE project. For this tutorial, your target is a ray-tracer application, tachyon. To learn how to install and set up the sample code, see Prerequisites.

To choose a target:

1. Set the EDITOR or VISUAL environment variable to associate your source files with the code editor (like emacs, vi, vim, gedit, and so on). For example:

   ```
   $ export EDITOR=gedit
   ```

2. From the `<install_dir>/bin32` directory (for IA-32 architecture) or from the `<install_dir>/bin64` directory (for Intel(R) 64 architecture), run the `amplxe-gui` script launching the VTune Amplifier XE GUI client.

   By default, the `<install_dir>` is `/opt/intel/vtune_amplifier_xe_2011`.

3. Create a new project via File > New > Project....

   The Create a Project dialog box opens.

4. Specify the project name `tachyon` that will be used as the project directory name.

   The VTune Amplifier XE creates the `tachyon` project directory under the `root/intel/My Amplifier XE Projects` directory and opens the Project Properties: Target dialog box.

5. In the Application to Launch pane of the Target tab, specify and configure your target as follows:

   - For the Application field, browse to: `<sample_code_dir>/tachyon_find_hotspots`, for example: `/home/intel/samples/en/tachyon_vtune_amp_xe/tachyon_find_hotspots`.
   - For the Application parameters field, enter `dat/balls.dat`. 
6. Click **OK** to apply the settings and exit the **Project Properties** dialog box.

**Recap**

You created the `tachyon_find_hotspots` project as the target for the Hotspots analysis.

**Key Terms and Concepts**

- Term: **target**
- Concept: **Hotspots Analysis**

**Run Hotspots Analysis**

**Note**: Before running an analysis, choose a configuration level to influence Intel(R) VTune(TM) Amplifier XE analysis scope and running time. In this tutorial, you run the Hotspots analysis to identify the hotspots that took much time to execute.

**To run an analysis:**

1. From the VTune Amplifier XE toolbar, click the **New Analysis** button.
   
   The VTune Amplifier XE result tab opens with the **Analysis Type** window active.

2. On the left pane of the **Analysis Type** window, locate the analysis tree and select **Algorithm Analysis > Hotspots**.
   
   The right pane is updated with the default options for the Hotspots analysis.

3. Click the **Start** button on the right command bar.
VTune Amplifier XE launches the `tachyon_find_hotspots` application that renders `balls.dat` as an input file, calculates the execution time, and exits. VTune Amplifier XE finalizes the collected results and opens the Hotspots viewpoint.

**NOTE.** To make sure the performance of the application is repeatable, go through the entire tuning process on the same system with a minimal amount of other software executing.

**Recap**

You launched the Hotspots data collection that analyzes function calls and CPU time spent in each program unit of your application.

**NOTE.** This tutorial explains how to run an analysis from the VTune Amplifier XE graphical user interface (GUI). You can also use the VTune Amplifier XE command-line interface (`amplxe-cl` command) to run an analysis. For more details, check the Command-line Interface Support section of the VTune Amplifier XE Help.

**Key Terms and Concepts**

- **Term:** hotspot, Elapsed time, viewpoint
- **Concept:** Hotspot Analysis, Finalization

**Interpret Result Data**

When the sample application exits, the Intel(R) VTune(TM) Amplifier XE finalizes the results and opens the Hotspots viewpoint that consists of the Summary, Bottom-up, and Top-down Tree windows. To interpret the data on the sample code performance, do the following:

- Understand the basic performance metrics provided by the Hotspots analysis.
- Analyze the most time-consuming functions.
- Analyze CPU usage per function.
NOTE. The screenshots and execution time data provided in this tutorial are created on a system with two CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

Understand the Basic Hotspots Metrics

Click the Summary tab and review the data provided in the Summary window for the whole application performance. The Summary window provides pop-up help for the basic performance metrics. Hover over the yellow question mark icon to read the help.

**Elapsed Time:** 89.876s

- **CPU Time:** 79.543s
- **Total Thread Count:** 1

**Top Hotspots**
This section lists the most active for improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize_2D_buffer</td>
<td>52.939s</td>
</tr>
<tr>
<td>grid_intersect</td>
<td>13.885s</td>
</tr>
<tr>
<td>sphere_intersect</td>
<td>10.361s</td>
</tr>
<tr>
<td>pos2grid</td>
<td>0.470s</td>
</tr>
<tr>
<td>Raytrt</td>
<td>0.319s</td>
</tr>
<tr>
<td>[Others]</td>
<td>1.569s</td>
</tr>
</tbody>
</table>

Note that **CPU Time** for the sample application is equal to 89.876 seconds. It is the sum of CPU time for all application threads. **Total Thread Count** is 1, so the sample application is single-threaded.

The **Top Hotspots** section provides data on the most time-consuming functions (hotspot functions) sorted by CPU time spent on their execution. For the sample application, the initialize_2D_buffer function, which took 52.939 seconds to execute, shows up at the top of the list as the hottest function.

The [Others] entry at the bottom shows the sum of CPU time for all functions not listed in the table.

Analyze the Most Time-consuming Functions

Click the Bottom-up tab to explore the Bottom-up pane. Analyze the **CPU Time** column values. This column is marked with a yellow star as the Data of Interest column. It means that the VTune Amplifier XE uses this type of data for some calculations (for example, filtering, stack contribution, and others). Functions that took most CPU time to execute are listed on top.

The initialize_2D_buffer function took 52.939 seconds to execute. Click the arrow sign at the initialize_2D_buffer function to expand the stacks calling this function. You see that it was called only by the setup_2D_buffer function.
Select the `initialize_2D_buffer` function in the grid and explore the data provided in the Call Stack pane on the right.

