



# **Intel(R) VTune(TM) Amplifier XE 2011 Getting Started Tutorials for Windows\* OS**

Document Number: 323906-005US

World Wide Web: <http://developer.intel.com>

Legal Information



# Contents

<b>Legal Information.....</b>	<b>5</b>
<b>Introducing the Intel® VTune™ Amplifier XE.....</b>	<b>7</b>
<b>Prerequisites.....</b>	<b>9</b>
<b>Navigation Quick Start.....</b>	<b>11</b>
<b>Key Terms and Concepts.....</b>	<b>13</b>
<b>Chapter 1: Tutorial: Finding Hotspots</b>	
Learning Objectives.....	17
Workflow Steps to Identify and Analyze Hotspots.....	17
Visual Studio* IDE: Choose Project and Build Application.....	18
Standalone GUI: Build Application and Create New Project.....	24
Run Hotspots Analysis.....	29
Interpret Result Data.....	30
Analyze Code.....	33
Tune Algorithms.....	34
Compare with Previous Result.....	37
Summary.....	39
<b>Chapter 2: Tutorial: Analyzing Locks and Waits</b>	
Learning Objectives.....	41
Workflow Steps to Identify Locks and Waits.....	41
Visual Studio* IDE: Choose Project and Build Application.....	42
Standalone GUI: Build Application and Create New Project.....	48
Run Locks and Waits Analysis.....	53
Interpret Result Data.....	54
Analyze Code.....	57
Remove Lock.....	58
Compare with Previous Result.....	60
Summary.....	63
<b>Chapter 3: Tutorial: Identifying Hardware Issues</b>	
Learning Objectives.....	65
Workflow Steps to Identify Hardware Issues.....	65
Visual Studio* IDE: Choose Project and Build Application.....	66
Standalone GUI: Build Application and Create New Project.....	70
Run General Exploration Analysis.....	74
Interpret Results.....	75
Analyze Code.....	78
Resolve Issue.....	79
Resolve Next Issue.....	82
Summary.....	85
<b>Chapter 4: More Resources</b>	
Getting Help.....	87
Product Website and Support.....	88

**Chapter 5: Intel® VTune™ Amplifier XE Tutorials Troubleshooting**  
Troubleshooting.....89

# 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/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/](http://www.intel.com/products/processor_number/)

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.

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, Puma, skool, the skool logo, SMARTi, Sound Mark, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or 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.

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

Java is a registered trademark of Oracle and/or its affiliates.

Copyright (C) 2010-2011, Intel Corporation. All rights reserved.



# *Introducing the Intel® VTune™ Amplifier XE*

---



The 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. 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](#)
- [Identifying Hardware Issues](#)

Check <http://software.intel.com/en-us/articles/intel-software-product-tutorials/> for the printable version (PDF) of product tutorials.

## **See Also**

[Getting Help](#)





# Prerequisites

---



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



**NOTE** The instructions and screen shots in these tutorials refer to the Visual Studio\* 2005 integrated development environment (IDE). They may slightly differ for other versions of Visual Studio IDE or for the standalone version of the Intel® VTune™ Amplifier XE. See online help for details.

---

## Required Tools

You need the following tools to use these tutorials:

- Intel® VTune™ Amplifier XE
- Sample code included with the VTune Amplifier XE. VTune Amplifier XE provides the following sample applications:
  - `tachyon` application used for the *Finding Hotspots* and *Analyzing Locks and Waits* tutorials
  - `matrix` application used for the *Identifying Hardware Issues* tutorial
- VTune Amplifier XE Help
- Microsoft Visual Studio\* 2005 or later

## 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.zip` and `matrix_vtune_amp_xe.zip` files from the `samples \<locale>\C++` folder in the IntelVTune Amplifier XE installation directory to a writable directory or share on your system.

The default installation directory is `C:\Program Files\Intel\VTune Amplifier XE 2011` (on certain systems, instead of `Program Files`, the folder name is `Program Files (x86)`).


2. Extract the sample(s) from the `.zip` 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:

- For Microsoft Visual Studio\*: VTune Amplifier XE integrates into Visual Studio when installation completes. To configure and run an analysis, open your solution and go to **Tools > Intel VTune Amplifier XE 2011 > New Analysis...** from the Visual Studio menu or click the  **New Analysis** button from the VTune Amplifier XE toolbar. See the [Navigation Quick Start](#) for more details.
- For the standalone interface: From the **Start** menu, select **All Programs > Intel Parallel Studio XE 2011 > Intel VTune Amplifier XE 2011**.

### To access VTune Amplifier XE Help:

See the [Getting Help](#) topic.

### Required Skills and Knowledge

These tutorials are designed for developers with the following skills and knowledge:

- Basic understanding of the Microsoft Visual Studio\* 2005 development environment (IDE), including how to:
  - Open a project/solution.
  - Display the **Solution Explorer** and **Output** windows.
  - Compile and link a project.
  - Ensure a project compiled successfully.
  - Access the **Document Explorer** window.

# Navigation Quick Start



## Intel® VTune™ Amplifier XE /Microsoft Visual Studio\* 2005 Integration



**NOTE** This topic describes integration into Microsoft Visual Studio\* 2005. Integration to other version of Visual Studio IDE or the standalone VTune Amplifier XE interface may slightly differ.

The VTune Amplifier XE integrates into the Visual Studio\* development environment (IDE) and can be accessed from the menus, toolbar, and **Solution Explorer** in the following manner:

Function / Call Stack	CPU Time by Utilization
multiply2	47.868s
ThreadFunction ← BaseThreadStar	47.868s
init_arr	0.092s
[Unknown]	0s
main	0s
CsrClientCallServer	0s
RtlpWaitForCriticalSection	0s

**A**

Use the VTune Amplifier XE toolbar to configure and control result collection.

**B**

VTune Amplifier XE results \*.amp1xe show up in the **Solution Explorer** under the **My Amplifier XE Results** folder. To configure and control result collection, right-click the project in the **Solution Explorer** and select the **Intel VTune Amplifier XE 2011** menu from the pop-up menu. To manage previously collected results, right-click the result (for example, r002hs.amp1xe) and select the required command from the pop-up menu.

- C** 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.
- D** Click the buttons on navigation toolbars to change window views and toggle window panes on and off.
- E** In the Timeline pane, analyze the thread activity and transitions presented for the user-mode sampling and tracing analysis results (for example, Hotspots, Concurrency, Locks and Waits) or analyze the distribution of the application performance per metric over time for the event-based sampling analysis results (for example, Memory Access, Bandwidth Breakdown).
- F** Use the Call Stack pane to view call paths for a function selected in the grid.
- G** Use the filter toolbar to filter out the result data according to the selected categories.
- H** In Microsoft Visual Studio\* 2005/2008, use the **Dynamic Help** window to access help topics related to the current VTune Amplifier XE window/pane.

# 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: **Wall clock time at end of application – Wall clock time at start of application.**

**hotspot:** A section of code that took a long time to execute. Some hotspots may indicate bottlenecks and can be removed, while other hotspots inevitably take a long time to execute due to their nature.

**target:** A *target* is an executable file you analyze using the Intel® VTune™ 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, 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.

## Key Concept: CPU Usage

For the user-mode sampling and tracing analysis types, the Intel® 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. You can change the utilization ranges by dragging the slider in the CPU Usage histogram in the Summary window.

Utilization Type	Default color	Description
Idle		All CPUs are waiting - no threads are running.
Poor		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.
OK		Acceptable (OK) usage. By default, OK usage is when the number of simultaneously running CPUs is between 51-85% of the target CPU usage.
Ideal		Ideal usage. By default, Ideal usage is when the number of simultaneously running CPUs is between 86-100% of the target CPU usage.

## 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: Event-based Metrics

When analyzing data collected during a hardware event-based sampling analysis, the VTune Amplifier XE uses the performance metrics. 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 (in pink) and provides recommendations how to fix it.

Each column in the Bottom-up pane provides data per metric. To read the metric description and see the formula used for the metric calculation, mouse over the metric column header. To read the description of the hardware issue and see the threshold formula used for this issue, mouse over the link cell in the grid.

For the full list of metrics used by the VTune Amplifier XE, see the *Hardware Event-based Metrics* topic in the online help.

### Key Concept: Event-based Sampling Analysis

VTune Amplifier XE introduces a set of advanced hardware analysis types based on the event-based sampling data collection and targeted for the Intel® Core™ 2 processor family, processors based on the Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Sandy Bridge. 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 event-based metrics* defined by Intel architects (for example, Clockticks per Instructions Retired, Contested Accesses, and so on).

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.

For more information on the event-based sampling analysis, see the *Hardware Event-based Sampling Collection* topic in the online help.

### Key Concept: Event Skid

Event skid is the recording of an event not exactly on the code line that caused the event. Event skids may even result in a caller function event being recorded in the callee function.

Event skid is caused by a number of factors:

- The delay in propagating the event out of the processor's microcode through the interrupt controller (APIC) and back into the processor.
- The current instruction retirement cycle must be completed.
- When the interrupt is received, the processor must serialize its instruction stream which causes a flushing of the execution pipeline.

The Intel(R) processors support accurate event location for some events. These events are called precise events. See the online help for more details.

### Key Concept: Finalization

*Finalization* is the process of the Intel® VTune™ 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: 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®VTune™ 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® VTune™ 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.

Utilization Type	Default color	Description
Idle		All threads in the application are waiting - no threads are running. There can be only one bar in the Thread Concurrency histogram 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.





# Tutorial: Finding Hotspots

---



## Learning Objectives

---



This tutorial shows how to use the Hotspots analysis of the Intel® VTune™ 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.

Sample application: `tachyon`.

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.

### Start Here

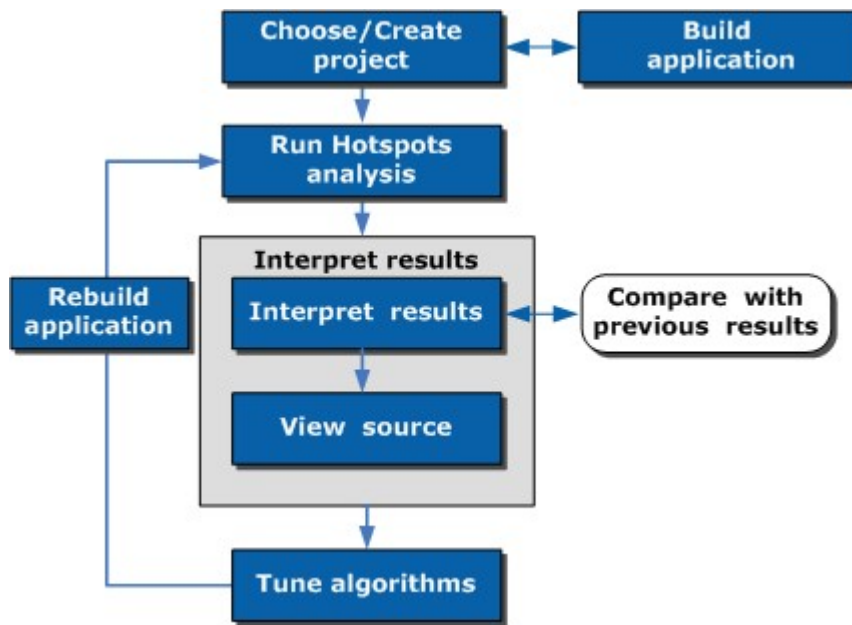
[Workflow Steps to Identify and Analyze Hotspots](#)

## Workflow Steps to Identify and Analyze Hotspots

---



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 ray-tracer application named `tachyon`.



