Getting Started Tutorial: Finding Hotspots

Intel® VTune™ Amplifier XE 2013 for Linux* OS

Fortran Sample Application Code

Document Number: 327359-001

Legal Information

Contents

Legal Information

Overview

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

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

About This Tutorial	This tutorial uses the sample `tachyon` and guides you through basic steps required to analyze the code for hotspots.
Estimated Duration	10-15 minutes.
Learning Objectives	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.
More Resources	• Intel® Parallel Studio XE tutorials (HTML, PDF): http://software.intel.com/en-us/articles/intel-software-product-tutorials/ • Intel® Parallel Studio XE support page: http://software.intel.com/en-us/articles/intel-parallel-studio-xe/

Navigation Quick Start

1

Intel® VTune™ Amplifier XE, an Intel® 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 algorithm choices and identify where and how your application can benefit from available hardware resources.

VTune Amplifier XE Access

VTune Amplifier XE installation includes shell scripts that you can run in your terminal window to set up environment variables:

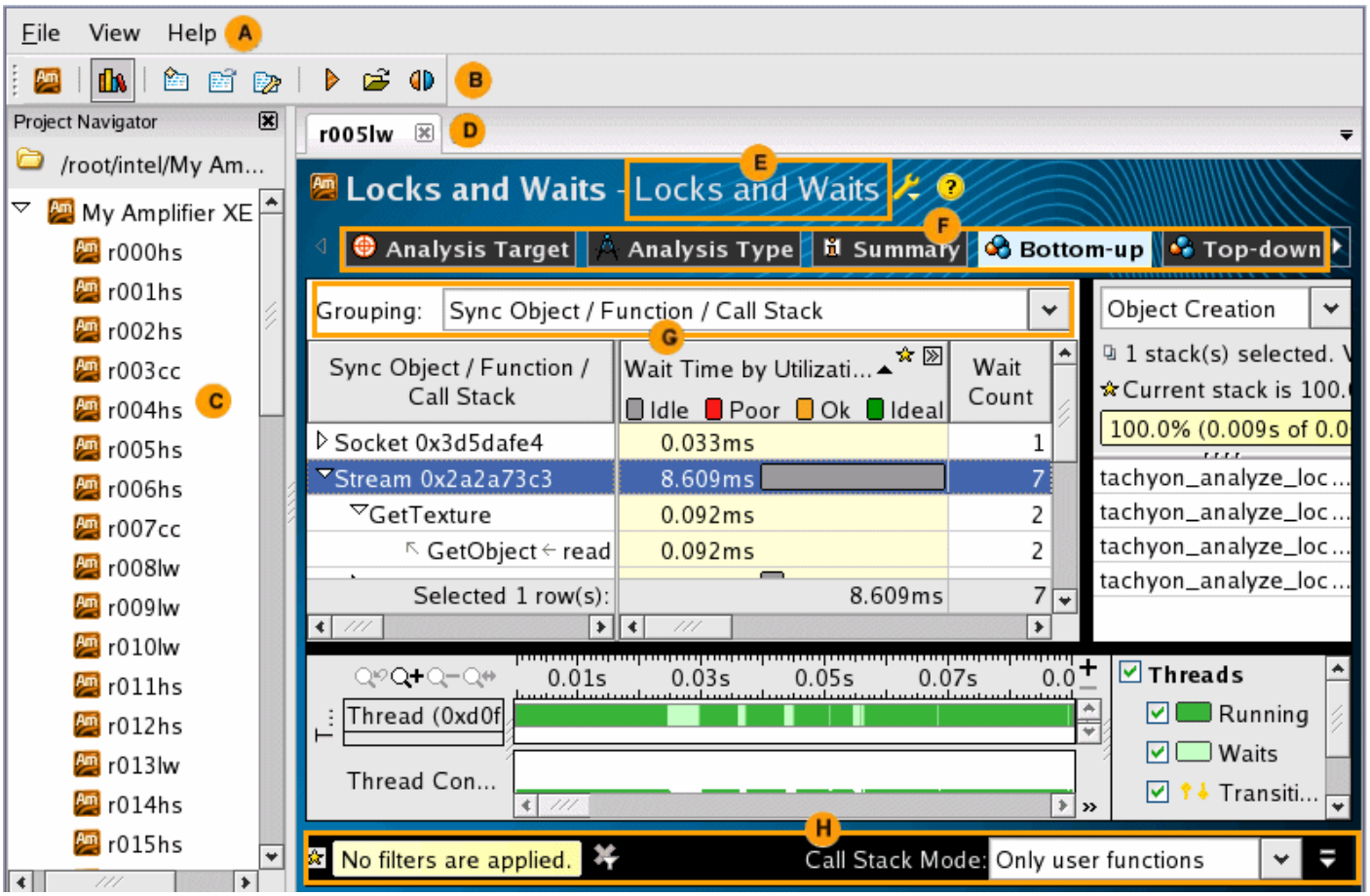
1. From the installation directory, type `source amplxe-vars.sh`.

   This script sets the PATH environment variable that specifies locations of the product graphical user interface utility and command line utility.

NOTE The default installation directory is `/opt/intel/vtune_amplifier_xe_2013`.

2. Type `amplxe-gui` to launch the product graphical interface.

VTune Amplifier XE GUI



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

B Configure and manage projects and results, and launch new analyses from the primary toolbar. Click the **Project Properties** button on this toolbar to manage result file locations.

C The **Project Navigator** provides an iconic representation of your projects and analysis results. Click the **Project Navigator** button on the toolbar to enable/disable the **Project Navigator**.

D Newly completed and opened analysis results along with result comparisons appear in the results tab for easy navigation.

E 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 viewpoints to focus on particular performance metrics. Click the yellow question mark icon to read the viewpoint description.

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

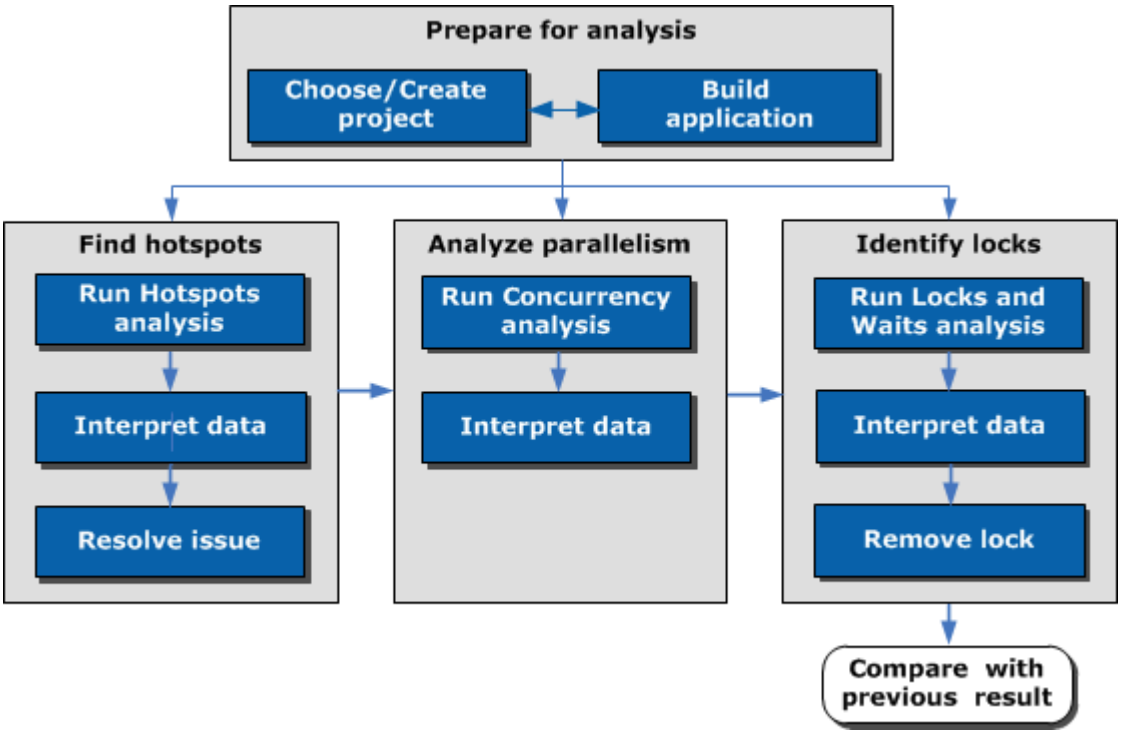
G Use the **Grouping** drop-down menu to choose a granularity level for grouping data in the grid.

