Getting Started Tutorial: Analyzing Locks and Waits

Intel® VTune™ Amplifier XE 2011 for Linux* OS

C++ Sample Application Code

Document Number: 326707-001

Legal Information
Contents

Legal Information........................................................................................................5
Overview.................................................................................................................7

Chapter 1: Navigation Quick Start

Chapter 2: Analyzing Locks and Waits
  Build Application and Create New Project.........................................................11
  Run Locks and Waits Analysis............................................................................14
  Interpret Result Data..........................................................................................15
  Analyze Code......................................................................................................19
  Remove Lock.......................................................................................................20
  Compare with Previous Result..........................................................................22

Chapter 3: Summary

Chapter 4: Key Terms
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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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/design/literature.htm

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


*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.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Copyright (C) 2010-2012, Intel Corporation. All rights reserved.
Overview

Discover how to use the Hotspots analysis of the Intel(R) VTune(TM) Amplifier XE to understand where your application is spending time, identify hotspots - the most time-consuming program units, and detect how they were called.

The Hotspots analysis is useful to analyze the performance of both serial and parallel applications.

<table>
<thead>
<tr>
<th>About This Tutorial</th>
<th>This tutorial uses the sample tachyon_find_hotspots application and guides you through basic steps required to analyze the code for hotspots.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Duration</td>
<td>10-15 minutes.</td>
</tr>
<tr>
<td>Learning Objectives</td>
<td>After you complete this tutorial, you should be able to:</td>
</tr>
<tr>
<td></td>
<td>• Choose an analysis target.</td>
</tr>
<tr>
<td></td>
<td>• Choose the Hotspots analysis type.</td>
</tr>
<tr>
<td></td>
<td>• Run the Hotspots analysis to locate most time-consuming functions in an application.</td>
</tr>
<tr>
<td></td>
<td>• Analyze the function call flow and threads.</td>
</tr>
<tr>
<td></td>
<td>• Analyze the source code to locate the most time-critical code lines.</td>
</tr>
<tr>
<td></td>
<td>• Compare results before and after optimization.</td>
</tr>
</tbody>
</table>

More Resources

Navigation Quick Start

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. VTune Amplifier XE helps you analyze the algorithm choices and identify where and how your application can benefit from available hardware resources.

VTune Amplifier XE Access

1. In a terminal session, type `uname -m`. If the command returns:
   - `x86`, change directory to the `bin32` directory in the VTune Amplifier XE installation directory.
   - `x86_64`, change directory to the `bin64` directory in the VTune Amplifier XE installation directory.

   **NOTE** The default installation directory is `/opt/intel/vtune_amplifier_xe_2011`.

2. Type `source amplxe-vars.sh`.
   This script sets PATH and MANPATH variables.

3. Type `./amplxe-gui`.

VTune Amplifier XE GUI
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 Project Navigator to manage your VTune Amplifier XE projects and collected analysis results. Click the **Project Navigator** button on the toolbar to enable/disable the Project Navigator.

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.

Use the **Grouping** drop-down menu to choose a granularity level for grouping data in the grid.

Use the filter toolbar to filter out the result data according to the selected categories.
Analyzing 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.

Step 1. Prepare for analysis
Do one of the following:
- Visual Studio* IDE: Choose a project, verify settings, and build application.
- Standalone GUI: Build an application to analyze for locks and waits and create a new VTune Amplifier XE project.

Step 2. Find lock
- Choose and run the Locks and Waits analysis.
- Interpret the result data.
- View and analyze code of the performance-critical function.

Step 3. Remove lock
Modify the code to remove the lock.

Step 4. Check your work
Re-build the target, re-run the Locks and Waits analysis, and compare the result data before and after optimization.

Build Application and Create New Project

Before you start analyzing your application for locks and waits, do the following:
1. Get software tools.
2. Build application in the release mode.
3. Create a performance baseline.
4. Create a VTune Amplifier XE project.

Get Software Tools
You need the following tools to try tutorial steps yourself using the tachyon sample application:

- VTune Amplifier XE, including sample applications
- .tgz file extraction utility
- Supported compiler (see Release Notes for more information)

Acquire Intel 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/.

Install and Set Up VTune Amplifier XE Sample Applications

1. Copy the tachyon_vtune_amp_xe.tgz file from the <install-dir>/samples/<locale>/C++/ directory to a writable directory or share on your system. The default installation path is opt/intel/vtune_amplifier_xe_2011.
2. Extract the sample from the .tgz file.