1. Do one of the following:
  - Visual Studio\* IDE: Choose a project, verify settings, and build application
  - Standalone GUI: Build an application to analyze for hotspots and create a new VTune Amplifier XE project
2. Choose and run the Hotspots analysis.
3. Interpret the result data.
4. View and analyze code of the performance-critical function.
5. Modify the code to tune the algorithms or rebuild the code with Intel® Compiler.
6. Re-build the target, re-run the Hotspots analysis, and compare the result data before and after optimization.

### 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

## Visual Studio\* IDE: Choose Project and Build Application



Before you start analyzing your application target for hotspots, do the following:

1. Choose a project with the analysis target in the Visual Studio IDE.
2. Configure the Microsoft Visual Studio\* environment to download the debug information for system libraries so that VTune Amplifier XE can properly identify system functions and classify and attribute functions.
3. Configure Visual Studio project properties to generate the debug information for your application so that VTune Amplifier XE can open the source code.
4. Build the target in the release mode with full optimizations, which is recommended for performance analysis.
5. Run the application without debugging to create a performance baseline.

For this tutorial, your target is a ray-tracer application, `tachyon`. To learn how to install and set up the sample code, see [Prerequisites](#).



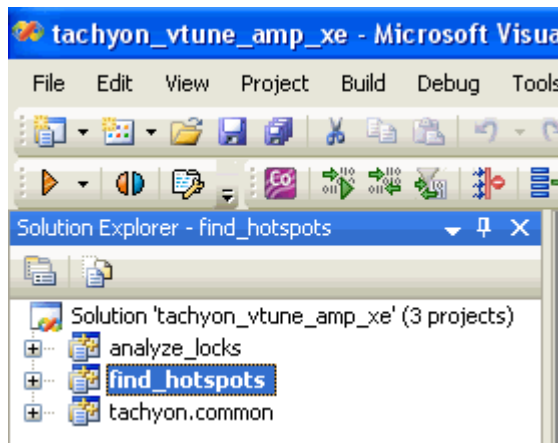
- The steps below are provided for Microsoft Visual Studio 2005. They may slightly differ for other versions of Visual Studio.
- Steps provided by this tutorial are generic and applicable to any application. You may choose to follow the proposed workflow using your own application.

## Choose a Project

1. From the Visual Studio menu, select **File > Open > Project/Solution....**

The **Open Project** dialog box opens.

2. In the **Open Project** dialog box, browse to the location you used to extract the `tachyon_vtune_amp_xe.zip` file and select the `tachyon_vtune_amp_xe.sln` file.



The solution is added to Visual Studio IDE and shows up in the Solution Explorer.

3. In the Solution Explorer, right-click the **find\_hotspots** project and select **Project > Set as StartUp Project**.

**find\_hotspots** appears in bold in the Solution Explorer.

When you choose a project in Visual Studio IDE, the VTune Amplifier XE automatically creates the `config.amplxproj` project file and sets the `find_hotspots` application as an analysis target in the project properties.


## Enable Downloading the Debug Information for System Libraries

1. Go to **Tools > Options....**

The **Options** dialog box opens.

2. From the left pane, select **Debugging > Symbols**.

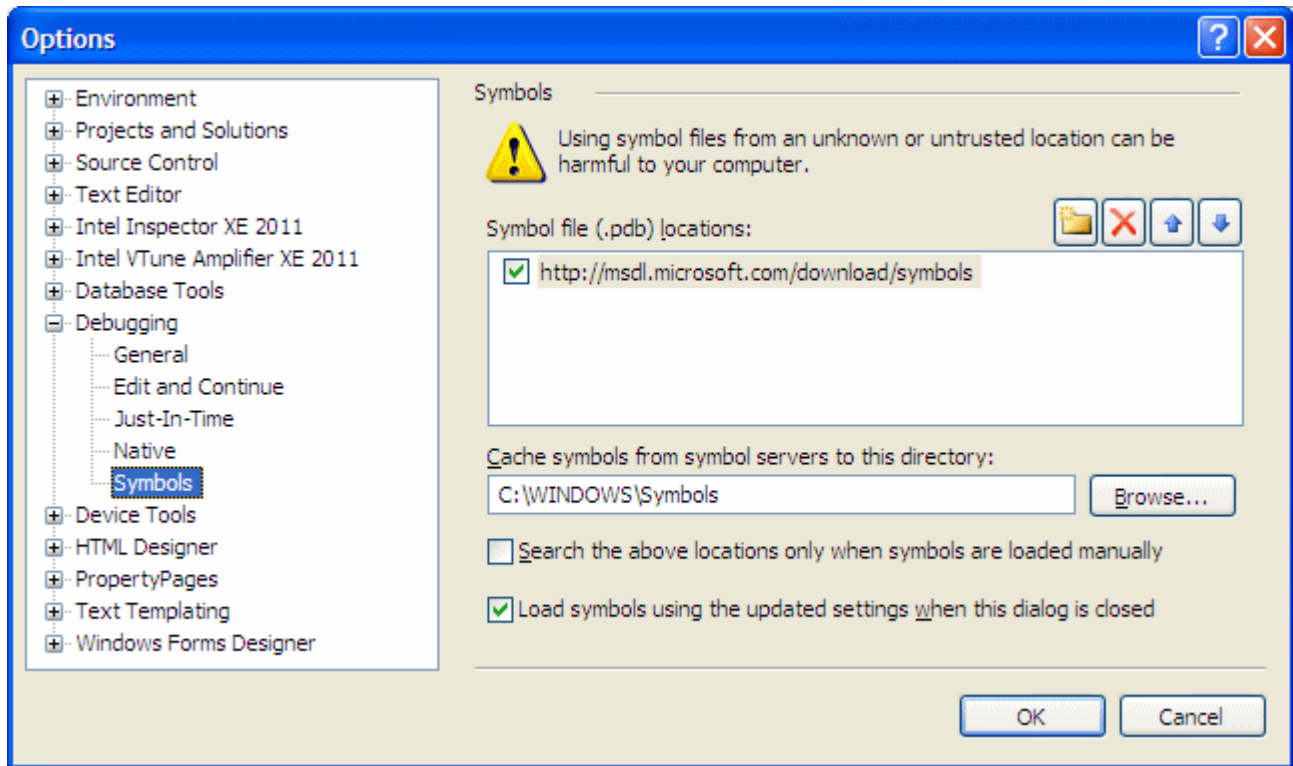
3.

In the **Symbol file (.pdb) locations** field, click the  button and specify the following address: `http://msdl.microsoft.com/download/symbols`.

4. Make sure the added address is checked.

5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.

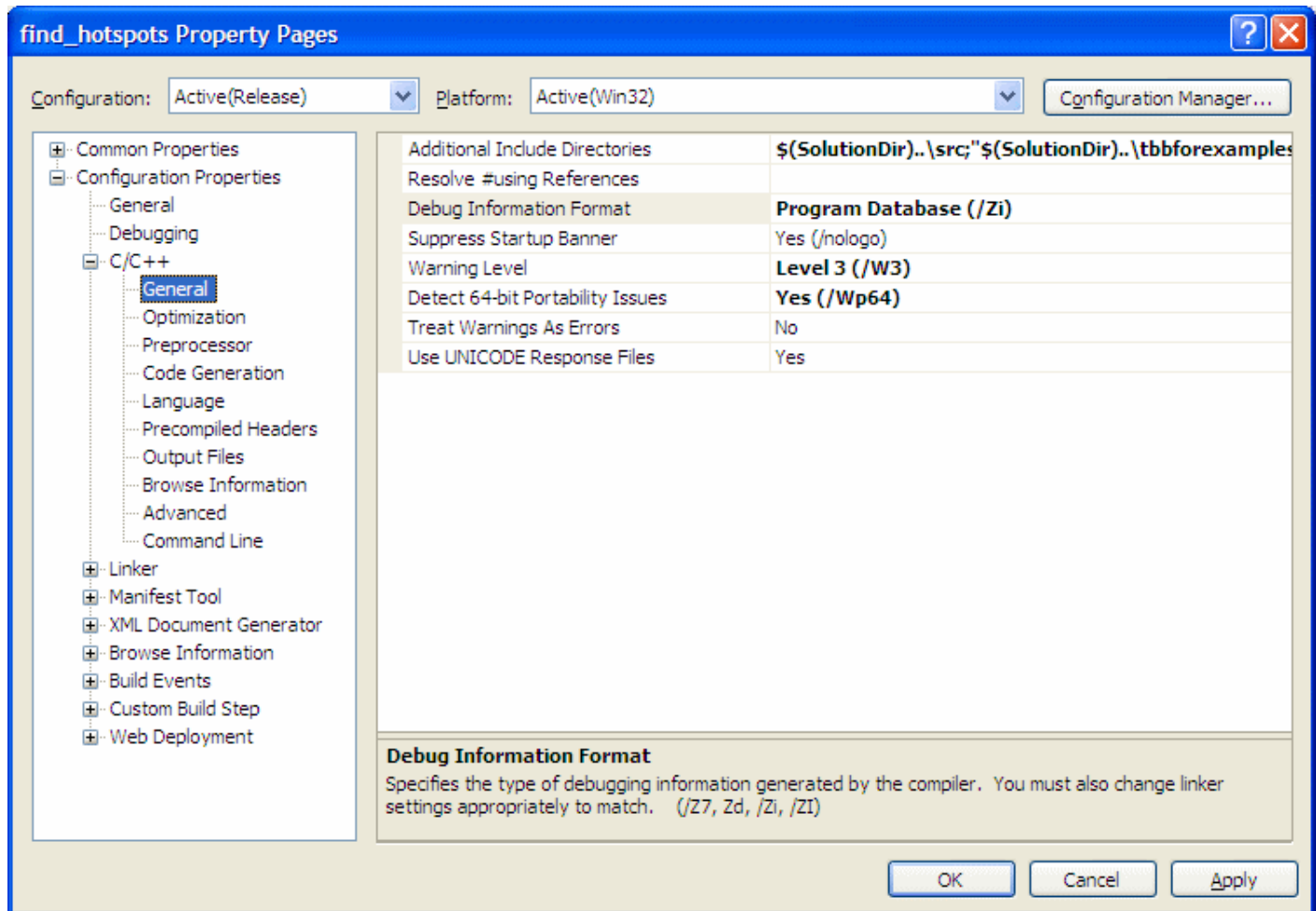
6. For Microsoft Visual Studio\* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.