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

Finding Hotspots

2

You can use the Intel® VTune™ 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 multithreaded application named `nqueens_parallel`.



Step 1: Prepare for analysis	Build an application to analyze for hotspots and create a new VTune Amplifier XE project
Step 2: Find hotspots	• Choose and run the Hotspots analysis. • Interpret the result data. • Resolve issue.
Step 3: Analyze parallelism	• Choose and run the Concurrency analysis. • Interpret the result data.
Step 4: Identify locks	• Choose and run the Locks and Waits analysis. • Interpret the result data. • Remove lock.
Step 5: 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 target for hotspots, 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 `nqueens_fortran` sample application:

- VTune Amplifier XE, including sample applications
- `.tgz` file extraction utility
- Supported Fortran 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 `nqueens_fortran.tgz` file from the `<install-dir>/samples/<locale>/Fortran` directory to a writable directory or share on your system. The default installation path is `opt/intel/vtune_amplifier_xe_2013` .
2. Extract the sample from the tgz file.

NOTE

- 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/fortran/linux`). 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
   ```

   The `nqueens_parallel` application is built.

Create a Performance Baseline

Run the application to create a performance baseline that will be used to identify optimization you achieve during performance tuning with the VTune Amplifier XE.

1. Run `nqueens_parallel` with the task size of 15. For example:

   ```
   $ ./nqueens_parallel 15
   ```

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

```
[vtune@nntpdsd52-082 linux]$ ./nqueens_parallel 15
Starting nqueens solver for size 15 with 4 thread(s)
Number of solutions: 2279184
Correct Result!
Calculations took 25335ms.
```

2. Note the execution time displayed in the shell window caption. In the example above, the execution time is 25335 milliseconds.

- 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.
- The screenshots and execution time data provided in this tutorial are created on a system with 4 CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

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:

   ```
   $ 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.

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

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

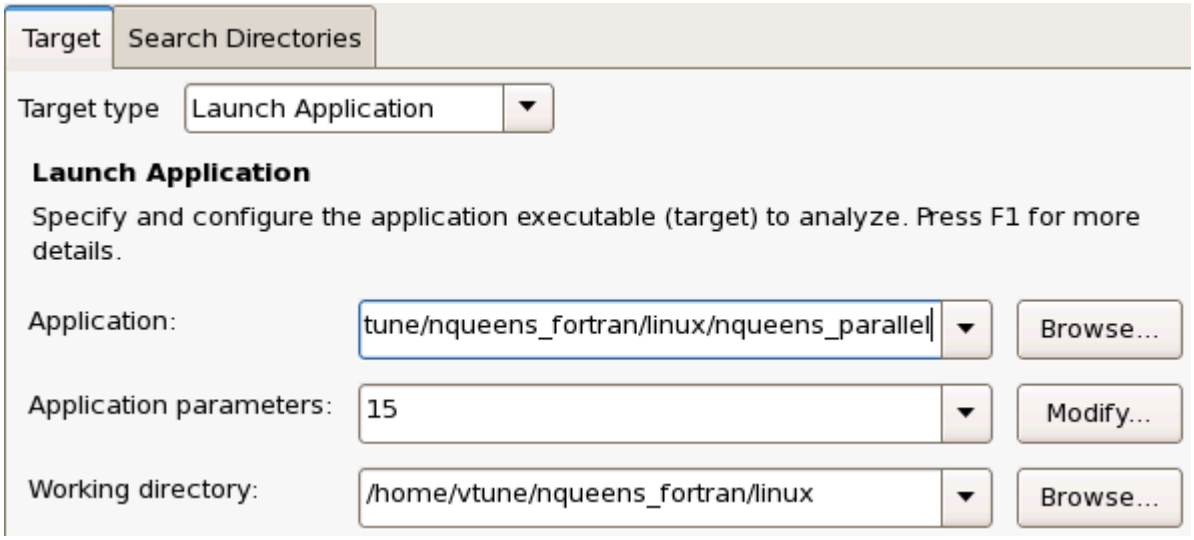
   The **Create a Project** dialog box opens.

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

   VTune Amplifier XE creates the `nqueens_parallel` project directory under the `root/intel/ampl/projects` directory and opens the **Project Properties: Target** dialog box.

5. In the **Target** tab, select the **Application to Launch** target type and specify your target as follows:
   - For the **Application** field, browse to: `<sample_code_dir>`, for example: `/home/vtune/nqueens_fortran/linux/nqueens_parallel`.

6. In the **Application parameters** field, specify the task size for this target: 15.



7. Click **OK** to apply the settings and exit the **Project Properties** dialog box.

Recap

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.

Key Terms

- Baseline
- Target

Next Step

Run Hotspots Analysis

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Run Hotspots Analysis

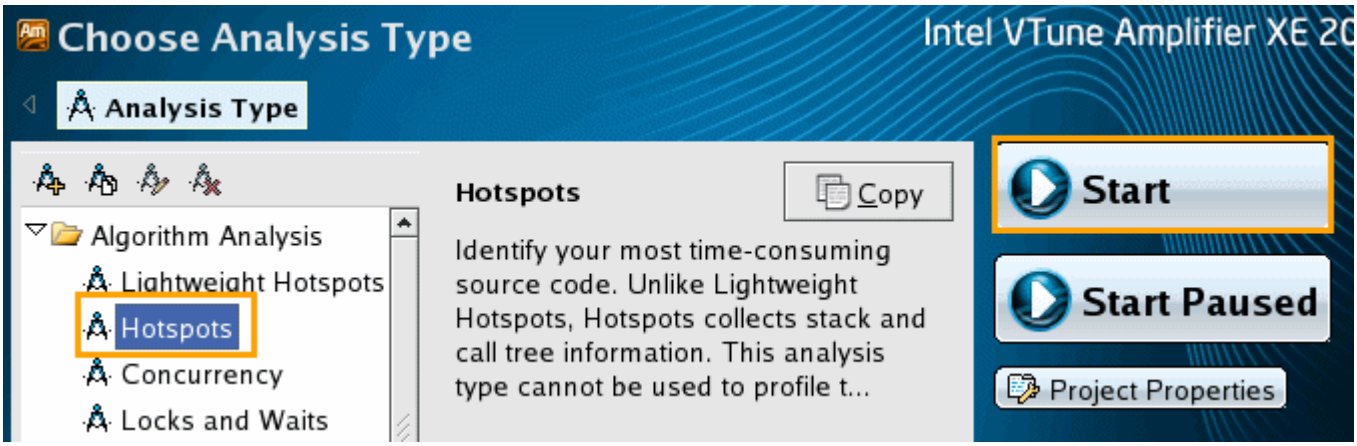
Before running an analysis, choose a configuration level to influence Intel(R) VTune(TM) Amplifier XE analysis scope and running time. In this tutorial, you run the Hotspots analysis to identify the hotspots that took much time to execute.

To run an analysis:

1. From the VTune Amplifier XE toolbar, click the **New Analysis** button.

   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 predefined settings for the Hotspots analysis.
3. Click the **Start** button on the right command bar.



VTune Amplifier XE launches the `nqueens_parallel` application that makes calculations, displays the execution time, and exits. VTune Amplifier XE finalizes the collected results and opens the analysis results in the **Hotspots** 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.

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

- Elapsed time
- Finalization
- Hotspot
- Viewpoint

Next Step

Interpret Hotspots Results

Interpret Hotspots Results

When the sample application exits, the Intel(R) VTune(TM) Amplifier XE finalizes the results and opens the **Hotspots** viewpoint where each window or pane is configured to display code regions that consumed a lot of CPU time. To interpret the data on the sample code performance, do the following:

1. Explore application-level performance.
2. Analyze the most time-consuming functions.
3. Identify the hotspot code region.