The Call Stack pane displays full stack data for each hotspot function, enables you to navigate between function call stacks and understand the impact of each stack to the function CPU time. The stack functions in the Call Stack pane are represented in the following format:

```
<module>!<function> - <file>:<line number>
```

where the line number corresponds to the line calling the next function in the stack.

For the sample application, the hottest function is called at line 87 of the `setup_2D_buffer` function in the `global.cpp` file.

### Analyze CPU Usage per Function

VTune Amplifier XE enables you to analyze the collected data from different perspectives by using multiple viewpoints.

For the Hotspots analysis result, you may switch to the **Hotspots by CPU Usage** viewpoint to understand how your hotspot function performs in terms of the CPU usage. Explore this viewpoint to determine how your application utilized available cores and identify the most serial code.

If you go back to the Summary window, you can see the **CPU Usage Histogram** that represents the Elapsed time and usage level for the available logical processors.

The `tachyon_find_hotspots` application ran mostly on one logical CPU. If you hover over the highest bar, you see that it spent 79.695 seconds using one core only, which is classified by the VTune Amplifier XE as a Poor utilization for a dual-core system. To understand what prevented the application from using all available logical CPUs effectively, explore the Bottom-up pane.

To get the detailed CPU usage information per function, use the button in the Bottom-up window to expand the **CPU Time** column.

Note that `initialize_2D_buffer` is the function with the longest poor CPU utilization (red bars). This means that the processor cores were underutilized most of the time spent on executing this function.
If you change the grouping level (highlighted in the figure above) in the Bottom-up pane from /Function/Call Stack to /Thread/Function/Call Stack, you see that the initialize_2D_buffer function belongs to the thread_video thread. This thread is also identified as a hotspot and shows up at the top in the Bottom-up pane. To get detailed information on the hotspot thread performance, explore the Timeline pane.

The Timeline analysis also identifies the thread_video thread as the most active. The tooltip shows that CPU time values are about 100% whereas the maximum CPU time value for dual-core systems is 200%. This means that the processor cores were half-utilized for most of the time spent on executing the tachyon_find_hotspots application.

Recap
You identified a function that took the most CPU time and could be a good candidate for algorithm tuning.

Key Terms and Concepts
- Term: Elapsed time, CPU time, viewpoint
- Concept: Hotspots Analysis, CPU Usage, Data of Interest
Analyze Code

You identified `initialize_2D_buffer` as the hottest function. In the Bottom-up pane, double-click this function to open the Source window and analyze the source code:

- Understand basic options provided in the Source window.
- Identify the hottest code lines.

Understand Basic Source Window Options

The table below explains some of the features available in the Source window when viewing the Hotspots analysis data.

1. **Source pane** displaying the source code of the application if the function symbol information is available. The code line that took the most CPU time to execute is highlighted. The source code in the Source pane is not editable.

   If the function symbol information is not available, the Assembly pane opens displaying assembler instructions for the selected hotspot function. To enable the Source pane, make sure to build the target properly.

2. **Assembly pane** displaying the assembler instructions for the selected hotspot function. Assembler instructions are grouped by basic blocks. The assembler instructions for the selected hotspot function are highlighted. To get help on an assembler instruction, right-click the instruction and select Instruction Reference.

The table below explains some of the features available in the Source window when viewing the Hotspots analysis data.

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xa272</td>
<td>lea (%edx,%eax,4), %ecx</td>
<td>0.010s</td>
</tr>
<tr>
<td>0xa275</td>
<td>jmp 0x805228a &lt; Z20initialize_2D</td>
<td>0.090s</td>
</tr>
<tr>
<td>0xa277</td>
<td><em>Z20initialize_2D_bufferPJs</em>+0x27</td>
<td></td>
</tr>
<tr>
<td>0xa277</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0xa278</td>
<td><em>Z20initialize_2D_bufferPJs</em>+0x28</td>
<td></td>
</tr>
<tr>
<td>0xa278</td>
<td>add $0x3, %esi</td>
<td>0.010s</td>
</tr>
<tr>
<td>0xa27b</td>
<td>add $0x870, %ecx</td>
<td>0.612s</td>
</tr>
<tr>
<td>0xa281</td>
<td>inc %ebx</td>
<td>0.428s</td>
</tr>
<tr>
<td>0xa282</td>
<td>cmp $0xb3, %esi</td>
<td></td>
</tr>
<tr>
<td>0xa288</td>
<td>jne 0x80522af &lt; Z20initialize_2D</td>
<td></td>
</tr>
<tr>
<td>0xa28a</td>
<td><em>Z20initialize_2D_bufferPJs</em>+0x3a</td>
<td></td>
</tr>
<tr>
<td>0xa28a</td>
<td>movl 0xc(%ebp), %edx</td>
<td></td>
</tr>
<tr>
<td>0xa28d</td>
<td>movl (%edx), %eax</td>
<td>1.602s</td>
</tr>
</tbody>
</table>

The Source pane displays the source code of the application if the function symbol information is available. The code line that took the most CPU time to execute is highlighted. The source code in the Source pane is not editable.

If the function symbol information is not available, the Assembly pane opens displaying assembler instructions for the selected hotspot function. To enable the Source pane, make sure to build the target properly.

Assembly pane displaying the assembler instructions for the selected hotspot function. Assembler instructions are grouped by basic blocks. The assembler instructions for the selected hotspot function are highlighted. To get help on an assembler instruction, right-click the instruction and select Instruction Reference.
NOTE. To get the help on a particular instruction, make sure to have the Adobe Acrobat Reader* 9 (or later) installed. If an earlier version of the Adobe Acrobat Reader is installed, the Instruction Reference opens but you need to locate the help on each instruction manually.