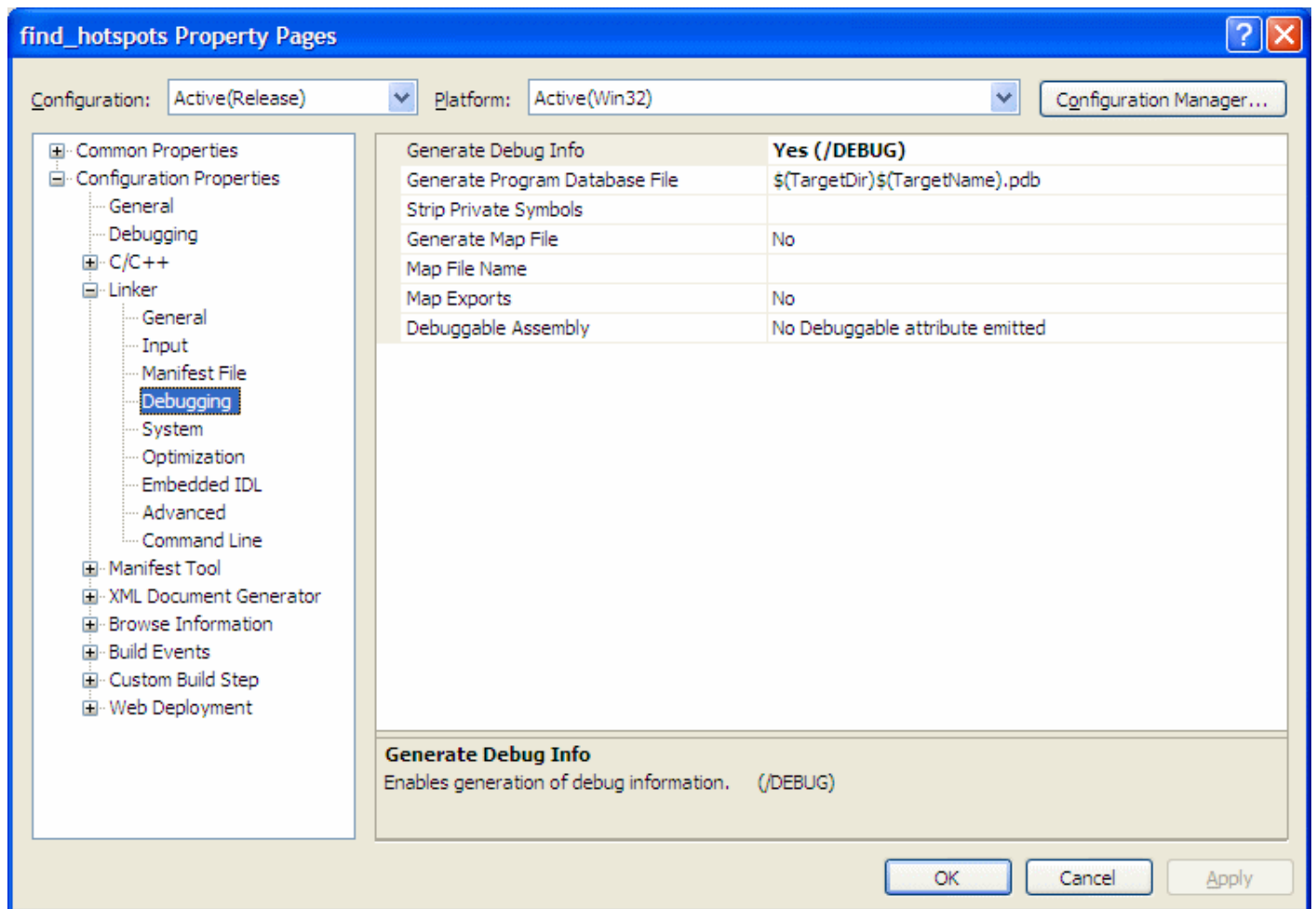
7. Click **OK**.

### Enable Generating Debug Information for Your Binary Files

1. Select the **find\_hotspots** project and go to **Project > Properties**.
2. From the **find\_hotspots Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **find\_hotspots Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/ZI)**.




4. From the **find\_hotspots Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



## Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build find\_hotspots**.


The `tachyon_find_hotspots.exe` application is built.

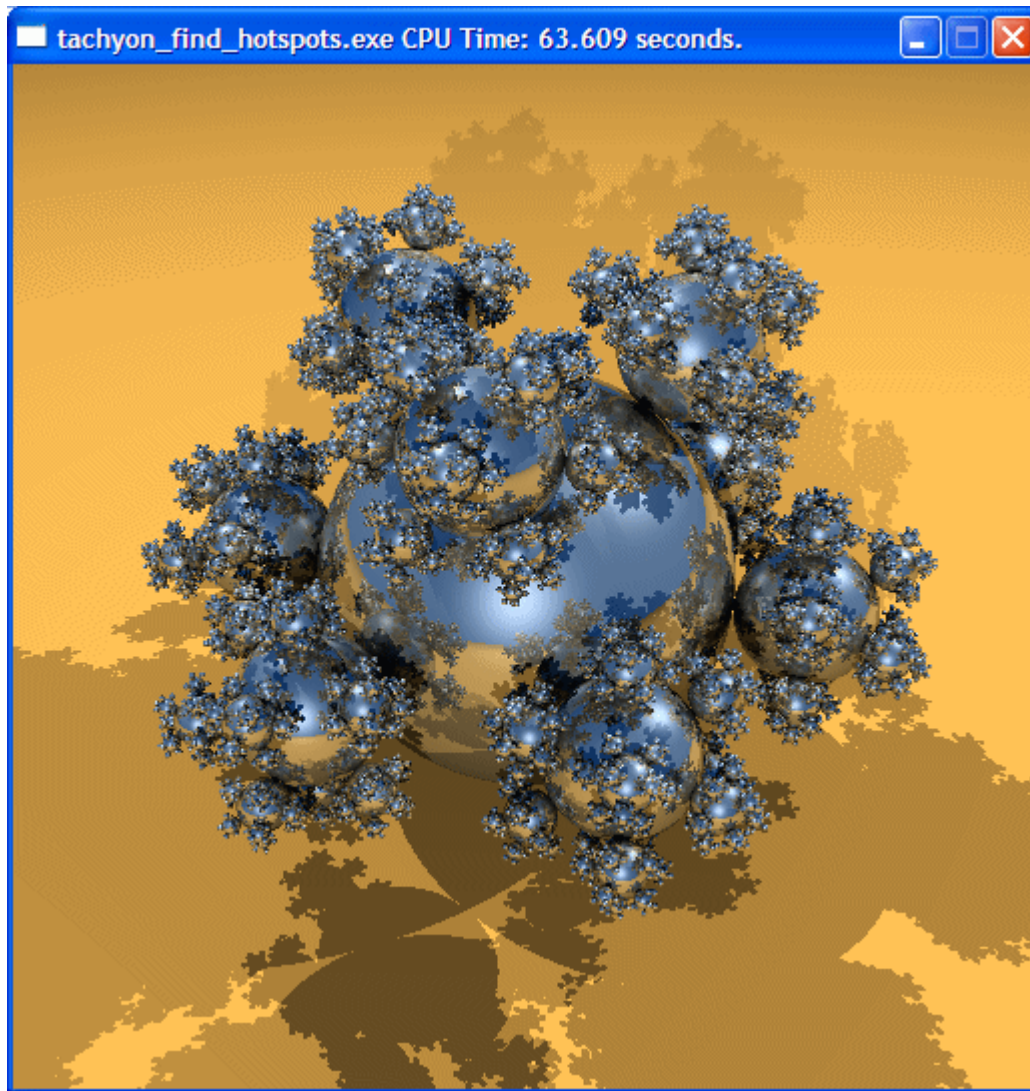
 **NOTE** The build configuration for `tachyon` may initially be set to Debug, which is typically used for development. When analyzing performance issues with the VTune Amplifier XE, you are recommended to use the Release build with normal optimizations. In this way, the VTune Amplifier XE is able to analyze the realistic performance of your application.

## Create a Performance Baseline

1. From the Visual Studio menu, select **Debug > Start Without Debugging**.

The `tachyon_find_hotspots.exe` 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. For the `tachyon_find_hotspots.exe` executable in the figure above, the execution time is 63.609 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 chose the target for the Hotspots analysis, set up your environment to enable generating symbol information for system libraries and your binary files, built the target in the Release mode, and created the performance baseline. Your application is ready for analysis.

## Key Terms and Concepts

- Term: [target](#)
- Concept: [Hotspots Analysis](#)

## Next Step

[Run Hotspots Analysis](#)



## Standalone GUI: Build Application and Create New Project

---



Before you start analyzing your application target for hotspots, do the following:

**1. Build application.**

If you build the code in Visual Studio\*, make sure to:

- [Configure the Microsoft Visual Studio\\* environment to download the debug information for system libraries](#) so that VTune Amplifier XE can properly identify system functions and classify and attribute functions.
- [Configure Visual Studio project properties to generate the debug information for your application](#) so that VTune Amplifier XE can open the source code.
- [Build the target in the release mode with full optimizations](#), which is recommended for performance analysis.

**2. Run the application without debugging to create a performance baseline.**

**3. Create a VTune Amplifier XE project.**



---

**NOTE** The steps below are provided for Microsoft Visual Studio 2005. They may differ slightly for other versions of Visual Studio.

---


### Enable Downloading the Debug Information for System Libraries

**1. Go to **Tools > Options...****

The **Options** dialog box opens.