NOTE The screenshots and execution time data provided in this tutorial are created on a system with 4 CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

Explore Application-level Performance

Start analysis with the **Summary** window that opens by default when data collection completes. To interpret the data, hover over the question mark icons to read the pop-up help and better understand what each performance metric means.

Note that **CPU Time** for the sample application is equal to 23.900 seconds. It is the sum of CPU time for all application threads. **Total Thread Count** is 5, so the sample application is multi-threaded.

The **Top Hotspots** section provides data on the most time-consuming functions (*hotspot functions*) sorted by CPU time spent on their execution.

Function	CPU Time
setqueen	81.715s
__kmp_wait_sleep	0.200s
solve	0.160s
__kmp_execute_tasks	0.060s
__kmp_x86_pause	0.050s
[Others]	0.076s

For the sample application, the `setqueen` function, which took 81.715 seconds to execute, shows up at the top of the list as the hottest function.

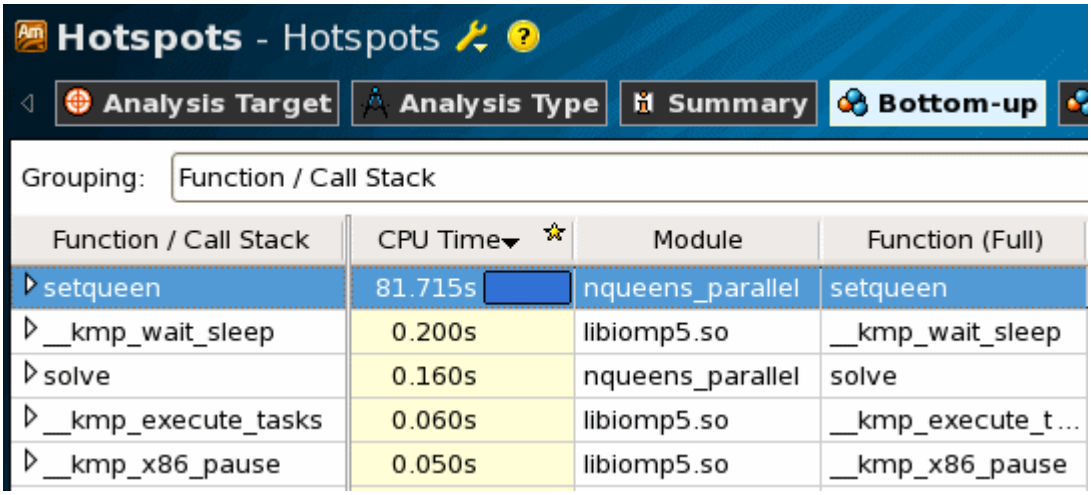
The `[Others]` entry at the bottom shows the sum of CPU time for all functions not listed in the table.

Analyze the Most Time-consuming Functions

Click the **Bottom-up** tab to explore the **Bottom-up** pane. By default, the data in the grid is sorted by **Function/Call Stack**.

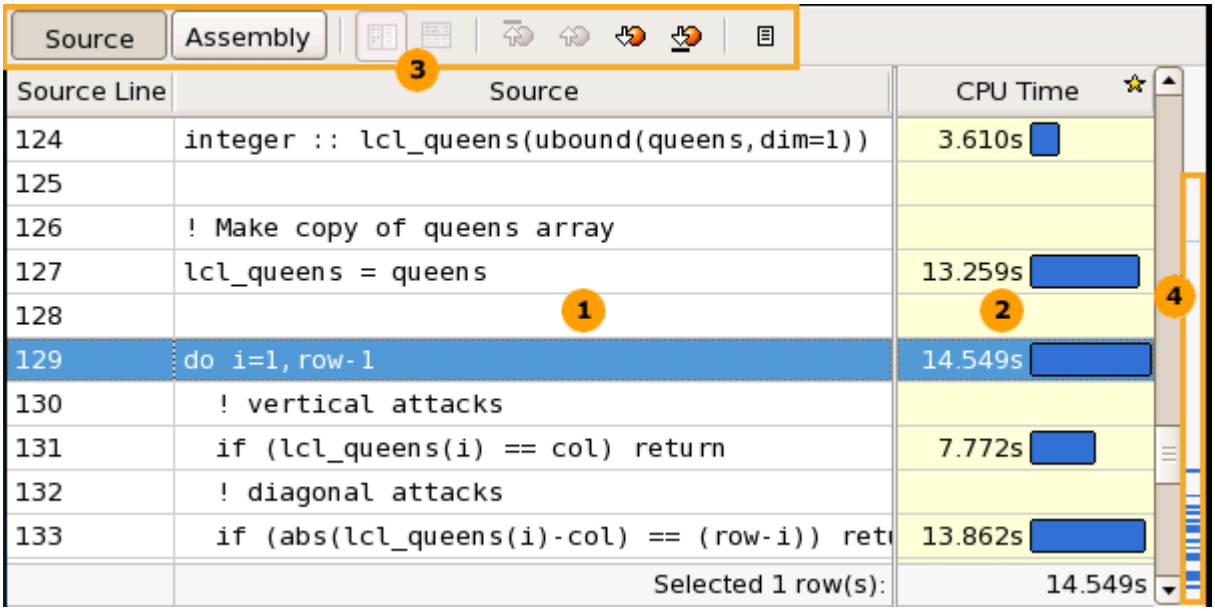
Analyze the **CPU Time** column values. This column is marked with a yellow star as the Data of Interest column. It means that the VTune Amplifier XE uses this type of data for some calculations (for example, filtering, stack contribution, and others). Functions that took most CPU time to execute are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

The `setqueen` function took 81.715 seconds to execute.



Double-click the hotspot function to open the source and identify the most time-critical code lines.

Identify the Hotspot Code Region



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 beginning of the hotspot function 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. 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.

3. **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.

4. 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.

By default, when you double-click the hotspot in the **Bottom-up** pane, the VTune Amplifier XE opens the source file related to this function. Click the hotspot navigation button to go to the code lines that took the most CPU time. For the `setqueen` function, focus on code line 129 that has one of the highest CPU time values for the selected function. This code is used to create a local copy of the `queens` array to avoid a data race. Click the **Source Editor** button on the **Source** window toolbar to open the default code editor and work on optimizing the code.

Key Terms

- CPU time
- Viewpoint

Next Step

Resolve Issue

Resolve Issue

You identified that the most time-consuming function is `setqueen`. If you click the  **Source Editor** button from the **Source** pane, the VTune Amplifier XE opens the source `nqueens_parallel.f90` file at the hotspot line in the default code editor. You see that the OpenMP* cycle is calling the recursive `setQueen` function that initializes the `queens` array. To avoid a data race, this array is copied in each thread (see line 127):

```
123   ! In order to avoid a data race on the "queens" array,
124   integer :: lcl_queens(ubound(queens,dim=1))
125
126   ! Make copy of queens array
127   lcl_queens = queens
128
129   do i=1,row-1
130     ! vertical attacks
131     if (lcl_queens(i) == col) return
132     ! diagonal attacks
133     if (abs(lcl_queens(i)-col) == (row-i)) return
134   end do
```

This means that the number of local copies is equal to the number of threads. Since the function is recursive, the array is also copied in every function call, which is unnecessary and creates a big overhead.

To resolve this issue, you may enable OpenMP to make a copy of the array per thread. To do this:

1. Comment out lines 124 and 127.

```
123   ! In order to avoid a data race on the "queens" array,
124   ! integer :: lcl_queens(ubound(queens,dim=1))
125
126   ! Make copy of queens array
127   ! lcl_queens = queens
```

2. Search and replace all `lcl_queens` entries with `queens`.
3. Edit line 159 to add the `PRIVATE(queens)` directive.

   This enables the OpenMP run-rime to create a private copy of the array for each thread.

```
158 ! Enable dynamic load scheduling
159 !$OMP PARALLEL DO PRIVATE(queens)
160   do i=1,size
161     ! try all positions in first row
162     call SetQueen (queens, 1, i)
163   end do
```

4. Save the changes made in the source file.
5. Browse to the directory where you extracted the sample code (for example, `/home/vtune/nqueens_fortran/linux`).
6. Rebuild your target in the release mode using the make command as follows:

   ```
   $ make clean
   ```

   ```
   $ make
   ```

   The `nqueens_parallel` application is rebuilt.
7. Run `nqueens_parallel` as follows:

   ```
   ./nqueens_parallel 15
   ```

```
[vtune@nntpdsd52-082 linux]$ ./nqueens_parallel 15
Starting nqueens solver for size 15 with 4 thread(s)
Number of solutions: 2279184
Correct Result!
Calculations took 14940ms.
```

System runs `nqueens_parallel`. Note that the execution time has reduced from 7417 ms to 14940 ms. This means that the proposed solution gives 10395 ms of CPU time reduction.