Processor time attributed to a particular code line. If the hotspot is a system function, its time, by default, is attributed to the user function that called this system function.

Source window toolbar. Use the hotspot navigation buttons to switch between most performance-critical code lines. Hotspot navigation is based on the metric column selected as a Data of Interest. For the Hotspots analysis, this is CPU Time. Use the Source/Assembly buttons to toggle the Source/Assembly panes (if both of them are available) on/off.

Heat map markers to quickly identify performance-critical code lines (hotspots). The bright blue markers indicate hot lines for the function you selected for analysis. Light blue markers indicate hot lines for other functions. Scroll to a marker to locate the hot code line it identifies.

**Identify the Hottest Code Lines**

When you identify a hotspot in the serial code, you can make some changes in the code to tune the algorithms and speed up that hotspot. Another option is to parallelize the sample code by adding threads to the application so that it performs well on multi-core processors. This tutorial focuses on algorithm tuning.

By default, when you double-click the hotspot in the Bottom-up pane, VTune Amplifier XE opens the source file related to this function. For the initialize_2D_buffer function, the hottest code line is 121. This code is used to initialize a memory array using non-sequential memory locations. Click the Source Editor button on the Source window toolbar to open the default code editor and work on optimizing the code.

**Recap**

You identified the code section that took the most CPU time to execute.

**Key Terms and Concepts**

- **Term:** hotspot, CPU time
- **Concept:** Hotspots Analysis, Data of Interest

**Tune Algorithms**

In the Source window, you identified that in the initialize_2D_buffer hotspot function the code line 121 took the most CPU time. Focus on this line and do the following:

- Open the code editor.
- Optimize the algorithm used in this code section.
Open the Code Editor

Click the Source Editor button to open the initbuffer.cpp file in the default code editor:

```c
107 // First (slower) method of filling array
108 // Array is NOT filled in consecutive memory address order
109 //*******************************************************************************
110 for (int i = 0; i < mem_array_i_max; i++)
111 {
112 //*******************************************************************************
113 // Try to defeat hardware prefetching by varying the stride
114 int j(0), iteration_count(0);
115
117 do {
118   mem_array[j*mem_array_i_max+i] = *fill_value + 2;
119 // Code to give the array accesses a non-uniform stride to defeat hardware prefetch
120   if ((iteration_count % 3) == 0) j+=3;
121     else j+=2;
122     iteration_count++;
123   } while (j < mem_array_j_max);
124 }
125 //*******************************************************************************
126 //*******************************************************************************
127 // Faster method of filling array
128 // The for loops are interchanged
129 // Array IS filled in consecutive memory address order
130 //*******************************************************************************
131 for (int j = 0; j < mem_array_j_max; j++)
132 {
133   for (int i = 0; i < mem_array_i_max; i++)
134     {  
135       mem_array[j*mem_array_i_max+i] = *fill_value + 2;
136     }
137   }
```

Hotspot line is used to initialize a memory array using non-sequential memory locations. For demonstration purposes, the code lines are commented as a slower method of filling the array.

Resolve the Problem

To resolve this issue, optimize your algorithm as follows:

1. Edit lines 110 and 113 to comment out code lines 111-125 marked as a "First (slower) method".
2. Edit line 144 to uncomment code lines 145-151 marked as a "Faster method". In this step, you interchange the for loops to initialize the code in sequential memory locations.

3. Save the changes made in the source file.

4. Browse to the directory you extracted the sample code (for example, /home/intel/samples/en/tachyon_vtune_amp_xe).

5. Rebuild your target in the release mode using the `make` command as follows:
   ```
   $ make clean
   $ make release
   ```
   The `tachyon_find_hotspots` application is rebuilt and stored in the `tachyon_vtune_amp_xe` directory.

6. Run `tachyon_find_hotspots` as follows:
   ```
   /home/intel/samples/en/tachyon_vtune_amp_xe/tachyon_find_hotspots dat/balls.dat
   ```
   System runs the `tachyon_find_hotspots` application. Note that execution time reduced from 83.539 seconds to 43.760 seconds.
Recap
You interchanged the loops in the hotspot function, rebuilt the application, and got performance gain of 40 seconds.

Key Terms and Concepts
- Term: hotspot

Compare with Previous Result
You optimized your code to apply a loop interchange mechanism that gave you 40 seconds of improvement in the application execution time. To understand whether you got rid of the hotspot and what kind of optimization you got per function, re-run the Hotspots analysis on the optimized code and compare results:
- Compare results before and after optimization.
- Identify the performance gain.

Compare Results Before and After Optimization
1. Run the Hotspots analysis on the modified code.
2. Click the button on the Intel(R) VTune(TM) Amplifier XE toolbar.
   - The Compare Results window opens.
3. Specify the Hotspots analysis results you want to compare and click the Compare button.

The Hotspots Bottom-up window opens, showing the CPU time usage across the two results and the differences side by side.
Difference in CPU time between the two results in the following format: \[ \text{Difference CPUTime} = \text{Result1 CPUTime} - \text{Result2 CPUTime}. \]

CPU time for the initial version of the `tachyon_find_hotspots` application.

CPU time for the optimized version of the `tachyon_find_hotspots` application.

**Identify the Performance Gain**

Compare CPU time data for the first hotspot: CPU Time: r001hs - CPU Time: r002hs = CPU Time: Difference. 52.939s - 11.971s = 40.968s, which means that you got the optimization of ~41 seconds for the `initialize_2D_buffer` function.