**2. From the left pane, select **Debugging > Symbols**.**

**3.**

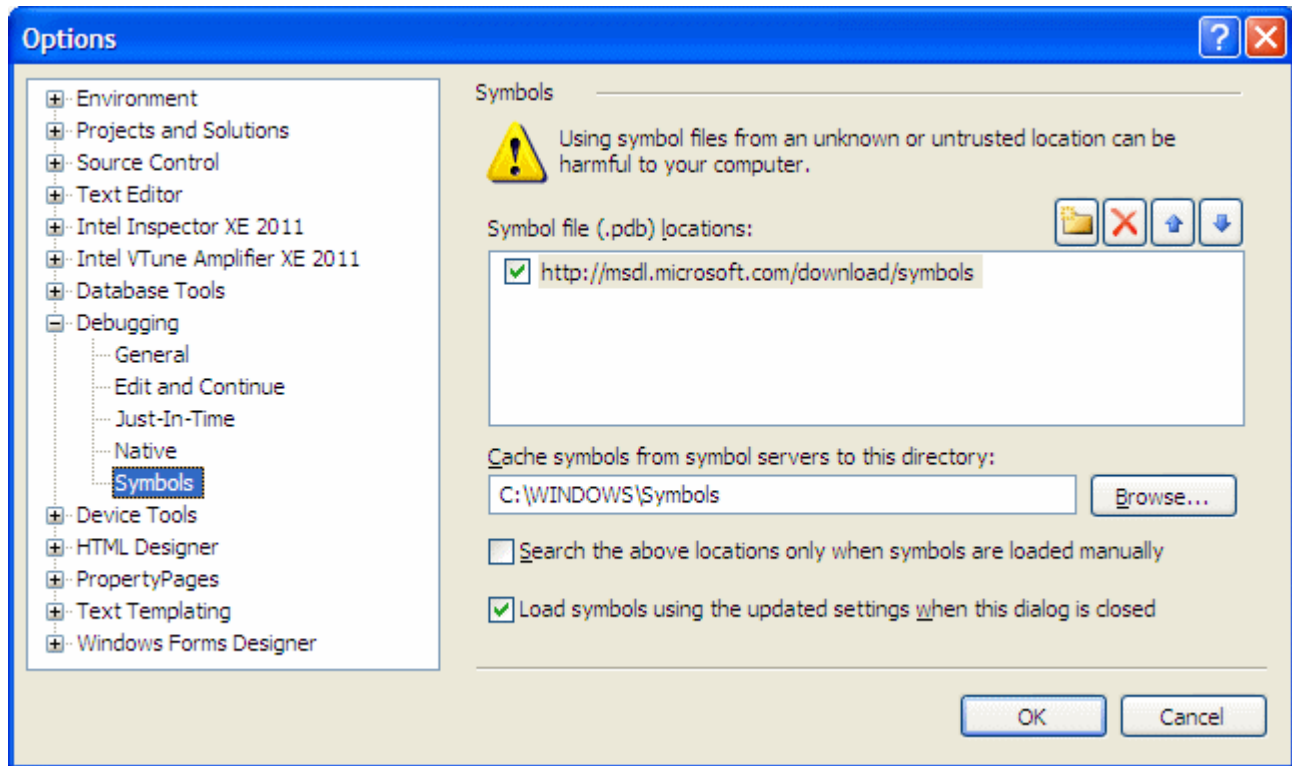
In the **Symbol file (.pdb) locations** field, click the  button and specify the following address: `http://msdl.microsoft.com/download/symbols`.

**4. Make sure the added address is checked.**

**5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.**

**6. For Microsoft Visual Studio\* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.**

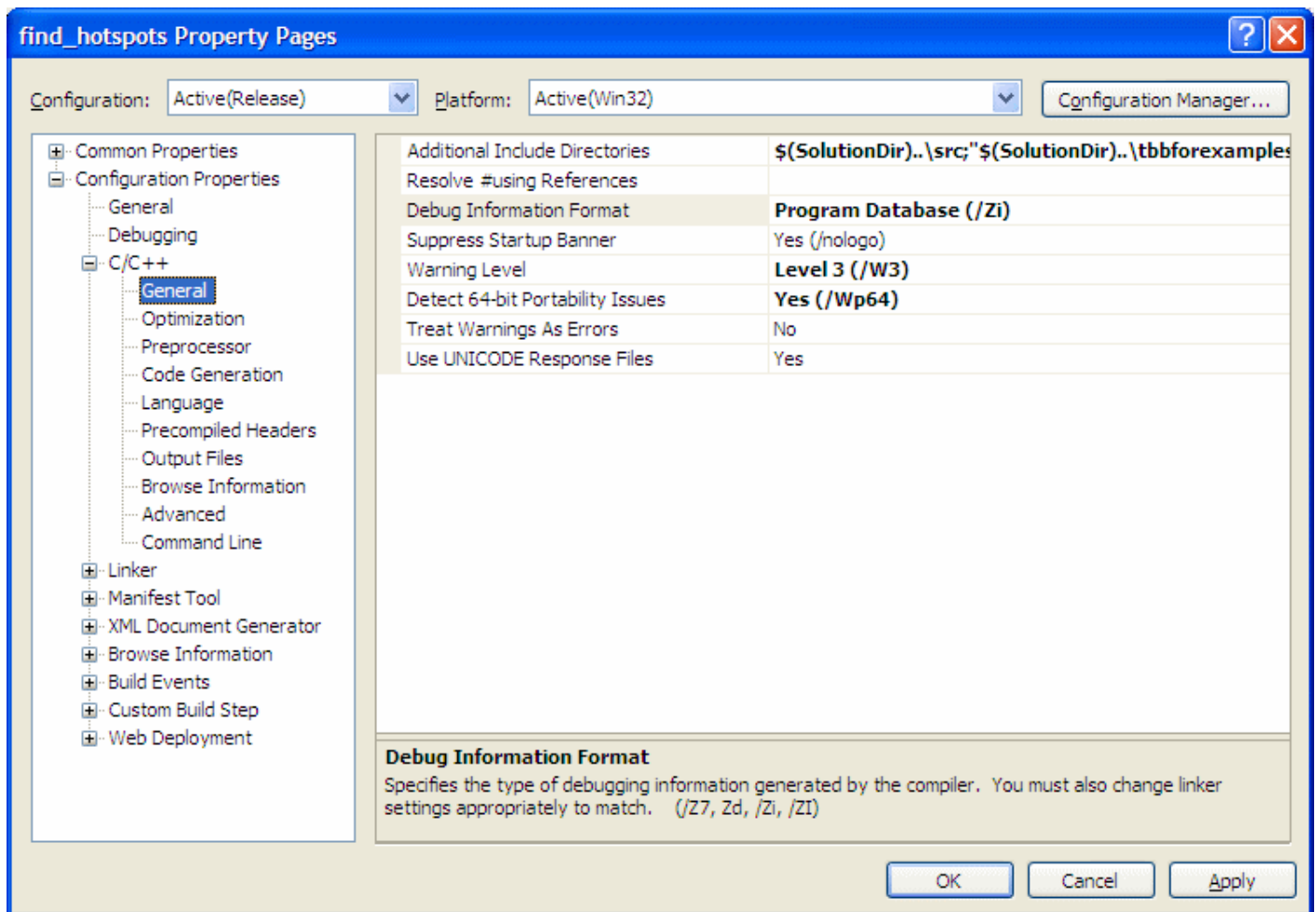




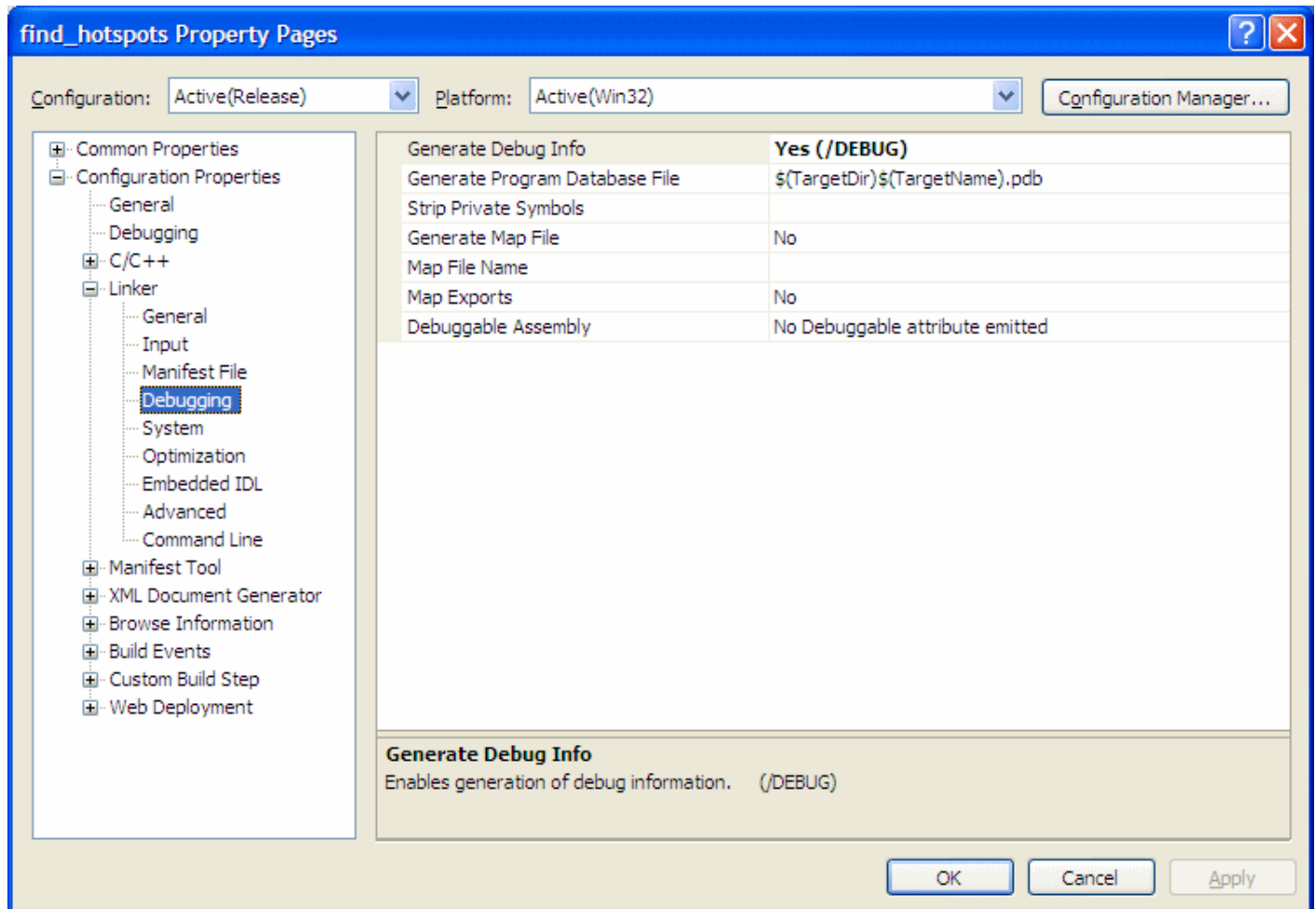
7. Click **OK**.

### Enable Generating Debug Information for Your Binary Files

1. Select the **find\_hotspots** project and go to **Project > Properties**.
2. From the **find\_hotspots Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **find\_hotspots Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.



4. From the **find\_hotspots Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



## Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build find\_hotspots**.

The `tachyon_find_hotspots.exe` application is built.



**NOTE** The build configuration for `tachyon` may initially be set to Debug, which is typically used for development. When analyzing performance issues with the VTune Amplifier XE, you are recommended to use the Release build with normal optimizations. In this way, the VTune Amplifier XE is able to analyze the realistic performance of your application.