To identify other possible performance issues, you may run the Concurrency analysis and see how effectively your application is parallelized.

Next Step

Run Concurrency Analysis

Run Concurrency Analysis

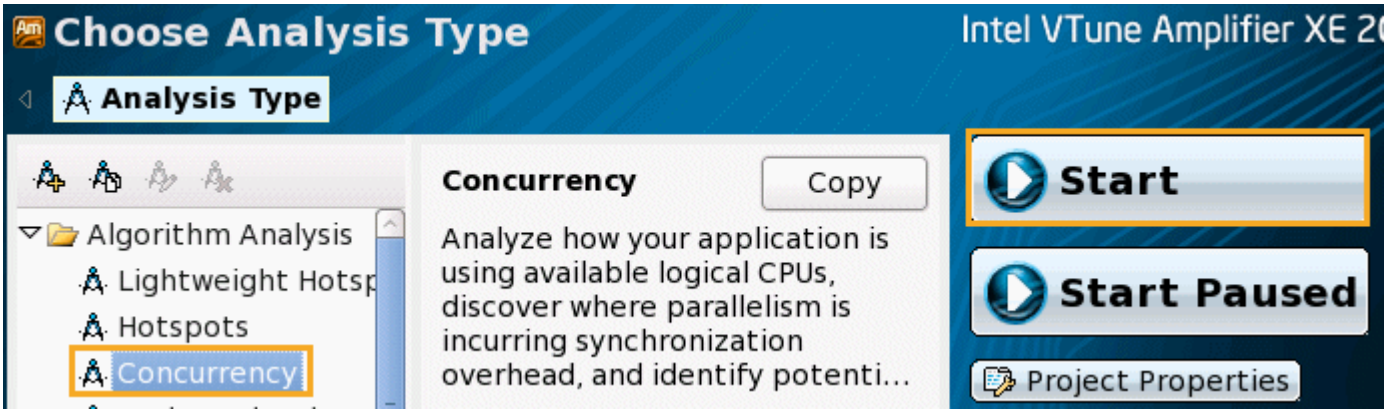
Run the Concurrency analysis to understand how effectively your application is parallelized.

To run an analysis:

1. From the VTune Amplifier XE toolbar, click the **New Analysis** button.

   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 > Concurrency**.

   The right pane is updated with the predefined settings for the Concurrency analysis.
3. Click the **Start** button on the right command bar.



VTune Amplifier XE launches the `nqueens_parallel` application that makes calculations, displays the execution time, and exits. VTune Amplifier XE finalizes the collected results and opens the analysis results in the **Hotspots by Thread Concurrency** 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.

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

- Finalization
- Viewpoint

Next Step

Interpret Concurrency Results

Interpret Concurrency Results

When the sample application exits, the Intel® VTune™ Amplifier E finalizes the results and opens the **Hotspots by Thread Concurrency** viewpoint where each window or pane is configured to display data on application parallelism and usage of processor cores. To interpret the data on the sample code performance, do the following:

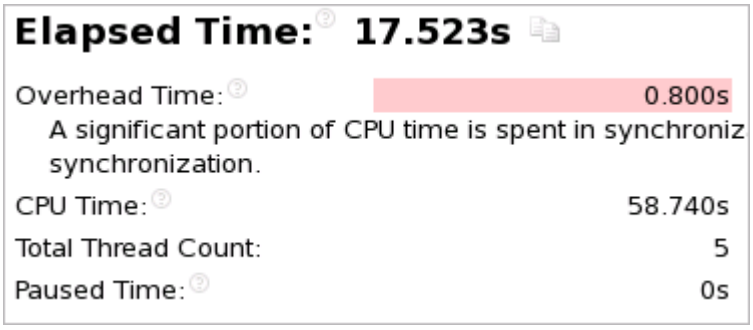
1. Explore application-level concurrency
2. Identify the most time-consuming function.

NOTE The screenshots and execution time data provided in this tutorial are created on a system with 4 CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

Explore Application-level Concurrency

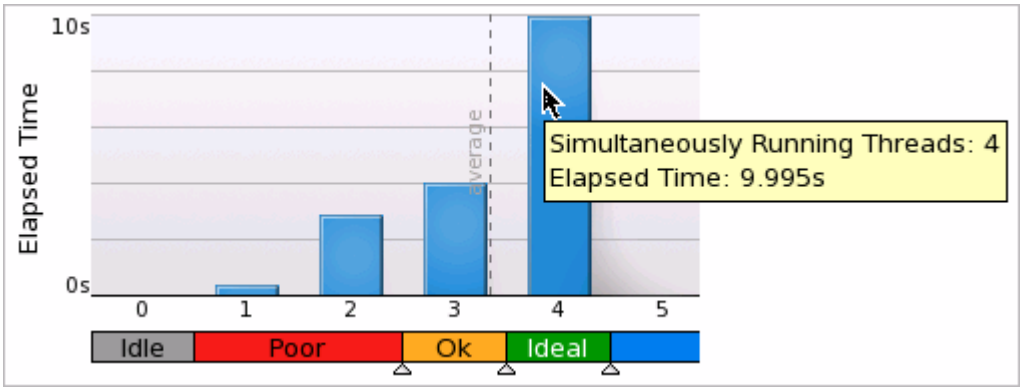
Start analysis with the **Summary** window that opens by default when data collection completes. To interpret the data, hover over the question mark icons to read the pop-up help and better understand what each performance metric means.

You see that after optimization the Elapsed time has reduced from 23.900 seconds to 17.523 seconds.



NOTE The Concurrency analysis adds an overhead to the application execution. The overhead often depends on the number of threads and synchronization objects used in the application. This is the reason why Elapsed time data provided in the **Summary** window may differ from the data reported after the application launch outside of the VTune Amplifier XE.

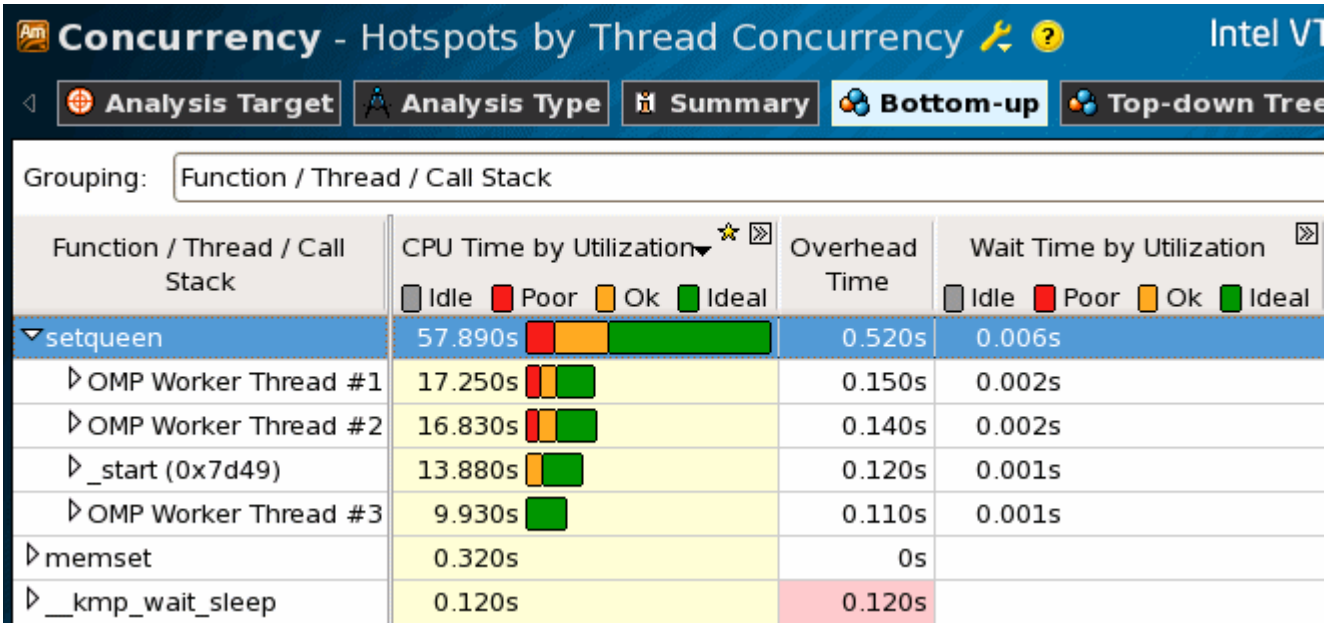
The **Thread Concurrency Histogram** shows that the average concurrency level of the sample application is about 3.3 while the target concurrency level for this application on the 4-core system is 4. If you hover over the highest bar, you see that this application has run 4 threads for almost 10 seconds, which is categorized by the VTune Amplifier XE as Ideal processor utilization. The application has run one and two threads simultaneously for more than 3.5 seconds, which is classified as poor parallelization.