**Recap**

You ran the Hotspots analysis on the optimized code and compared the results before and after optimization using the Compare mode of the VTune Amplifier XE. Compare analysis results regularly to look for regressions and to track how incremental changes to the code affect its performance. You may also want to use the VTune Amplifier XE command-line interface and run the `amplxe-cl` command to test your code for regressions. For more details, see the Command-line Interface Support section in the VTune Amplifier XE online help.

**Key Terms and Concepts**

- Term: hotspot, CPU time
- Concept: Hotspots Analysis
Summary

You have completed the Finding Hotspots tutorial. Here are some important things to remember when using the Intel(R) VTune(TM) Amplifier XE to analyze your code for hotspots:

Step 1. Choose and Build Your Target

- Create a performance baseline to compare the application versions before and after optimization. Make sure to use the same workload for each application run.
- Create a VTune Amplifier XE project and use the Project Properties: Target tab to choose and configure your analysis target.

Step 2. Run Analysis

- Use the Analysis Type configuration window to choose, configure, and run the analysis. For example, you may limit the data collection to a predefined amount of data or enable the VTune Amplifier XE to collect more accurate CPU time data. You can also run the analysis from command line using the amplxe-cl command.

Step 3. Interpret Results and Resolve the Issue

- Start analyzing the performance of your application from the Summary window to explore the performance metrics for the whole application. Then, move to the Bottom-up window to analyze the performance per function. Focus on the hotspots - functions that took the most CPU time. By default, they are located at the top of the table.
- Double-click the hotspot function in the Bottom-up pane or Call Stack pane to open its source code at the code line that took the most CPU time.

Step 4. Compare Results Before and After Optimization

- Perform regular regression testing by comparing analysis results before and after optimization. From GUI, click the Compare Results button on the VTune Amplifier XE toolbar. From command line, use the amplxe-cl command.
Tutorial: Analyzing Locks and Waits

Learning Objectives

This tutorial shows how to use the Locks and Waits analysis of the Intel(R) VTune(TM) Amplifier XE to identify one of the most common reasons for an inefficient parallel application - threads waiting too long on synchronization objects (locks) while processor cores are underutilized. Focus your tuning efforts on objects with long waits where the system is underutilized.

Estimated completion time: 15 minutes.

After you complete this tutorial, you should be able to:

• Choose an analysis target.
• Choose the Locks and Waits analysis type.
• Run the Locks and Waits analysis.
• Identify the synchronization objects with long waits and poor CPU utilization.
• Analyze the source code to locate the most critical code lines.
• Compare results before and after optimization.

Workflow Steps to Identify Locks and Waits

You can use the Intel(R) VTune(TM) Amplifier XE to understand the cause of the ineffective processor utilization by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample ray-tracer application named tachyon.
1. Build the target.
2. Choose a target to analyze for locks and waits in the VTune Amplifier XE project.
3. Run the Locks and Waits analysis.
4. Interpret the result data.
5. View and analyze code of the performance-critical function.
6. Modify the code to remove the lock.
7. Re-build the target, re-run the Locks and Waits analysis, and compare the result data before and after optimization.

**Build Target**

Before you start, you need to do the following to ensure the Intel(R) VTune(TM) Amplifier XE provides the most accurate information on the performance of your application:

- Build the target in the release mode, which is recommended for performance analysis.
- Run the application without debugging to create a performance baseline.

**Choose a Build Mode and Build a Target**

1. Browse to the directory where you extracted the sample code (for example, /home/intel/samples/en/tachyon_vtune_amp_xe). Make sure this directory contains Makefile.
2. Clean up all the previous builds using the following command:
   ```bash
   $ make clean
   ```
3. Build your target in the release mode using the following command:
$ make release

The `tachyon_analyze_locks` application is built and stored in the `tachyon_vtune_amp_xe` directory.

**Create a Performance Baseline**

1. Run `tachyon_analyze_locks` with `dat/balls.dat` as an input parameter. For example:
   `/home/intel/sampleen/tachyon_vtune_amp_xe/tachyon_analyze_locks dat/balls.dat`
   
The `tachyon_analyze_locks` application runs in multiple sections (depending on the number of CPUs in your system).

   **NOTE.** Before you start the application, minimize the amount of other software running on your computer to get more accurate results.

2. Note the execution time displayed in the window caption and in the shell window. For the `tachyon_analyze_locks` executable in the figure above, the execution time is 29.647 seconds. The total execution time is the baseline against which you will compare subsequent runs of the application.
NOTE. Run the application several times, note the execution time for each run, and use the average number. This helps to minimize skewed results due to transient system activity.

Recap
You built the target in the Release mode, and created the performance baseline. Your application is ready for analysis.

Key Terms and Concepts
- Term: target, baseline
- Concept: Locks and Waits Analysis

Choose Target

After you built the target, launch the Intel(R) VTune(TM) Amplifier XE GUI client and specify the analysis target in the VTune Amplifier XE project. For this tutorial, your target is a ray-tracer application, tachyon. To learn how to install and set up the sample code, see Prerequisites.

To choose a target:
1. Set the EDITOR or VISUAL environment variable to associate your source files with the code editor (like emacs, vi, vim, gedit, and so on). For example:
   
   `$ export EDITOR=gedit`

2. From the `<install_dir>/bin32` directory (for IA-32 architecture) or from the `<install_dir>/bin64` directory (for Intel(R) 64 architecture), run the `amplxe-gui` script launching the VTune Amplifier XE GUI client.
   