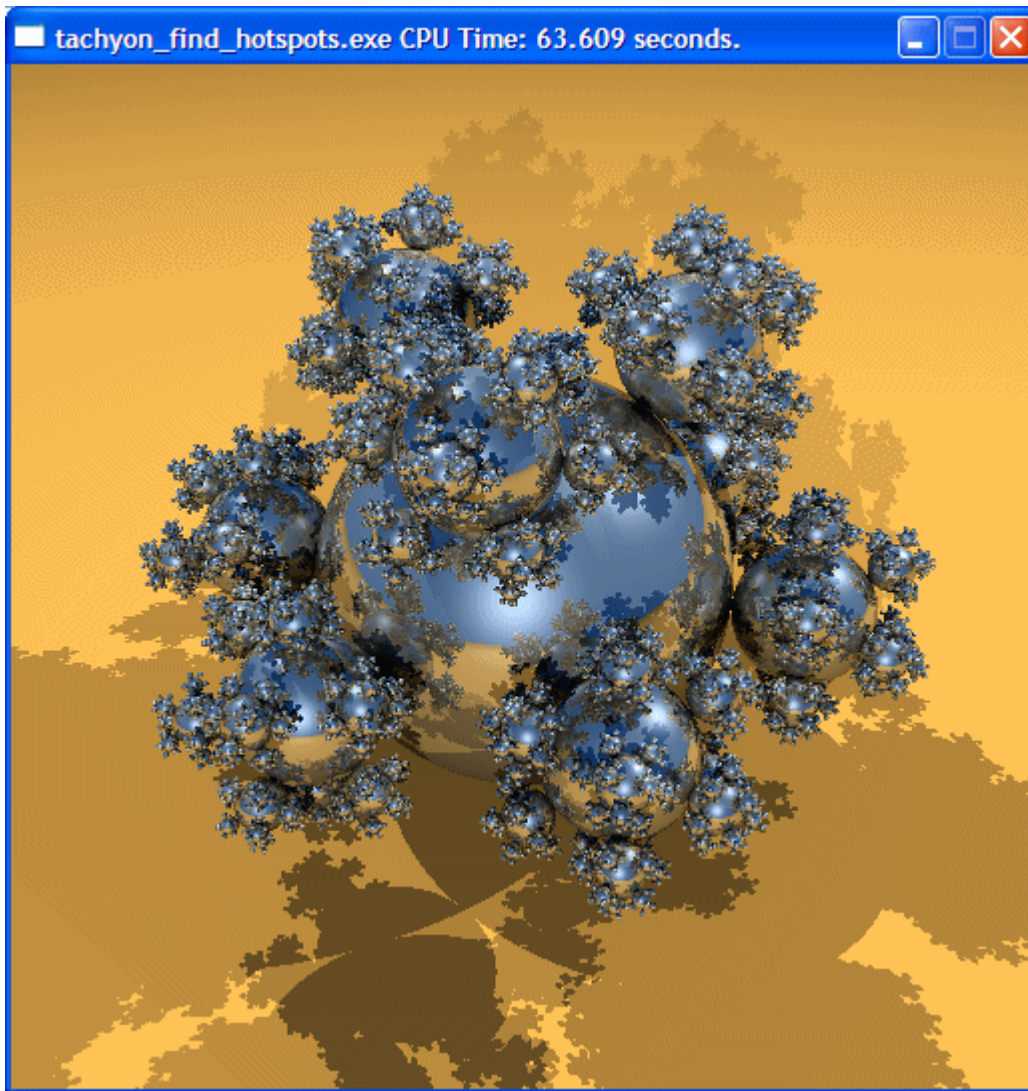
## Create a Performance Baseline

1. From the Visual Studio menu, select **Debug > Start Without Debugging**.


The `tachyon_find_hotspots.exe` 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. For the `tachyon_find_hotspots.exe` executable in the figure above, the execution time is 63.609 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. From the **Start** menu select **Intel Parallel Studio XE 2011 > Intel VTune Amplifier XE 2011** to launch the VTune Amplifier XE standalone GUI.
2. Create a new project via **File > New > Project...**

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

3. 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 `%USERPROFILE%\My Documents\My Amplifier XE Projects` directory and opens the **Project Properties: Target** dialog box.

4. 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>\find_hotspots.exe`, for example: `C:\samples\tachyon_vtune_amp_xe\vc8\find_hotspots_win32_Release\find_hotspots.exe`.

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

## Recap

You set up your environment to enable generating symbol information for system libraries and your binary files, 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 and Concepts

- Term: [target](#), [baseline](#)

## Next Step


Run Hotspots Analysis

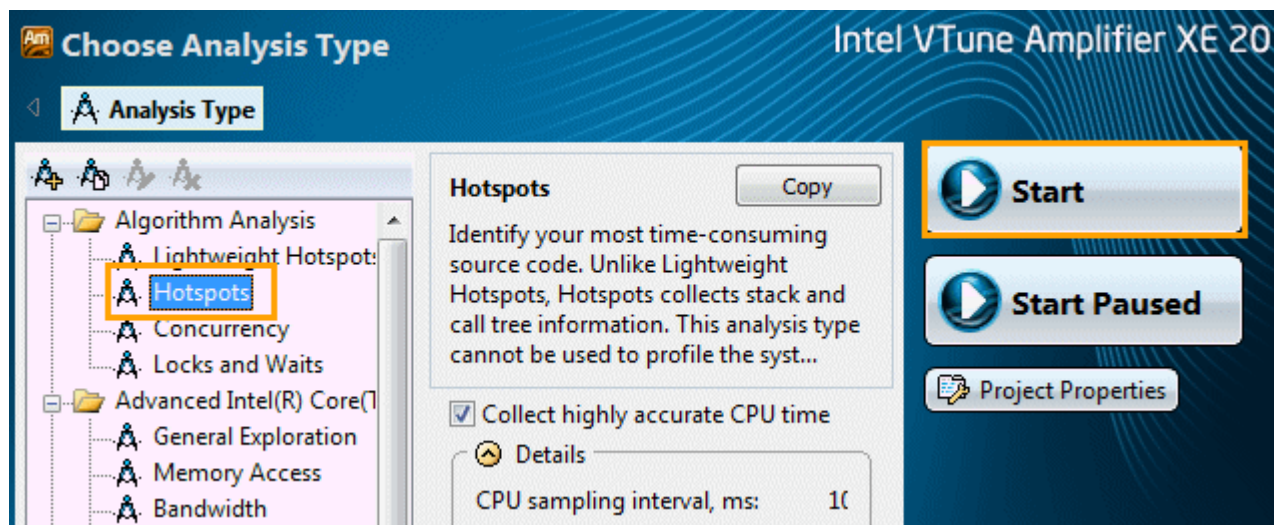
# Run Hotspots Analysis



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**.
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](#)

## Next Step

[Interpret Result Data](#)

# Interpret Result Data



When the sample application exits, the Intel® VTune™ 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 four CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

## Understand the Basic Hotspots Metrics

Start analysis with the Summary window. To interpret the data, hover over the question mark icons to read the pop-up help and better understand what each performance metric means.

**Elapsed Time:** **74.608s**

CPU Time: 64.907s  
Total Thread Count: 3

Note that **CPU Time** for the sample application is equal to 64.907 seconds. It is the sum of CPU time for all application threads. **Total Thread Count** is 3, 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.

### Top Hotspots

This section lists the most active functions

<u>Function</u>	<u>CPU Time</u>
initialize_2D_buffer	27.671s
grid_intersect	17.475s
sphere_intersect	11.644s
grid_bounds_intersect	1.786s
video::main_loop	1.698s
[Others]	4.633s



For the sample application, the `initialize_2D_buffer` function, which took 27.671 seconds to execute, shows up at the top of the list as the hottest function.

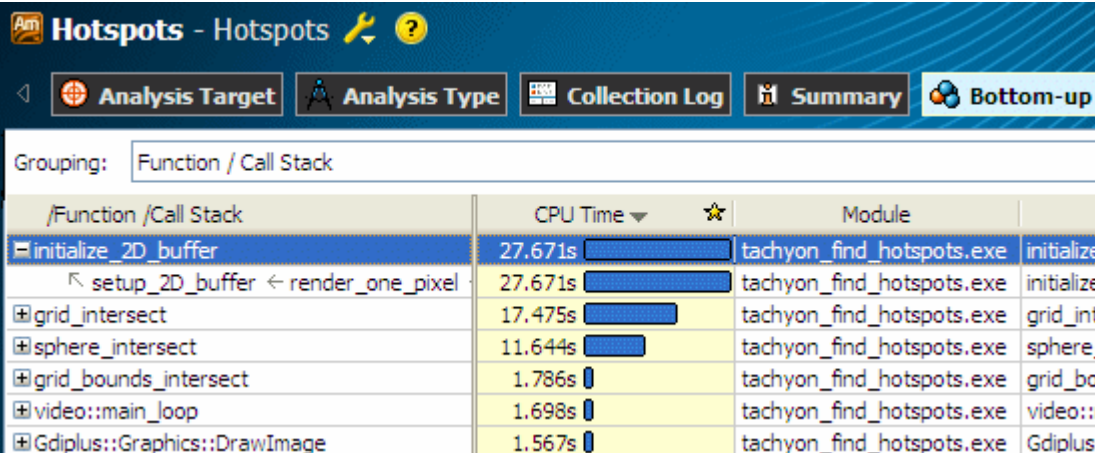
The [Others] entry at the bottom shows the sum of CPU time for all functions not listed in the table.

## 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. You may change the grouping level using the **Grouping** drop-down menu at the top of the grid.

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 27.671 seconds to execute. Click the plus sign  $\oplus$  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.



/Function /Call Stack	CPU Time	Module	
initialize_2D_buffer	27.671s	tachyon_find_hotspots.exe	initialize_...
└─ setup_2D_buffer ← render_one_pixel	27.671s	tachyon_find_hotspots.exe	initialize_...
└─ grid_intersect	17.475s	tachyon_find_hotspots.exe	grid_inte...
└─ sphere_intersect	11.644s	tachyon_find_hotspots.exe	sphere_...
└─ grid_bounds_intersect	1.786s	tachyon_find_hotspots.exe	grid_bo...
└─ video::main_loop	1.698s	tachyon_find_hotspots.exe	video::n...
└─ Gdiplus::Graphics::DrawImage	1.567s	tachyon_find_hotspots.exe	Gdiplus:

Select the `initialize_2D_buffer` function in the grid and explore the data provided in the Call Stack pane on the right.

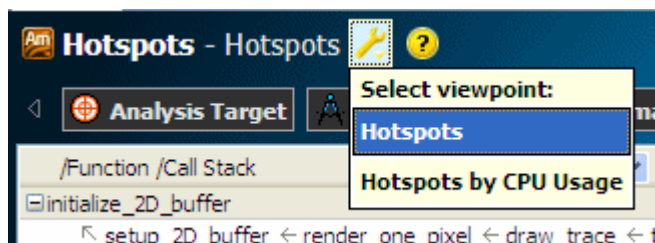
The Call Stack pane displays full stack data for each hotspot function, enables you to navigate between function call stacks and understand the impact of each stack to the function CPU time. The stack functions in the Call Stack pane are represented in the following format:

`<module>!<function> - <file>:<line number>`, where the line number corresponds to the line calling the next function in the stack.

```
tachyon_find_hotspots.exe!initialize_2D_buffer(unsigned int * const,unsigned int *) - fin ...
tachyon_find_hotspots.exe!setup_2D_buffer(void) - global.cpp:86
tachyon_find_hotspots.exe!render_one_pixel - find_hotspots.cpp:131
```

For the sample application, the hottest function `initialize_2D_buffer` is called at line 86 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

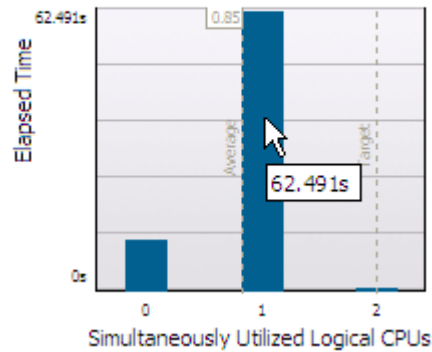
terms of the CPU usage. Explore this viewpoint to determine how your application utilized available cores and identify the most serial code.