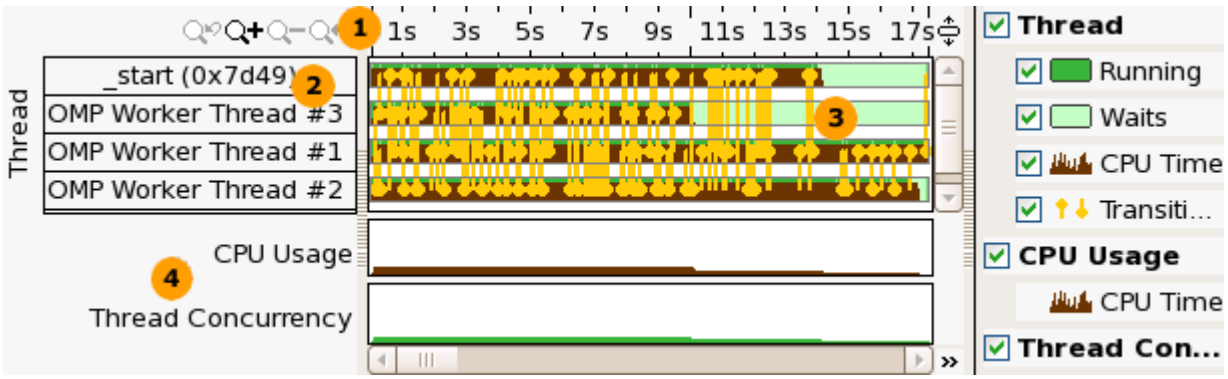
Identify the Most Time-consuming Function

Click the **Bottom-up** tab to switch to the **Bottom-up** window and analyze application performance by function. By default, the grid is sorted by the **CPU Time by Utilization** metric in the descending order. Select the **Function/Thread/Call Stack** grouping level from the **Grouping** menu. This granularity enables you to visualize threads where the hotspots functions were executed.

After initial optimization, the `setqueen` function is still a bottleneck. Click the arrow sign ▷ at the `setqueen` function. You see that this function's execution was parallelized among four threads.



Select these threads in the grid, right-click and choose the **Filter In by Selection** context menu option. The **Timeline** pane below is updated to display data for the selected threads only.



1. **Timeline area**. When you hover over the graph element, the timeline tooltip displays the time passed since the application has been launched.
2. **Threads area** that shows the distribution of CPU time utilization per thread. Hover over a bar to see the CPU time utilization in percent for this thread at each moment of time. Dark green zones show the time threads are active. Light-green zones show the time threads were waiting.
3. **Transitions**. The execution flow between threads where one thread signals to another thread waiting to receive that signal. You may zoom in to a time region to get more detailed view of the transitions. To do this, drag and drop to select the region and right-click to select the **Zoom In on Selection** option from the context menu.
4. **Performance metric area** that shows application performance over time by a performance metric. In the **Hotspots by Thread Concurrency** viewpoint, **CPU Usage** and **Thread Concurrency** metrics are used.

   The CPU Usage chart shows the distribution of CPU time utilization for the whole application. Hover over a bar to see the application-level CPU time utilization in percent at the particular moment. VTune Amplifier XE calculates the overall **CPU Usage** metric as the sum of CPU time per each thread of the Threads area. Maximum **CPU Usage** value is equal to [number of processor cores] x 100%.

   The **Thread Concurrency** chart shows the application-level thread concurrency at each moment of time. Hover over a bar to see an exact concurrency level at the particular moment.

The **Timeline** pane for the sample application shows a large number of transitions between threads, which means that the threads spent noticeable time transferring execution to each other. If you uncheck the **Transitions** display option on the right, you see that workload balance is also poor since three of four threads were waiting for `OMP Worker Thread #1` to complete execution.

Run the Locks and Waits analysis to understand what prevents the sample code from effective thread concurrency and processor utilization.

Key Terms

- Thread concurrency
- Viewpoint

Next Step

Run Locks and Waits Analysis

Run Locks and Waits Analysis

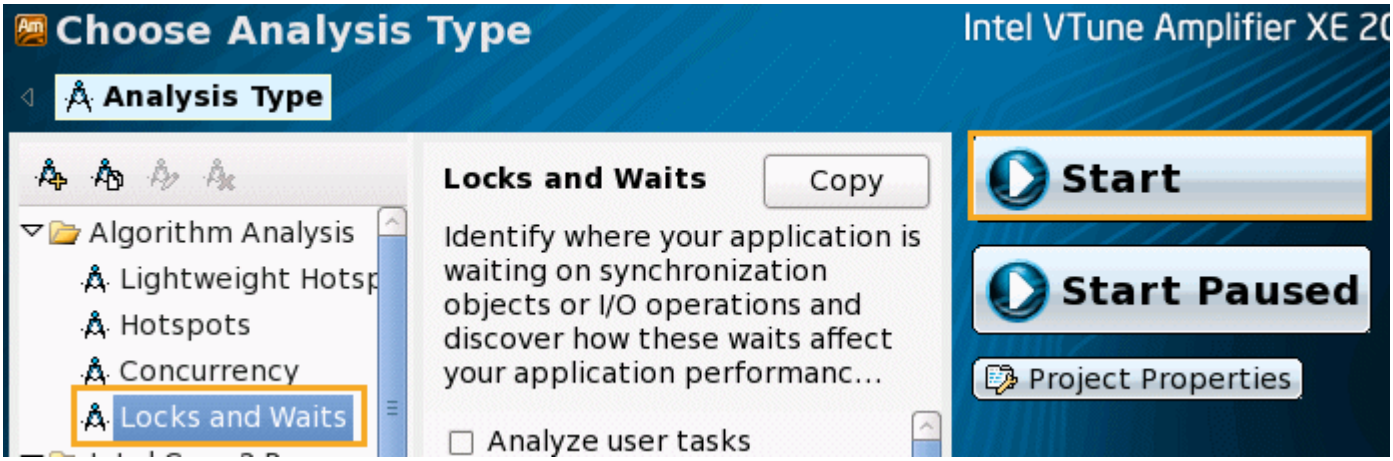

Run the Locks and Waits analysis to identify synchronization objects that caused contention and fix the problem in the source.

To run an analysis:

1. From the VTune Amplifier XE toolbar, click the  **New Analysis** button.

   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.