   By default, the `<install_dir>` is `/opt/intel/vtune_amplifier_xe_2011`.

3. Create a new project via File > New > Project....
   
   The Create a Project dialog box opens.

4. Specify the project name tachyon that will be used as the project directory name.
   
   The VTune Amplifier XE creates a project directory under the `root/intel/My Amplifier XE Projects` directory and opens the Project Properties: Target dialog box.

5. In the Application to Launch pane of the Target tab, specify and configure your target as follows:
   
   - For the Application field, browse to: `<tachyon_dir>/tachyon_analyze_locks` (for example, /home/intel/samples/tachyon_vtune_amp_xe/tachyon_analyze_locks).
   
   - For the Application parameters field, specify dat/balls.dat.
6. Click OK to apply the settings and exit the Project Properties dialog box.

Recap

You selected the tachyon_analyze_locks project as the target for the Locks and Waits analysis.

Key Terms and Concepts

- Term: target

Run Locks and Waits Analysis

Before running an analysis, choose a configuration level to define the Intel(R) VTune(TM) Amplifier XE analysis scope and running time. In this tutorial, you run the Locks and Waits analysis to identify synchronization objects that caused contention and fix the problem in the source.

To run an analysis:

1. From the VTune Amplifier XE toolbar, click the New Analysis button.
   The VTune Amplifier XE result tab opens with the Analysis Type window active.
2. From the analysis tree on the left, select Algorithm Analysis > Locks and Waits.
   The right pane is updated with the default options for the Locks and Waits analysis.
3. Click the Start button on the right command bar.
The VTune Amplifier XE launches the tachyon_analyze_locks executable that renders balls.dat as an input file, calculates the execution time, and exits. The VTune Amplifier XE finalizes the collected data and opens the results in the Locks and Waits viewpoint.

**NOTE.** To make sure the performance of the application is repeatable, go through the entire tuning process on the same system with a minimal amount of other software executing.

**Recap**

You ran the Locks and Waits data collection that analyzes how long the application had to wait on each synchronization object, or on blocking APIs, such as sleep() and blocking I/O, and estimates processor utilization during the wait.

**NOTE.** This tutorial explains how to run an analysis from the VTune Amplifier XE graphical user interface (GUI). You can also use the VTune Amplifier XE command-line interface (amplxe-cl command) to run an analysis. For more details, check the Command-line Interface Support section of the VTune Amplifier XE Help.

**Key Terms and Concepts**

- Term: viewpoint
- Concept: Locks and Waits Analysis, Finalization

**Interpret Result Data**

When the sample application exits, the Intel(R) VTune(TM) Amplifier XE finalizes the results and opens the Locks and Waits viewpoint that consists of the Summary window, Bottom-up pane, Top-down Tree pane, Call Stack pane, and Timeline pane. To interpret the data on the sample code performance, do the following:

- Analyze summary statistics
- Identify locks
NOTE. The screenshots and execution time data provided in this tutorial are created on a system with two CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

### Analyze Summary Statistics

Click the **Summary** tab and start with exploring the data provided in the Summary window for the whole application performance. The Summary window provides pop-up help for the basic performance metrics. Hover over the yellow question mark icon to read the help.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>39.089s</td>
</tr>
<tr>
<td>Wait Time</td>
<td>48.435s</td>
</tr>
<tr>
<td>Wait Count</td>
<td>2,347</td>
</tr>
<tr>
<td>CPU Time</td>
<td>28.680s</td>
</tr>
<tr>
<td>Total Thread Count</td>
<td>2</td>
</tr>
<tr>
<td>Spin Time</td>
<td>0.010s</td>
</tr>
</tbody>
</table>

The Result Summary section provides data on the overall application performance per the following metrics:

- **Elapsed Time** is the total time for each core when it was either waiting or not utilized by the application.
- **Wait Time** is the amount of time the application threads waited for some event to occur, such as synchronization waits and I/O waits.
- **Wait Count** is the overall number of times the system wait API was called for the analyzed application.

- **CPU Time** is the sum of CPU time for all threads.
- **Total Thread Count** is the number of threads in the application.
- **Spin Time** is the Wait time during which the CPU is busy.

#### Top Waiting Objects

This section lists the objects that spent the most time in sleep() or I/O, or on contended synchronization object reflects high contention for that object, and, in the case of a synchronization object, high contention for the lock.

<table>
<thead>
<tr>
<th>Sync Object</th>
<th>Wait Time</th>
<th>Wait Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex 0x02d2bb60</td>
<td>27.659s</td>
<td>508</td>
</tr>
<tr>
<td>TBB Scheduler</td>
<td>10.001s</td>
<td>2</td>
</tr>
<tr>
<td>Unknown 0x04a1e656</td>
<td>9.945s</td>
<td>834</td>
</tr>
<tr>
<td>select</td>
<td>0.794s</td>
<td>528</td>
</tr>
<tr>
<td>Stream 0x2d41a05c</td>
<td>0.036s</td>
<td>468</td>
</tr>
<tr>
<td>Others</td>
<td>0.000s</td>
<td>7</td>
</tr>
</tbody>
</table>

For the `tachyon_analyze_locks` application, the Wait time is high. To identify the cause, you need to understand how this Wait time was distributed per synchronization objects.

The **Top Waiting Objects** section provides the list of five synchronization objects with the highest Wait Time and Wait Count, sorted by the Wait Time metric. For the `tachyon_analyze_locks` application, focus on the first three objects and explore the Bottom-up window data for more details.

The **Thread Concurrency Histogram** represents the Elapsed time and concurrency level for the specified number of running threads. Ideally, the highest bar of your chart should be within the Ok or Ideal utilization range.