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. Ideally, the highest bar of your chart should match the Target level.


The `tachyon_find_hotspots` application ran mostly on one logical CPU. If you hover over the highest bar, you see that it spent 62.491 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.

### CPU Usage Histogram

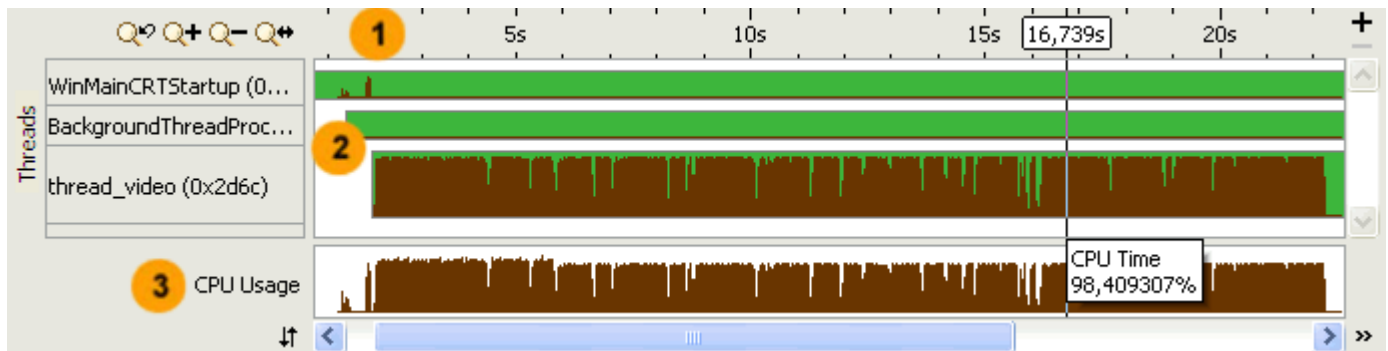
This histogram represents a breakdown of the simultaneously. CPU Usage may be higher than



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.



- 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. Green zones show the time threads are active.
- 3 **CPU Usage** area that 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 each moment of time.

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 Timeline analysis also identifies the `thread_video` thread as the most active. The tooltip shows that CPU time values rarely exceed 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](#)

## Next Step

[Analyze Code](#)

# 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

Line	Source	CPU Time	Address	Line	Assembly	CPU Time
81	{		0x1120	82	mov eax, esi	0.058s
82	for (int j = 0; j < mem_array	0.116s	0x1122	82	mov ecx, 0xb4	0.058s
83	{		0x1127		Block 3:	
84	mem_array [j+mem_array j	27.555s	0x1127	84	mov ebx, dword ptr [edx]	8.822s
85	}		0x1129	84	add ebx, 0x2	2.214s
86	}		0x112c	84	mov dword ptr [eax], ebx	0.237s
87	/*****		0x112e	84	add eax, 0x2d	3.634s
88			0x1133	84	sub ecx, 0x1	12.186s
89			0x1136	84	jnz 0x1127 <Block 3>	
90	// Faster method of filling array		0x1138		Block 4:	
91	// The for loops are interchanged		0x1138	84	add esi, 0x4	
92	// Array IS filled in consecutive		0x113b	84	sub edi, 0x1	0.404s
93	/*****		0x113e	84	jnz 0x1120 <Block 2>	0.057s
94	for (int j = 0; j < mem_array_j_m		0x1140		Block 5:	
95			0x1140	84	mov edi, 0	

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**.




**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.

---

- 3 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.
- 4 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.
- 5 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 highlighting the code line that took the most CPU time. For the `initialize_2D_buffer` function, the hottest code line is 84. 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](#)

## Next Step

[Tune Algorithms](#)

# Tune Algorithms


---



In the Source window, you identified that in the `initialize_2D_buffer` hotspot function the code line 84 took the most CPU time. Focus on this line and do the following:

1. Open the code editor.
2. Resolve the performance problem using any of these options:
  - Optimize the algorithm used in this code section.
  - Recompile the code with the Intel® Compiler.

## Open the Code Editor

In the Source window, click the  Source Editor button to open the `find_hotspots.cpp` file in the default code editor at the hotspot line:

```

75 void initialize_2D_buffer (unsigned int mem_array [], unsigned int *fill_value)
76 {
77     // First (slower) method of filling array
78     // Array is NOT filled in consecutive memory address order
79     /*****/
80     for (int i = 0; i < mem_array_i_max; i++)
81     {
82         for (int j = 0; j < mem_array_j_max; j++)
83         {
84             mem_array [j*mem_array_j_max+i] = *fill_value + 2;
85         }
86     }
87     /*****/
88
89
90     // Faster method of filling array
91     // The for loops are interchanged
92     // Array IS filled in consecutive memory address order
93     /*****/
94     for (int j = 0; j < mem_array_j_max; j++)
95     {
96         for (int i = 0; i < mem_array_i_max; i++)
97         {
98             mem_array [j*mem_array_j_max+i] = *fill_value + 2;
99         }
100     }
101     /*****/
102 }
103

```

Hotspot line 84 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, use one of the following methods:

### Option 1: Optimize your algorithm

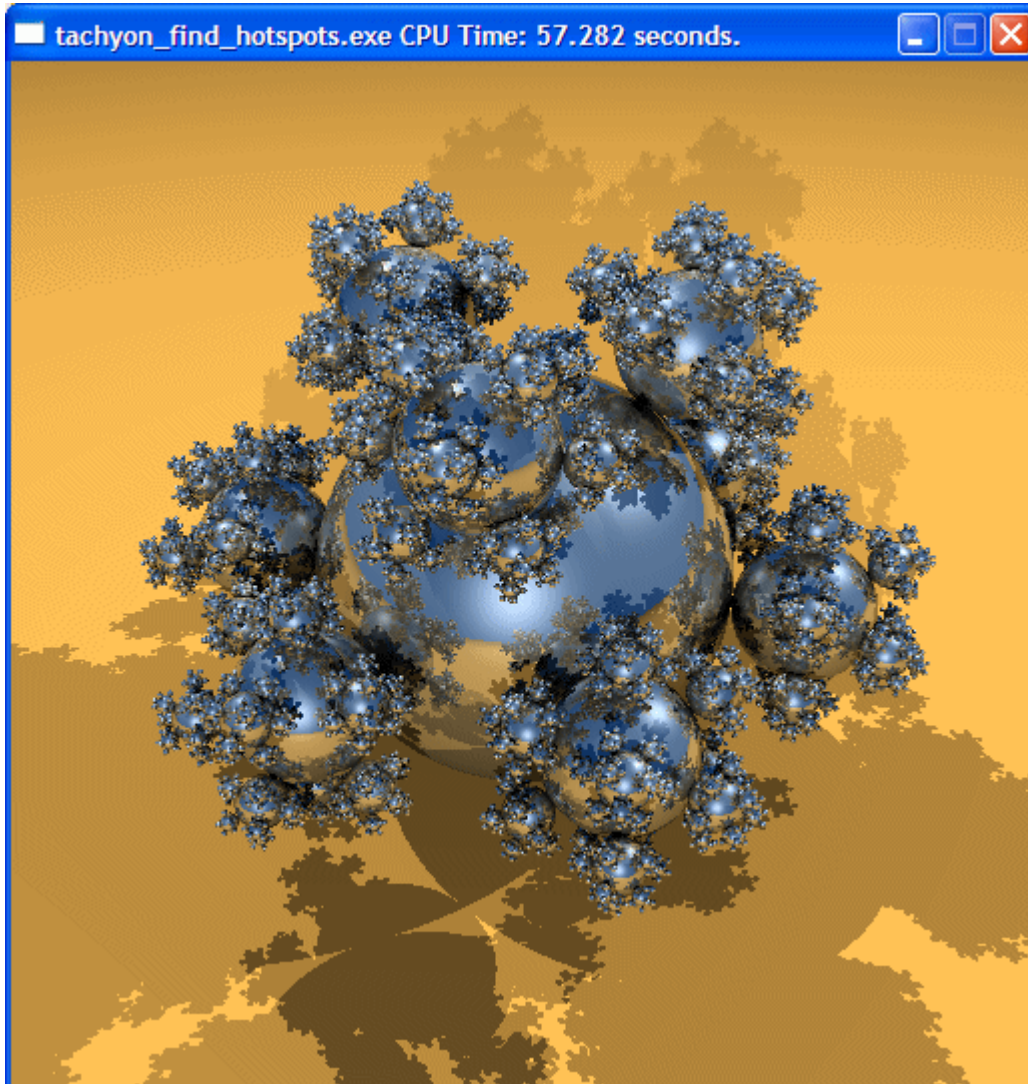
1. Edit line 79 to comment out code lines 82-88 marked as a "First (slower) method".
2. Edit line 95 to uncomment code lines 98-104 marked as a "Faster method".

In this step, you interchange the `for` loops to initialize the code in sequential memory locations.

3. From the Visual Studio menu, select **Build > Rebuild find\_hotspots**.

The project is rebuilt.

4. From Visual Studio **Debug** menu, select **Start Without Debugging** to run the application.



Visual Studio runs the `tachyon_find_hotspots.exe`. Note that execution time has reduced from 63.609 seconds to 57.282 seconds.

### Option 2: Recompile the code with Intel® Compiler

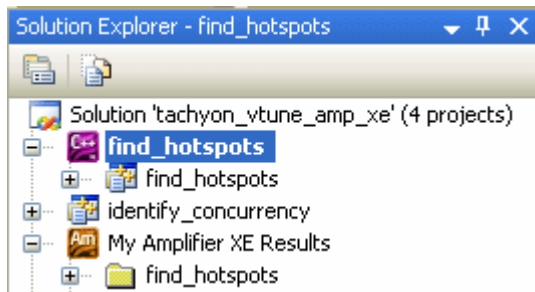
This option assumes that you have Intel® Composer XE installed. Composer XE is part of Intel® Parallel Studio XE. By default, the Intel® Compiler, one of the Composer components, uses powerful optimization switches, which typically provides some gain in performance. For more details on the Intel compiler, see the Intel Composer documentation.

As an alternative, you may consider running the default Microsoft Visual Studio compiler applying more aggressive optimization switches.