VTune Amplifier XE launches the `nqueens_parallel` executable that makes calculations, displays the execution time, and exits. 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 Locks and Waits Results

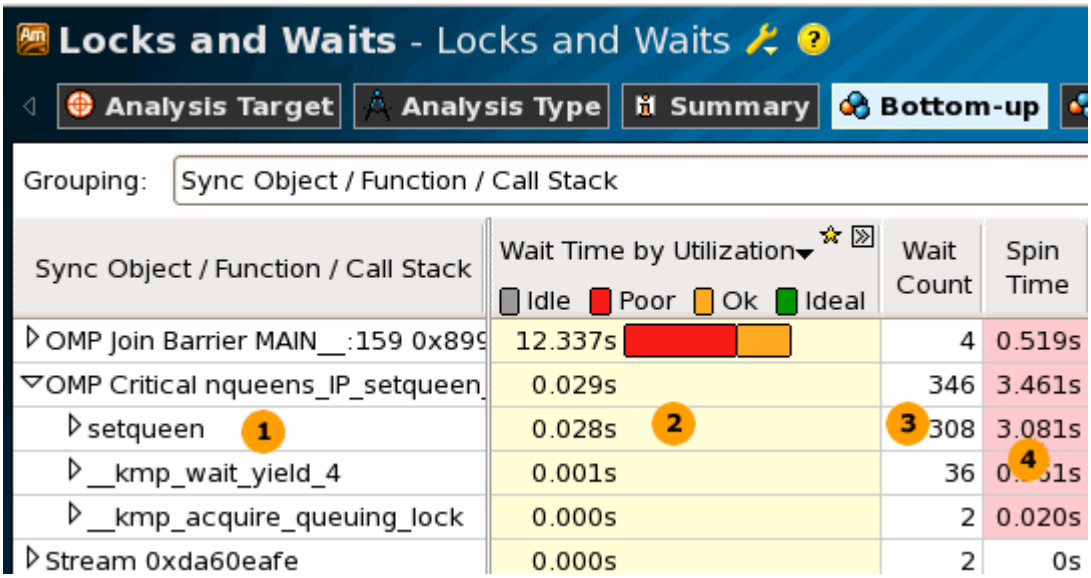
Interpret Locks and Waits Results

When the sample application exits, the Intel® VTune™ Amplifier XE finalizes the results and opens the Locks and Waits viewpoint that is configured to display synchronization objects sorted by Wait time. To interpret the data on the sample code performance, do the following:

1. Identify locks.
2. Analyze source code.

Identify Locks

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



The table below explains the type of data provided in the **Bottom-up** pane:

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.

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

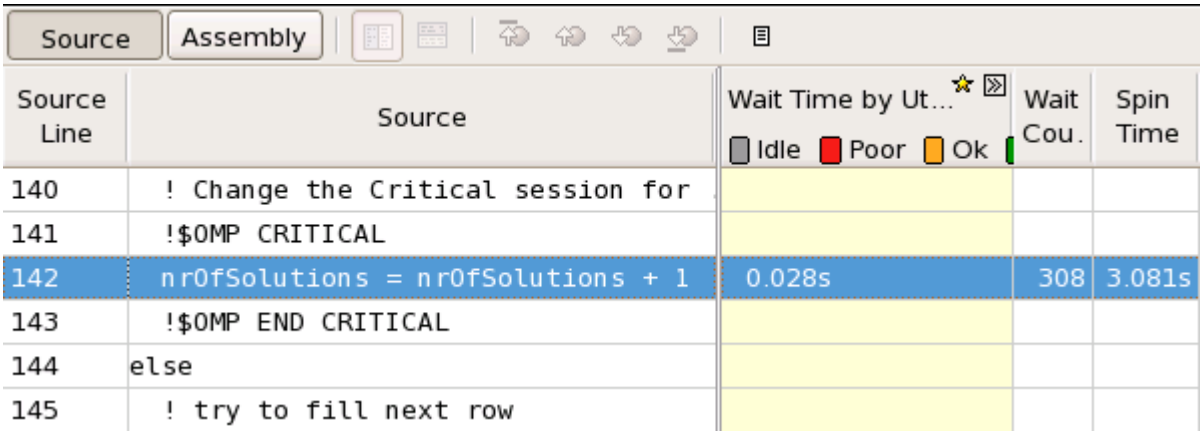
3. Number of times the corresponding system wait API was called. For a lock, it is the number of times the lock was contended and caused a wait. Usually you are recommended to focus your tuning efforts on the waits with both high Wait Time and Wait Count values, especially if they have poor utilization.

4. 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.

In the `nqueens_parallel` sample code, there are two critical wait objects, `OMP Critical nqueens_IP_setqueen` and `OMP Join Barrier`, that caused redundant synchronization and took the longest Wait time and highest Wait count. The bar indicator in the **Wait Time** column indicates that most of the time for this object processor cores were underutilized.

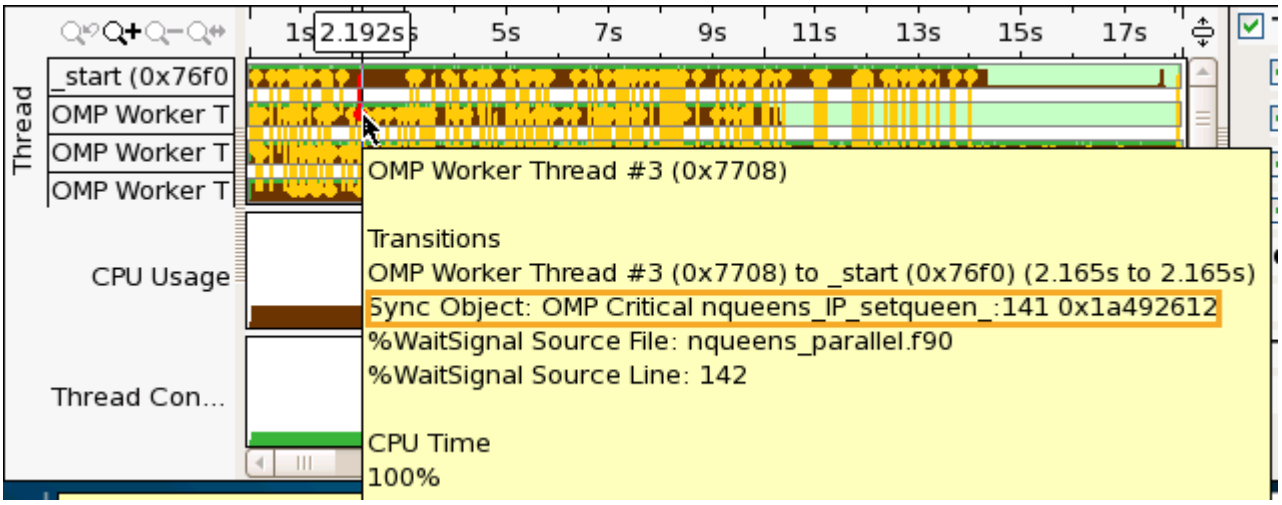
Analyze Source Code

Explore the source of the critical synchronization objects that caused significant Wait time and poor processor utilization. Double-click the `OMP Critical nqueens_IP_setqueen` object to analyze the source of the `setqueen` wait function. Click the button on the **Source** pane toolbar to go to the biggest hotspot code line in the function. VTune Amplifier XE highlights line 142 protected by the OpenMP* critical section.



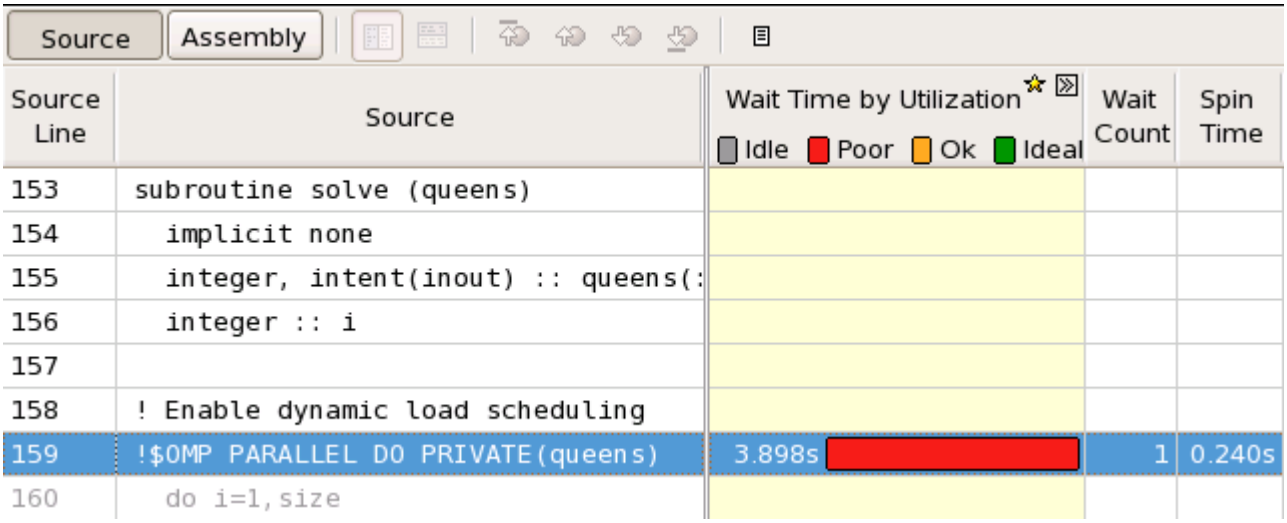
The `setqueen` function was waiting for 0.028 seconds while this code line was executing. During this time, this operation was contended 308 times.