Note the **Target** value. By default, this number is equal to the number of physical cores. Consider this number as your optimization goal.
The **Average** metric is calculated as CPU time / Elapsed time. Use this number as a baseline for your performance measurements. The closer this number to the number of cores, the better.

For the sample code, the chart shows that `tachyon_analyze_locks` is a multithreaded application running four threads on a machine with two cores. But it is not using available cores effectively. The Average CPU Usage on the chart is about 0.75 while your target should be making it as close to 2 as possible (for the system with two cores).

Hover over the second bar to understand how long the application ran serially. The tooltip shows that the application ran one thread for almost 29 seconds, which is classified by the VTune Amplifier XE as Poor concurrency.

The **CPU Usage Histogram** represents the Elapsed time and usage level for the logical CPUs. Ideally, the highest bar of your chart should be within the Ok or Ideal utilization range.

The `tachyon_analyze_locks` application ran mostly on one logical CPU. If you hover over the highest bar, you see that it spent 24.964 seconds using one core only, which is classified by the VTune Amplifier XE as a Poor utilization. To understand what prevented the application from using all available logical CPUs effectively, explore the Bottom-up pane.

### Identify Locks

Click the **Bottom-up** tab to explore the Bottom-up pane.

<table>
<thead>
<tr>
<th>Selected 1 row(s):</th>
<th>Wait Time</th>
<th>Wait Count</th>
<th>Spin Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex 0x3de2bb60</td>
<td>27.659s</td>
<td>508</td>
<td>0ms</td>
</tr>
<tr>
<td>draw_task:operate</td>
<td>27.659s</td>
<td>508</td>
<td>0ms</td>
</tr>
<tr>
<td>TBB Scheduler</td>
<td>10.001s</td>
<td>2</td>
<td>0ms</td>
</tr>
<tr>
<td>Unknown 0x04a1e6</td>
<td>9.045s</td>
<td>834</td>
<td>0ms</td>
</tr>
<tr>
<td>select</td>
<td>0.794s</td>
<td>528</td>
<td>9.962ms</td>
</tr>
<tr>
<td>Stream 0x2d41a05c</td>
<td>0.036s</td>
<td>468</td>
<td>0ms</td>
</tr>
<tr>
<td>Stream fusi/X11R6/</td>
<td>0.000s</td>
<td>1</td>
<td>0ms</td>
</tr>
</tbody>
</table>
1. Synchronization objects that control threads in the application. The hash (unique number) appended to some names of the objects identify the stack creating this synchronization object.

VTune Amplifier XE recognizes all types of Intel(R) Threading Building Blocks (Intel(R) TBB) objects. To display an overhead introduced by Intel TBB library internals, the VTune Amplifier XE creates a pseudo synchronization object TBB scheduler that includes all waits from the Intel TBB runtime libraries.

2. The utilization of the processor time when a given thread waited for some event to occur. By default, the synchronization objects are sorted by Poor processor utilization type. Bars showing OK or Ideal utilization (orange and green) are utilizing the processors well. You should focus your optimization efforts on functions with the longest poor CPU utilization (red bars if the bar format is selected). Next, search for the longest over-utilized time (blue bars).

This is the Data of Interest column for the Locks and Waits analysis results.

3. Number of times the corresponding system wait API was called. For a lock, it is the number of times the lock was contended and caused a wait. Typically, you should focus your tuning efforts on the waits with both high Wait Time and Wait Count values, especially if they have poor utilization.

For the analyzed sample code, you see that the top three synchronization objects caused the longest Wait time. The red bar in the Wait Time column indicate that during most of the CPU time for the first object, the processor cores were underutilized.

Focus on the first item in the Bottom-up pane. It is a Mutex that shows much serial time and is causing a wait. Click the arrow sign at the object name to expand the node and see the draw_task wait function that contains this mutex and call stack. Double-click the Mutex to see the source code for the wait function.

Recap

You identified a synchronization object with the high Wait Time and Wait Count values and poor CPU utilization that could be a lock affecting application parallelism. Your next step is to analyze the code of this function.

Key Terms and Concepts

- Term: Elapsed time, CPU time
- Concept: Locks and Waits Analysis, CPU Usage, Thread Concurrency, Data of Interest

Analyze Code

You identified the mutex that caused significant Wait time and poor processor utilization. Double-click this critical section in the Bottom-up pane to view the source. The Intel(R) VTune(TM) Amplifier XE opens source and disassembly code. Focus on the Source pane and analyze the source code:

- Understand basic options provided in the Source window.
- Identify the hottest code lines.
Understand Basic Source View Options

<table>
<thead>
<tr>
<th>Source</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>Source</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>for (int y=r.begin(); y!=r.end(); ++y) {</td>
</tr>
<tr>
<td>162</td>
<td>drawing_area drawing(startx, totaly-y, stopx-1);</td>
</tr>
<tr>
<td>163</td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>// Acquire mutex to protect pixel calculation for</td>
</tr>
<tr>
<td>165</td>
<td>pthread_mutex_lock (&amp;rgb_mutex);</td>
</tr>
<tr>
<td>166</td>
<td>for (int x = startx; x &lt; stopx; x++) {</td>
</tr>
<tr>
<td>167</td>
<td>color_t c = render_one_pixel (x, y, local_m);</td>
</tr>
<tr>
<td>168</td>
<td>drawing.put_pixel(c);</td>
</tr>
<tr>
<td>169</td>
<td>}</td>
</tr>
</tbody>
</table>

The table below explains some of the features available in the Source pane for the Locks and Waits viewpoint.