- Samples are non-deterministic. Your screens may vary from the screen captures shown throughout this tutorial.
- Samples are designed only to illustrate the VTune Amplifier XE features; they do not represent best practices for creating code.

Build the Target in the Release Mode
Build the target in the Release mode with full optimizations, which is recommended for performance analysis.

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:
   $ 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/samples/en/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.

---

**Create a Project**

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:

   ```bash
   $ 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. By default, the `<install_dir>` is `/opt/intel/vtune_amplifier_xe_2013`.

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.
   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.

Key Terms
• Baseline
• Target

Next Step
Run Locks and Waits Analysis

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, analysis type from the drop-down menu 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.

- 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.
- 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**
- Finalization
- Viewpoint

**Next Step**
Interpret Result Data

**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:

1. Analyze the basic performance metrics provided by the Locks and Waits analysis.
2. Identify locks.

**NOTE** The screenshots and execution time data provided in this tutorial are created on a system with four CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

**Analyze the Basic Locks and Waits Metrics**

Start with exploring the data provided in the Summary window for the whole application performance. To interpret the data, hover over the question mark icons to read the pop-up help and better understand what each performance metric means.
The **Result Summary** section provides data on the overall application performance per the following metrics:

**Elapsed Time: 39.089s**

- **Wait Time:** 48.435s
- **Wait Count:** 2,347
- **CPU Time:** 28.680s
- **Total Thread Count:** 2
- **Spin Time:** 0.010s

1) **Elapsed Time** is the total time for each core when it was either waiting or not utilized by the application; 2) **Total Thread Count** is the number of threads in the application; 3) **Wait Time** is the amount of time the application threads waited for some event to occur, such as synchronization waits and I/O waits; 4) **Wait Count** is the overall number of times the system wait API was called for the analyzed application; 5) **CPU Time** is the sum of CPU time for all threads; 6) **Spin Time** is the time a thread is active in a synchronization construct.

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.

**Top Waiting Objects**

This section lists the objects that spent the most time, as sleep() or I/O, or on contended synchronization object, reflects high contention for that object and, therefore, a possible performance bottleneck.

<table>
<thead>
<tr>
<th>Sync Object</th>
<th>Wait Time</th>
<th>Wait Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex 0x3de2bb60</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 0x04a1e655</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, focus on the first three objects and explore the Bottom-up pane 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 two threads on a machine with four cores. But it is not using available cores effectively. The Average CPU Usage on the chart is about 0.8 while your target should be making it as closer to 4 as possible (for the system with four 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 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 second bar, you see that it spent 24.897 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 open the Bottom-up pane.
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.

For Intel(R) Threading Building Blocks (Intel(R) TBB), VTune Amplifier XE is able to recognize all types of Intel 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.

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 that is used for different types of calculations, for example: call stack contribution, percentage value on the filter toolbar.

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. Usually you are recommended to focus your tuning efforts on the waits with both high Wait Time and Wait Count values, especially if they have poor utilization.

Wait time, during which the CPU is busy. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting. Some Spin time may be preferable to the alternative of the increased thread context switches. However, too much Spin time can reflect lost opportunity for productive work.

For the analyzed sample code, you see that the top three synchronization objects caused the longest Wait time. The red bars in the **Wait Time** column indicate that most of the time for these objects processor cores were underutilized.

Consider the first item in the Bottom-up pane that is more interesting. 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.
Key Terms
- CPU usage
- Elapsed time
- Wait time

Next Step
Analyze Code

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:

1. Understand basic options provided in the Source window.
2. 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></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td></td>
</tr>
<tr>
<td>162</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>164</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>pthread_mutex_lock (&amp;rgb_mutex);</td>
</tr>
<tr>
<td>166</td>
<td></td>
</tr>
<tr>
<td>167</td>
<td></td>
</tr>
<tr>
<td>168</td>
<td></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 pane, 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.
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.

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 Locks and Waits analysis, this is Wait 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 contented 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.

**Key Terms**

- CPU usage
- Wait time

**Next Step**

Remove Lock

**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:

1. Open the code editor.
2. 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:
   
   ```
   $ 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:
   
   ```
   $ /home/intel/samples/en/tachyon_vtune_amp_xe/tachyon_analyze_locks dat/balls.dat
   ```

---

```cpp
151 class draw_task {
152 public:
153     void operator () (const tbb::blocked_range<int> &r) const {
154         unsigned int serial = 1;
155         unsigned int nboxsize = sizeof(unsigned int)*(max_objectid() + 20);
156         unsigned int * local_nbox = (unsigned int *) alloca(nboxsize);
157         memset(local_nbox, 0, nboxsize);
158
159         for (int y=r.begin(); y!=r.end(); ++y) {
160             drawing_area drawing(startx, totaly-y, stopx-startx, 1);
161
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_nbox, 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     }
173
174     draw_task () {};
```
System runs the `tachyon_analyze_locks` application. Note that execution time reduced from 29.647 seconds to 14.615 seconds.