Hover over any transition line in the **Timeline** pane below to explore the infotip and make sure that all the transitions are caused by the `OMP Critical nqueens_IP_setqueen` critical section.

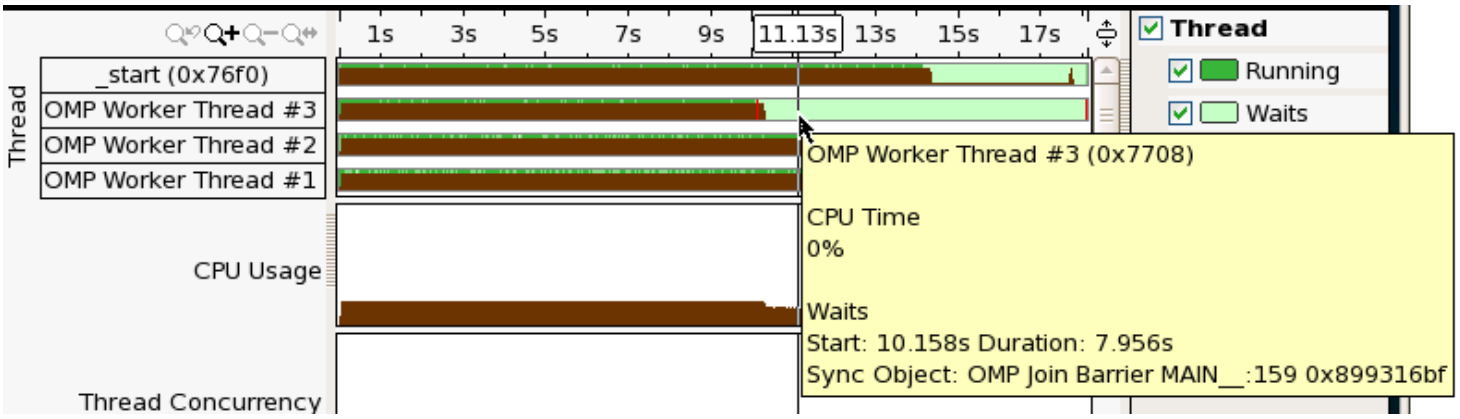


The `OMP Critical nqueens_IP_setqueen` section is the place where the application is serializing. Each thread has to wait for the critical section to be available before it can proceed. Only one thread can be in the critical section at a time.

To explore the next issue, double-click the `OMP Join Barrier` synchronization object to open the source function and go to the hottest line.



The `OMP Join Barrier` object creates a barrier for threads synchronization: a thread should wait until other threads complete execution. The **Timeline** pane illustrates the thread imbalance displaying light-green wait regions for each thread. If you hover over a wait region, the infotip shows that the wait happened on the `OMP Join Barrier` synchronization object.



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

- Elapsed time
- Wait time

Next Step

Remove Lock

Remove Locks

In the Source window, you located the synchronization objects that caused significant waits while the processor cores were underutilized and generated multiple wait count. To resolve the issues, do the following:

1. Open the code editor.

2. Modify the code to remove locks.
3. Recompile the project and check the result.

Open the Code Editor

Click the  **Source Editor** button to open the `nqueens_parallel.f90` file in your default editor:

```
139   if (row == size) then
140     ! Change the Critical session for the Atomic directive OMP ATOMIC
141     !$OMP CRITICAL
142     nrOfSolutions = nrOfSolutions + 1
143     !$OMP END CRITICAL
144   else
145     ! try to fill next row
146     do i=1,size
147       call setQueen (queens, row+1, i)
148     end do
149   end if
150 end subroutine SetQueen
```

Remove Locks

The critical section introduced in line 141 protects the global variable from a race condition in a multithreaded application but it spawns a redundant synchronization. To resolve this issue, you may replace the critical section with an atomic operation as follows:

1. Edit like 141 to replace the `OMP CRITICAL` with the `OMP ATOMIC` directive.
2. Comment out or remove line 143.

A threads barrier, created by OpenMP* directive in line 159, synchronizes the threads but creates a lock with long Wait time. You may resolve this by enabling dynamic load scheduling as follows:

1. Edit line 159 to add the `SCHEDULE(DYNAMIC)` directive to the OpenMP pragma:

```
141     !$OMP ATOMIC
142     nrOfSolutions = nrOfSolutions + 1
143     !!$OMP END CRITICAL
144   else
145     ! try to fill next row
146     do i=1,size
147       call setQueen (queens, row+1, i)
148     end do
149   end if
150 end subroutine SetQueen
151
152 ! Main solver routine
153 subroutine solve (queens)
154   implicit none
155   integer, intent(inout) :: queens(:)
156   integer :: i
157
158 ! Enable dynamic load scheduling
159 !$OMP PARALLEL DO PRIVATE(queens) SCHEDULE(DYNAMIC)
```

2. Save your changes.

Recompile the Project and Check the Result

1. Browse to the directory where you extracted the sample code (for example, `/home/vtune/nqueens_fortran/linux`).
2. Rebuild your target in the release mode using the make command as follows:

   ```
   $ make clean
   ```

   ```
   $ make
   ```

   The `nqueens_parallel` application is rebuilt.
3. Run `nqueens_parallel` as follows:

   ```
   ./nqueens_parallel 15
   ```

   ```
   [vtune@nntpdsd52-082 linux]$ ./nqueens_parallel 15
   Starting nqueens solver for size 15 with 4 thread(s)
   Number of solutions: 2279184
   Correct Result!
   Calculations took 13575ms.
   ```

   System runs the `nqueens_parallel`. Note that execution time reduced from 14940 ms to 13575 ms.

Key Terms

- Wait time

Next Step

Compare with Previous Result

Compare with Previous Result

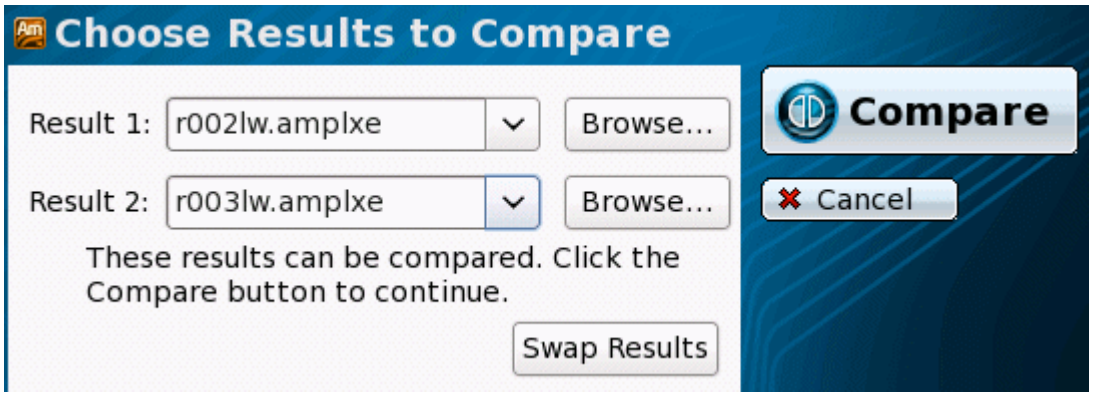
You made sure that removing the critical section gave you 689 ms of optimization in the application execution time. To understand the impact of your changes and how the CPU utilization has changed, re-run the Locks and Waits analysis on the optimized code and compare results:

1. Compare results before and after optimization.
2. Identify the performance gain by metrics.
3. Compare timeline data.

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 VTune Amplifier XE toolbar.