1. Source code of the application displayed if the function symbol information is available. When you go to the source by double-clicking the synchronization object in the Bottom-up window, the VTune Amplifier XE opens the wait function containing this object and highlights the code line that took the most Wait time. The source code in the Source pane is not editable.

   If the function symbol information is not available, the Assembly pane opens displaying assembler instructions for the selected wait function. To view the source code in the Source pane, make sure to build the target properly.

2. Processor time and utilization bar attributed to a particular code line. The colored bar represents the distribution of the Wait time according to the utilization levels (Idle, Poor, Ok, Ideal, and Over) defined by the VTune Amplifier XE. The longer the bar, the higher the value. Ok utilization level is not available for systems with a small number of cores.

   This is the Data of Interest column for the Locks and Waits analysis.

3. Number of times the corresponding system wait API was called while this code line was executing. For a lock, it is the number of times the lock was contended and caused a wait.

4. Source window toolbar. Use hotspot navigation buttons to switch between most performance-critical code lines. Hotspot navigation is based on the metric column selected as a Data of Interest. For the Hotspots analysis, this is CPU Time. Use the source file editor button to open and edit your code in your default editor.

Identify the Hottest Code Lines

The VTune Amplifier XE highlights line 165 entering the rgb_mutex mutex in the draw_task function. The draw_task function was waiting for almost 27 seconds while this code line was executing and most of the time the processor was underutilized. During this time, the critical section was contended 491 times.
The `rgb_mutex` is the place where the application is serializing. Each thread has to wait for the mutex to be available before it can proceed. Only one thread can be in the mutex at a time. You need to optimize the code to make it more concurrent. Click the Source Editor button on the Source window toolbar to open the code editor and optimize the code.

**Recap**
You identified the code section that caused a significant wait and during which the processor was poorly utilized.

**Key Terms and Concepts**
- Term: Wait time
- Concept: CPU Utilization, Locks and Waits Analysis, Data of Interest

**Remove Lock**

In the Source window, you located the mutex that caused a significant wait while the processor cores were underutilized and generated multiple wait count. Focus on this line and do the following:
- Open the code editor.
- Modify the code to remove the lock.

**Open the Code Editor**
Click the Source Editor button to open the `analyze_locks.cpp` file in your default editor at the hotspot code line:
The `rgb_mutex` was introduced to protect calculation from multithreaded access. The brief analysis shows that the code is thread safe and the mutex is not really needed.

To resolve this issue:

1. Comment out code lines 165 and 172 to disable the mutex.
2. Save the changes made in the source file.
3. Browse to the directory where you extracted the sample code (for example, `/home/intel/samples/en/tachyon_vtune_amp_xe`).
4. Rebuild your target in the release mode using the `make` command as follows:
   
   ```bash
   $ make clean
   $ make release
   ```

   The `tachyon_analyze_locks` application is rebuilt and stored in the `tachyon_vtune_amp_xe` directory.
5. Run `tachyon_analyze_locks` as follows:

   ```bash
   $ /home/intel/samples/en/tachyon_vtune_amp_xe/tachyon_analyze_locks dat/balls.dat
   ```

---

```cpp
151 class draw_task {
152
153 public:
154     void operator () (const tbb::blocked_range<int> &r) const {
155         unsigned int serial = 1;
156         unsigned int mboxsize = sizeof(unsigned int)*(max_objectid() + 20);
157         unsigned int * local_mbox = (unsigned int *) alloc(mboxsize);
158         memset(local_mbox, 0, mboxsize);
159
160         for (int y=r.begin(); y!=r.end(); ++y) {
161             drawing_area drawing(startx, totally-y, stopx-startx, 1);
162             // Acquire mutex to protect pixel calculation from multithreaded access (Needed?)
163             pthread_mutex_lock (&rgb_mutex);
164             for (int x = startx; x < stopx; x++) {
165                 color_t c = render_one_pixel (x, y, local_mbox, serial, startx, stopx, starty,
166                     drawing.put_pixel(c);
167             }
168             // Release the mutex after pixel calculation complete
169             pthread_mutex_unlock (&rgb_mutex);
170             if(!video->next_frame()) tbb::task::self().cancel_group_execution();
171         }
172         draw_task () {}
```
System runs the `tachyon_analyze_locks` application. Note that execution time reduced from 29.647 seconds to 14.615 seconds.

**Recap**

You optimized the application execution time by removing the unnecessary critical section that caused a lot of Wait time.

**Key Terms and Concepts**

- Term: hotspot
- Concept: Locks and Waits Analysis
Compare with Previous Result

You made sure that removing the mutex gave you 15 seconds of optimization in the application execution time. To understand the impact of your changes and how the CPU utilization has changed, re-run the Locks and Waits analysis on the optimized code and compare results:

- Compare results before and after optimization.
- Identify the performance gain.

Compare Results Before and After Optimization

1. Run the Locks and Waits analysis on the modified code.
2. Click the button on the Intel(R) VTune(TM) Amplifier XE toolbar.
   
   The Compare Results window opens.
3. Specify the Locks and Waits analysis results you want to compare:

   The Locks and Waits Bottom-up pane opens, showing the list of synchronization objects used in the code, Wait time utilization across the two results, and the differences side by side:

   Difference in Wait time per utilization level between the two results in the following format: <Difference Wait Time> = <Result 1 Wait Time> – <Result 2 Wait Time>. By default, the Difference column is expanded to display comparison data per utilization level. You may collapse the column to see the total difference data per Wait time.
Wait time and CPU utilization for the initial version of the code.