### Key Terms
- Hotspot

### Next Step
- Compare with Previous Result

### 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:
1. Compare results before and after optimization.
2. Identify the performance gain.

**Compare Results Before and After Optimization**

1. Run the Locks and Waits analysis on the modified code.
2. Click the **Compare Results** 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:

   ![Choose Results to Compare](image)

   The Summary window opens providing the statistics for the difference between collected results.

   Click the **Bottom-up** tab to see the list of synchronization objects used in the code, Wait time utilization across the two results, and the differences side by side:

   ![Locks and Waits - Locks and Waits](image)

   **Difference in Wait time per utilization level between the two results in the following format:**
   
   
   
   \[
   \text{Difference in Wait Time} = \text{Result 1 Wait Time} - \text{Result 2 Wait Time}.
   \]

   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:**
   
   \[
   \text{Difference in Wait Count} = \text{Result 1 Wait Count} - \text{Result 2 Wait Count}.
   \]

   **Wait count for the initial version of the code.**
Identify the Performance Gain

The Elapsed time data in the Summary window shows the optimization of 4 seconds for the whole application execution and Wait time decreased by 37.5 seconds.

**Elapsed Time: \(21.395s - 17.373s = 4.022s\)**

- **Total Thread Count:** Not changed, 6
- **Wait Time:** \(105.404s - 67.874s = 37.530s\)
- **Spin Time:** Not changed, 0s
- **Wait Count:** \(3,253 - 1,458 = -9,223,372,036,854,775,808\)
- **CPU Time:** \(10.419s - 16.419s = -6.000s\)
- **Paused Time:** Not changed, 0s

According to the **Thread Concurrency** histogram, before optimization (blue bar) the application ran serially for 9 seconds poorly utilizing available processor cores but after optimization (orange bar) it ran serially only for 2 seconds. After optimization the application ran 5 threads simultaneously overutilizing the cores for almost 5 seconds. Further, you may consider this direction as an additional area for improvement.

**Thread Concurrency Histogram**

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

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.
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**

- CPU usage
- Elapsed time
- Hotspot
- Wait time
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Tutorial Recap</th>
<th>Key Tutorial Take-aways</th>
</tr>
</thead>
</table>
| 1. Prepare for analysis   | You built the target in the Release mode, created the performance baseline, and created the VTune Amplifier XE project for your analysis target. Your application is ready for analysis. | • 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. |
| 2. Find lock              | You ran the Locks and Waits data collection and identified the following hotspots:  
• 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.  
• Identified the code section that caused a significant wait and during which the processor was poorly utilized. | • 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.  
• 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. |
| 3. Remove lock            | You optimized the application execution time by removing the unnecessary critical section that caused a lot of Wait time. | Expand the most time-critical synchronization object in the Bottom-up pane 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.  
• 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. |
| 4. Check your work         | 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 |  
Step | Tutorial Recap | Key Tutorial Take-aways
--- | --- | ---
tachyon_analyze_locks application (r004lw result), you managed to remove the lock preventing application parallelism and significantly reduce the application execution time. | • Expand each data column by clicking the button to identify the performance gain per CPU utilization level.

**Next step:** Prepare your own application(s) for analysis. Then use the VTune Amplifier XE to find and eliminate locks preventing parallelism.
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.

**CPU usage**: A performance metric when the VTune Amplifier XE identifies a processor utilization scale, calculates the target CPU usage, and defines default utilization ranges depending on the number of processor 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 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>
<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>

**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.

**finalization**: A process during which the Intel(R) VTune(TM) Amplifier XE converts the collected data to a database, resolves symbol information, and pre-computes data to make further analysis more efficient and responsive.

**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.

**thread concurrency**: A performance metric that helps identify how an application utilizes the processors in the system by comparing the application concurrency level (the number of active threads) and target concurrency level (by default, equal to the number of physical cores). Thread concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

<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 program are waiting - no threads are running. There can be only one node in the Summary 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>
</tbody>
</table>
Utilities

<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 utilization. By default, ideal utilization is when the number of threads is between 86-115% of the target concurrency.</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>

**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, click the button and 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.