To recompile the code with the Intel compiler:

1. From Visual Studio **Project** menu, select **Intel Composer XE> Use Intel C++....**
2. In the **Confirmation** window, click **OK** to confirm your choice.

The project in Solution Explorer appears with the ComposerXE icon:



- From the Visual Studio menu, select **Build > Rebuild find\_hotspots**.

The project is rebuilt with the Intel compiler.

- From the Visual Studio menu, select **Debug > Start Without Debugging**.

Visual Studio runs the `tachyon_find_hotspots.exe`. Note that the execution time reduced.

## Recap

You interchanged the loops in the hotspot function, rebuilt the application, and got performance gain of 6 seconds. You also considered an alternative optimization technique using the Intel C++ compiler.

## Key Terms and Concepts

- Term: `hotspot`

## Next Step

[Compare with Previous Result](#)

### 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


## Compare with Previous Result



You optimized your code to apply a loop interchange mechanism that gave you 6 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.

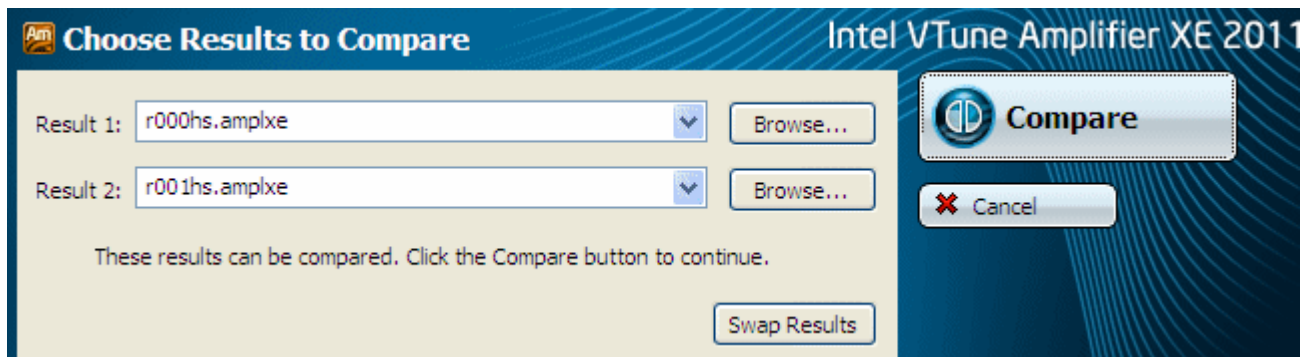
### Compare Results Before and After Optimization

- Run the Hotspots analysis on the modified code.
- Click the **Compare Results**  button on the Intel® VTune™ Amplifier XE toolbar.

The **Compare Results** window opens.

- Specify the Hotspots analysis results you want to compare and click the **Compare Results** button:





The Hotspots Bottom-up window opens, showing the CPU time usage across the two results and the differences side by side.

Function	CPU Time:Difference	CPU Time: r000hs	CPU Time: r001hs	Module
initialize_2D_buffer	6.350s	27.671s	21.321s	tachyon_find_hotspots.e
tri_intersect	0.118s	0.342s	0.225s	tachyon_find_hotspots.e
Gdiplus::Graphics::Draw	0.101s	1.567s	1.465s	tachyon_find_hotspots.e
VScale	0.092s	0.242s	0.150s	tachyon_find_hotspots.e
InternalWndProc	0.047s	0.065s	0.018s	tachyon_find_hotspots.e
closest_intersection	0.045s	0.045s	0s	tachyon_find_hotspots.e
video::terminate	0.033s	0.085s	0.051s	tachyon_find_hotspots.e
rt_getmem	0.029s	0.029s	0s	tachyon_find_hotspots.e
camray	0.026s	0.069s	0.043s	tachyon_find_hotspots.e
ColorAddS	0.023s	0.046s	0.023s	tachyon_find_hotspots.e
draw_trace	0.023s	0.035s	0.012s	tachyon_find_hotspots.e
Selected 1 row(s):	6.350s	27.671s	21.321s	

- 1 Difference in CPU time between the two results in the following format: <Difference CPU Time> = <Result 1 CPU Time> - <Result 2 CPU Time>.
- 2 CPU time for the initial version of the tachyon\_find\_hotspots.exe application.
- 3 CPU time for the optimized version of the tachyon\_find\_hotspots.exe.

### Identify the Performance Gain

Explore the Bottom-up pane to compare CPU time data for the first hotspot: CPU Time:r000hs - CPU Time:r001hs = CPU Time: Difference. 27.671s - 21.321s = 6.350s, which means that you got the optimization of ~6 seconds for the initialize\_2D\_buffer function.

If you switch to the Summary window, you see that the Elapsed time also shows 3.6 seconds of optimization for the whole application execution:

**Elapsed Time:** 34.441s - 30.841s = 3.600s  
 Total Thread Count: Not changed, 3  
 CPU Time: 23.262s - 19.830s = 3.433s  
 Paused Time: Not changed, 0s

## 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](#)

## Next Step

Read [Summary](#)

# Summary

---



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

## Step 1. Choose and Build Your Target

- Configure the Microsoft\* symbol server and your project properties to get the most accurate results for system and 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. For Visual Studio\* projects, the analysis target settings are inherited automatically.


## 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.
- Consider using Intel® Compiler, part of the Intel® Composer XE, to optimize your tuning algorithms. Explore the compiler documentation for more details.

## 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.

**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



# Tutorial: Analyzing Locks and Waits

---

# 2



## Learning Objectives

---



This tutorial shows how to use the Locks and Waits analysis of the Intel® VTune™ 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.

Sample application: `tachyon`.

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.

### Start Here

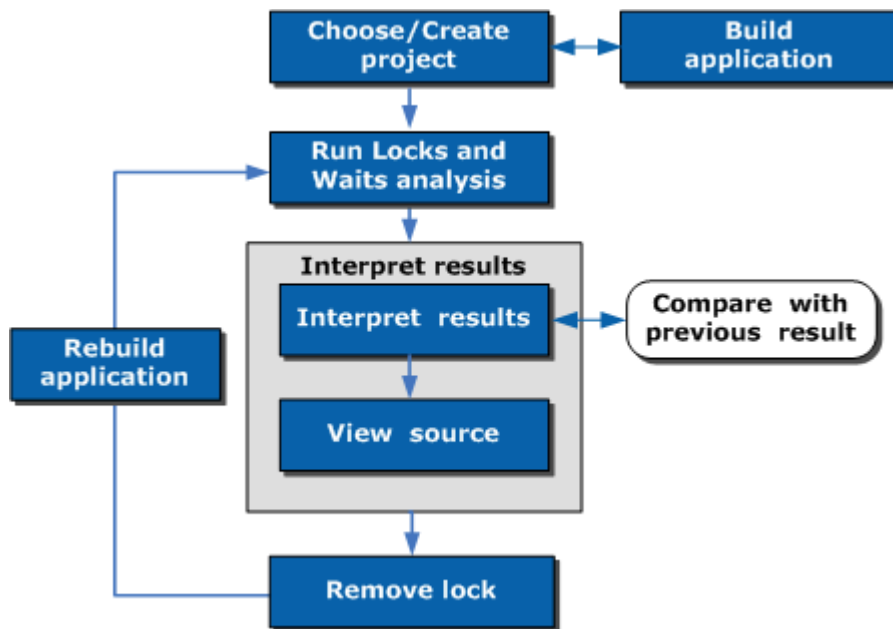
[Workflow Steps to Identify Locks and Waits](#)

## Workflow Steps to Identify Locks and Waits

---



You can use the Intel® VTune™ 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. 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.
2. Run the Locks and Waits analysis.
3. Interpret the result data.
4. View and analyze code of the performance-critical function.
5. Modify the code to remove the lock.
6. Re-build the target, re-run the Locks and Waits analysis, and compare the result data before and after optimization.

## Visual Studio\* IDE: Choose Project and Build Application



Before you start analyzing your application for locks, do the following:

1. Choose a project with the analysis target in the Visual Studio IDE.
2. Configure the Microsoft Visual Studio\* environment to download the debug information for system libraries so that VTune Amplifier XE can properly identify system functions and classify and attribute functions.
3. Configure Visual Studio project properties to generate the debug information for your application so that VTune Amplifier XE can open the source code.
4. Build the target in the release mode with full optimizations, which is recommended for performance analysis.
5. Run the application without debugging to create a performance baseline.

For this tutorial, your target is a ray-tracer application, *tachyon*. To learn how to install and set up the sample code, see [Prerequisites](#).



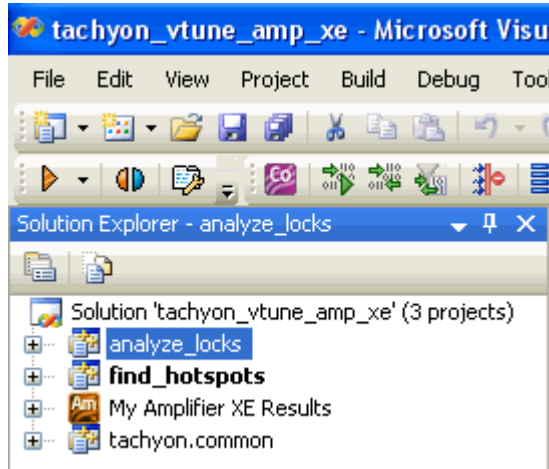
- The steps below are provided for Microsoft Visual Studio\* 2005. Steps for other versions of Visual Studio IDE may slightly differ. See online help for details.
- Steps provided by this tutorial are generic and applicable to any application. You may choose to follow the proposed workflow using your own application.

## Choose a Project

1. From the Visual Studio menu, select **File > Open > Project/Solution....**

The **Open Project** dialog box opens.

2. In the **Open Project** dialog box, browse to the location you used to unzip the `tachyon_vtune_amp_xe.zip` file and select the `tachyon_vtune_amp_xe.sln` file.



The solution is added to Visual Studio and shows up in the Solution Explorer.

3. In Solution Explorer, right-click the **analyze\_locks** project and select **Project > Set as StartUp Project**.

**analyze\_locks** appears in bold in Solution Explorer.


## Enable Downloading the Debug Information for System Libraries

1. Go to **Tools > Options....**

The **Options** dialog box opens.