Wait time and CPU utilization for the optimized version of the code.

Difference in Wait count between the two results in the following format: \(<\text{Difference Wait Count}>=<\text{Results 1 Wait Count}>-<\text{Result 2 Wait Count}>.\)

Wait count for the initial version of the code.

Wait count for the optimized version of the code.

**Identify the Performance Gain**

In the Bottom-up pane, locate the Mutex you identified as a bottleneck in your code. Since you removed it during optimization, the optimized result r004lw does not show any performance data for this synchronization object. If you collapse the **Wait Time:Difference** column by clicking the button, you see that with the optimized result you got almost 27 seconds of optimization in Wait time.

<table>
<thead>
<tr>
<th>Mutex 0x3de2bb50</th>
<th>Wait Time:Difference</th>
<th>Wait Time: r003lw</th>
<th>Wait Time: r004lw</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>0.350s</td>
<td>0.794s</td>
<td>0.415s</td>
</tr>
<tr>
<td>Stream 0x9b855f4b</td>
<td>0.000s</td>
<td>0.000s</td>
<td>0.000s</td>
</tr>
<tr>
<td>Stream 0xf60515f9</td>
<td>0.000s</td>
<td>0.000s</td>
<td>0.000s</td>
</tr>
<tr>
<td>Socket 0x4c905243</td>
<td>0.000s</td>
<td>0.000s</td>
<td>0.000s</td>
</tr>
</tbody>
</table>

**Recap**

You ran the Locks and Waits analysis on the optimized code and compared the results before and after optimization using the Compare mode of the VTune Amplifier XE. The comparison shows that, with the optimized version of the tachyon_analyze_locks application (r004lw result), you managed to remove the lock preventing application parallelism and significantly reduce the application execution time. Compare analysis results regularly to look for regressions and to track how incremental changes to the code affect its performance. You may also want to use the VTune Amplifier XE command-line interface and run the amplxe-cl command to test your code for regressions. For more details, see the Command-line Interface Support section in the VTune Amplifier XE online help.

**Key Terms and Concepts**

- Term: hotspot, Wait time
- Concept: Locks and Waits Analysis, CPU Utilization
Summary

You have completed the Analyzing Locks and Waits tutorial. Here are some important things to remember when using the Intel(R) VTune(TM) Amplifier XE to analyze your code for locks and waits:

Step 1. Choose and Build Your Target

- Create a performance baseline to compare the application versions before and after optimization. Make sure to use the same workload for each application run.
- Create a VTune Amplifier XE project and use the Project Properties: Target tab to choose and configure your analysis target.

Step 2. Run Analysis

- Use the Analysis Type configuration window to choose, configure, and run the analysis. For example, you may limit the data collection to a predefined amount of data or enable the VTune Amplifier XE to collect more accurate CPU time data. You can also run the analysis from command line using the amplxe-cl command.

Step 3. Interpret Results and Resolve the Issue

- Start analyzing the performance of your application with the Summary pane to explore the performance metrics for the whole application. Then, move to the Bottom-up window to analyze the synchronization objects. Focus on the synchronization objects that under- or over-utilized the available logical CPUs and have the highest Wait time and Wait Count values. By default, the objects with the highest Wait time values show up at the top of the window.
- Expand the most time-critical synchronization object in the Bottom-up window and double-click the wait function it belongs to. This opens the source code for this wait function at the code line with the highest Wait time value.

Step 4. Compare Results Before and After Optimization

- Perform regular regression testing by comparing analysis results before and after optimization. From GUI, click the Compare Results button on the VTune Amplifier XE toolbar. From command line, use the amplxe-cl command.
- Expand each data column by clicking the button to identify the performance gain per CPU utilization level.
More Resources

Getting Help and Support

Intel(R) VTune(TM) Amplifier XE provides a number of Getting Started tutorials. These tutorials use a sample application to demo you the basic product features and workflows. You can access these documents through the Help menu or by clicking the VTune Amplifier XE icon.

For the standalone user interface, the tutorials are available via Help > Getting Started Tutorials menu.

To view help in the standalone user interface, select Help > Intel VTune Amplifier XE 2011 Help from the menu.

Navigating in the Product Usage Workflow

Where applicable, the VTune Amplifier XE help topics provide a Where am I in the workflow? button. Click the button to view the workflow with a highlight on the stage that this topic discusses.

Using Context-Sensitive Help

Context-sensitive help enables easy access to help topics on active GUI elements. The following context-sensitive help features are available on a product-specific basis:

- **F1 Help**: Press F1 to get help for an active dialog box, property page, pane, or window.

Product Website and Support

Product Website and Support

The following links provide information and support on Intel software products, including Intel(R) Parallel Studio XE:

  At this site, you will find comprehensive product information, including:
  - Links to each product, where you will find technical information such as white papers and articles
  - Links to user forums
  - Links to news and events
Intel(R)Software Network, Parallel Studio XE Support page, with links to support forums, startup help, knowledge base, and getting started video.

Intel(R) Software Development Products Knowledge Base.

http://www.intel.com/software/products/support/
Technical support information, to register your product, or to contact Intel.

For additional support information, see the Technical Support section of your Release Notes.

**System Requirements**
For detailed information on system requirements, see the Release Notes.
Troubleshooting

Problem: The Start button is disabled

The Start button on the command toolbar is disabled.

Solution: Make sure you specified an analysis target. If the target is not specified, click the Project Properties button on the command toolbar and enter the target name in the Application to Launch pane.

For the General Exploration analysis, the Start button may be disabled if you mistakenly chose the incorrect processor type. The selected analysis type should match your processor type.