   The **Compare Results** window opens.
3. Specify the Locks and Waits analysis results you want to compare:



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

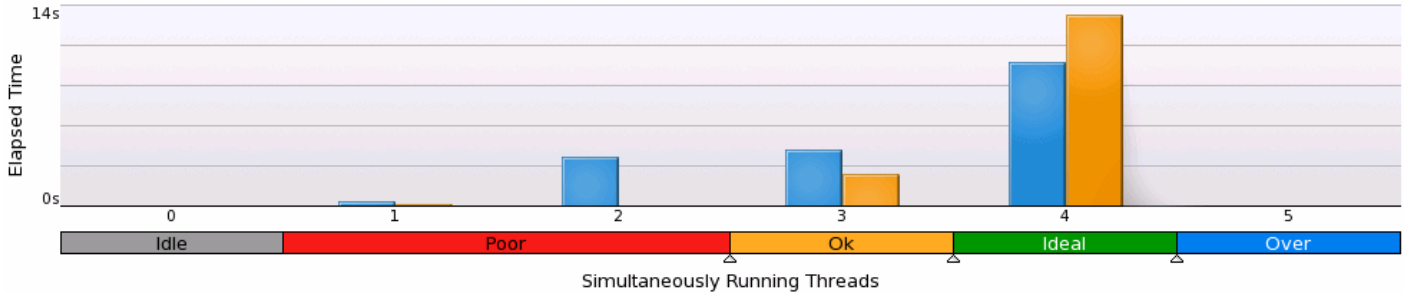
Identify the Performance Gain by Metrics

The **Result Summary** section of the **Summary** window shows that after optimization all critical metric values has reduced significantly. The **Elapsed Time** data shows the optimization of 2 seconds for the whole application. **Wait Time** decreased by 11 seconds, **Wait Count** - by 17.

Elapsed Time:	**18.165s - 16.122s = 2.043s**
Total Thread Count:	Not changed, 5
Wait Time:	30.498s - 19.468s = 11.029s
Spin Time:	3.981s - 3.940s = 0.041s
Wait Count:	443 - 426 = 17
CPU Time:	60.490s - 61.310s = -0.820s
Paused Time:	Not changed, 0s

NOTE The Locks and Waits analysis adds an overhead to the application execution. The overhead often depends on the number of threads and synchronization objects used in the application. This is the reason why Elapsed time data provided in the **Summary** window may differ from the data reported after the application launch outside of the VTune Amplifier XE.

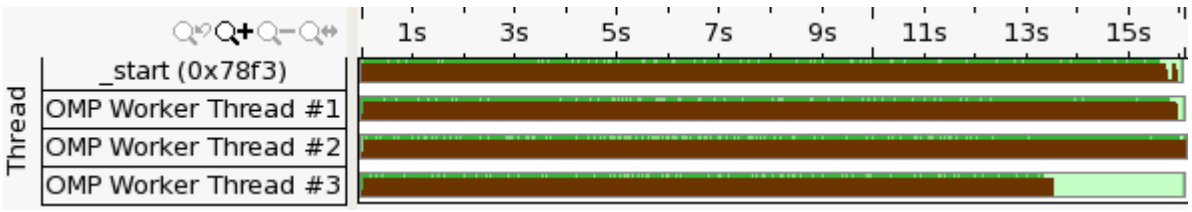
According to the **Thread Concurrency** histogram, after optimization (an orange bar) 4 threads ran in parallel effectively utilizing CPU cores for almost 14 seconds, which is categorized by the VTune Amplifier XE as the Ideal processor utilization. The previous version of the application ran on 4 threads for 10 seconds.



In the **Bottom-up** pane, locate the OpenMP* critical section you identified as a bottleneck in your code. Since you removed it during optimization, the optimized result **r003lw** does not show any performance data for this synchronization object. If you collapse the **Wait Time:Difference by Utilization** column by clicking the button, you see that with the optimized result you got 0.029 seconds of optimization in Wait time. Using dynamic scheduling for the threads barrier gave you 9 seconds of optimization in Wait time.

Compare Timeline Data

Open the optimized result of the Locks and Waits analysis **r003lw**, click the **Bottom-up** tab and analyze the **Timeline** pane.



Disable the transitions display to better see the changes in Wait Time. Though the threads are not fully balanced, the wait regions have reduced significantly.

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

- Elapsed time
- Thread concurrency
- Wait time

Summary

3


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

Step	Tutorial Recap	Key Tutorial Take-aways
1. Prepare for analysis	You set up your environment to enable generating symbol information for your binary files, built the target, created the performance baseline, and created the VTune Amplifier XE project for your analysis target.	• Configure your project properties to get the most accurate results for user binaries and to analyze the performance of your application at the code line level. • Create a performance baseline to compare the application versions before and after optimization. Make sure to use the same workload for each application run. • Use the **Project Properties: Target** tab to choose and configure your analysis target. • Use the **Analysis Type** configuration window to choose, configure, and run the analysis. You can also run the analysis from command line using the `amplxe-cl` command.
2. Find hotspots	You launched the Hotspots data collection that analyzed function calls and CPU time spent in each program unit of your application and identified the following hotspots: • A function that took the most CPU time and could be a good candidate for algorithm tuning. • The code section that took the most CPU time to execute.	• 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 and navigate between hotspots using the **Source** window navigation buttons.
3. Eliminate hotspots	You optimized the algorithm by enabling the OpenMP* library create a private copy of the array. You rebuilt the application and got performance gain of 10395 ms.	Click the  **Source Editor** button to open your default source editor directly from the VTune Amplifier XE **Source** window.
4. Analyze concurrency	You launched the Concurrency analysis and identified poor thread concurrency for the whole application execution. You analyzed the timeline and identified poor thread balance: all OpenMP threads were constantly transferring	• Start your analysis with the **Summary** window. Consider the **Target** concurrency metric specified in the **Thread Concurrency Histogram** as your optimization goal. The **Average** metric is calculated as CPU time / Elapsed time. Use this number as another baseline for your performance measurements. The closer this number to the number of cores, the better.

Step	Tutorial Recap	Key Tutorial Take-aways
	execution to each other and were waiting for all threads to complete execution.	• In the **Bottom-up** window, use the **Filter In by Selection** context menu option to focus on the performance-critical functions in the grid and analyze their performance over time in the **Timeline** pane.
5. Find lock	You ran the Locks and Waits analysis and identified the following hotspots: • Two synchronization objects with the high Wait Time and Wait Count values and poor CPU utilization that could be locks affecting application parallelism. Your next step is to analyze the code of their wait functions. • The code sections that caused significant waits and numerous transitions between threads.	• Use the **Analysis Type** configuration window to choose, configure, and run the analysis. For recently used analysis types, you may use the shortcuts to run a recent analysis: • From the **File** menu, select **New > [recent_analysis_type]**. • In the **Bottom-up** window, 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.
6. Remove lock	You optimized the application execution time by removing the unnecessary critical section that caused redundant synchronization and by adding the dynamic load scheduling.	Double-click the most time-critical synchronization object in the **Bottom-up** pane. This opens the source code for the wait function it belongs to. Use the hotspot navigation buttons to identify the most time-critical code lines.
7. 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.	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.

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

Key Terms

4

baseline: A performance metric used as a basis for comparison of the application versions before and after optimization. Baseline should be measurable and reproducible.

CPU time: The amount of time a thread spends executing on a logical processor. For multiple threads, the CPU time of the threads is summed. The application CPU time is the sum of the CPU time of all the threads that run the application.

Elapsed time: The total time your target ran, calculated as follows: **Wall clock time at end of application - Wall clock time at start of application**.

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

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 VTune 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.

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

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.