2. From the left pane, select **Debugging > Symbols**.

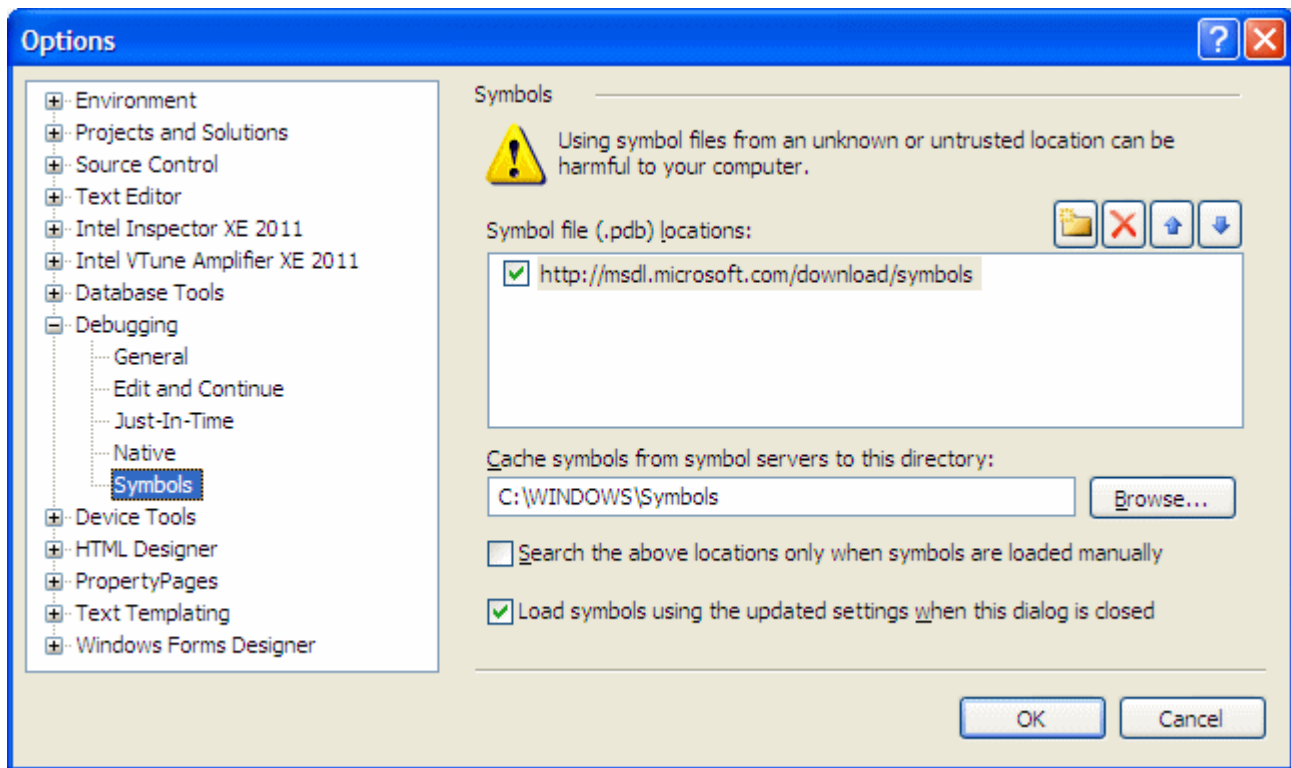
3.

In the **Symbol file (.pdb) locations** field, click the  button and specify the following address: `http://msdl.microsoft.com/download/symbols`.

4. Make sure the added address is checked.

5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.

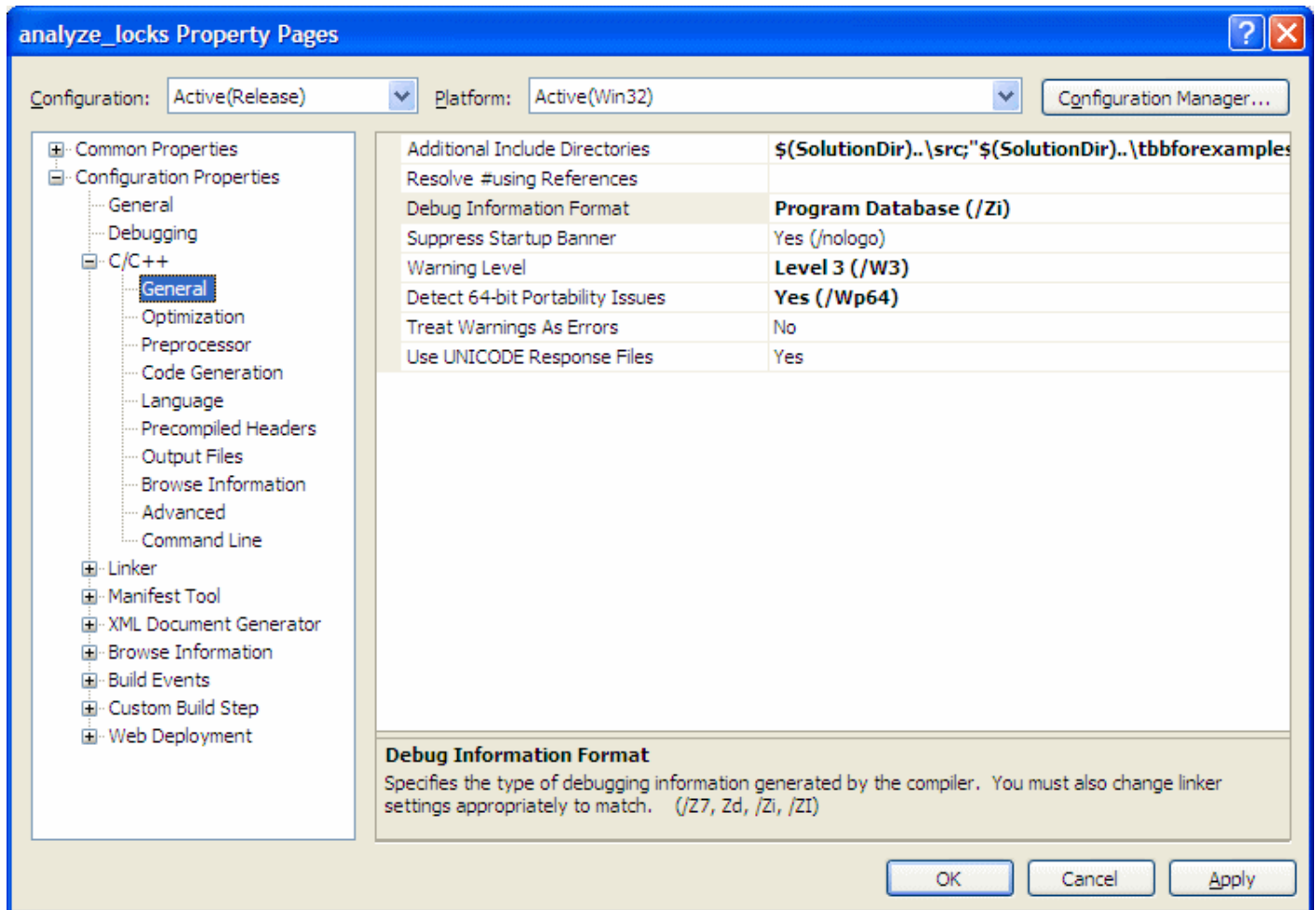
6. For Microsoft\* Visual Studio\* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.



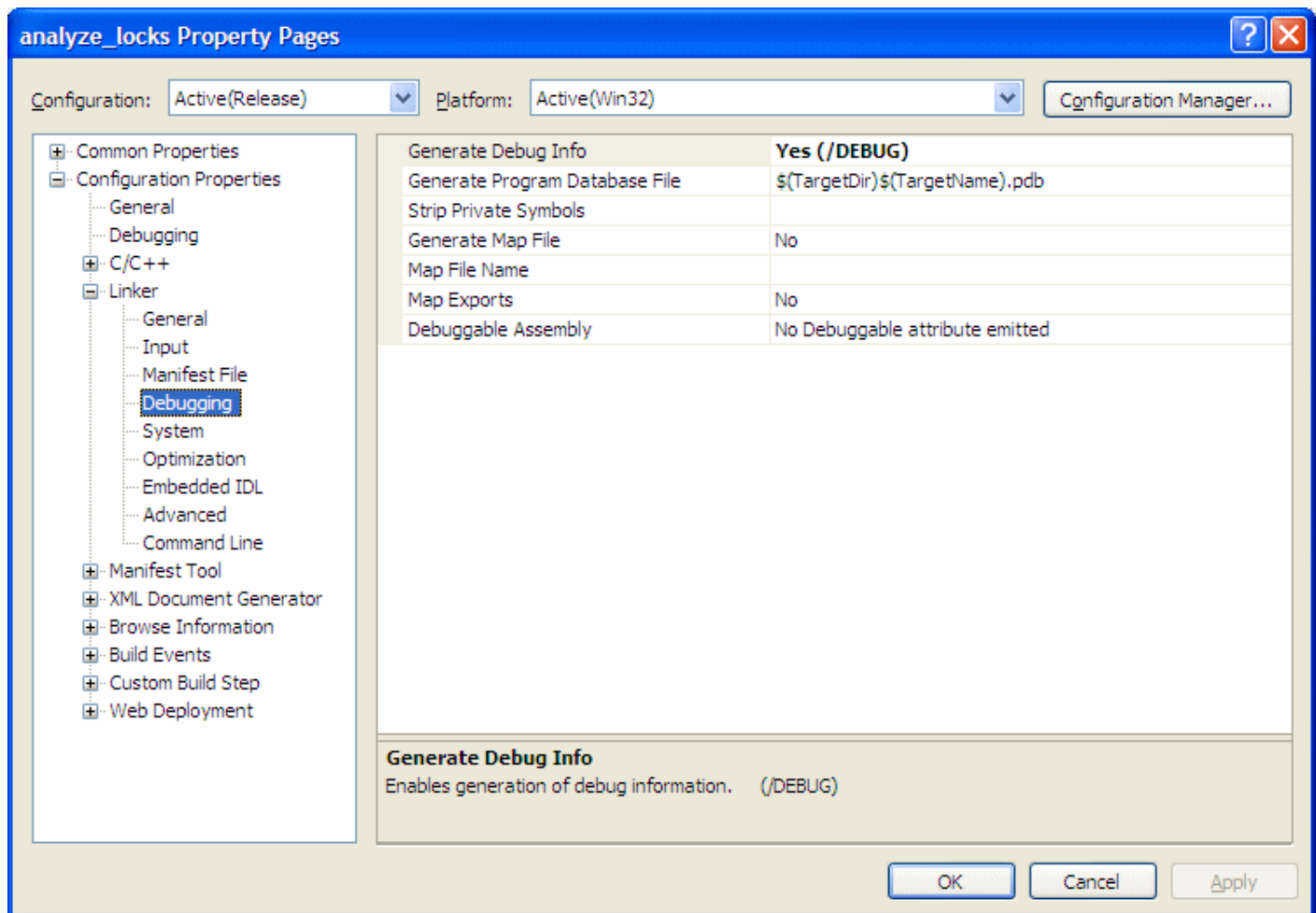
7. Click **Ok**.

### Enable Generating Debug Information for Your Binary Files

1. Select the **analyze\_locks** project and go to **Project > Properties**.
2. From the **analyze\_locks Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **analyze\_locks Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.



- From the **analyze\_locks Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



### Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build analyze\_locks**.

The `tachyon_analyze_locks` application is built.



**NOTE** The build configuration for `tachyon` may initially be set to Debug, which is typically used for development. When analyzing performance issues with the VTune Amplifier XE, you are recommended to use the Release build with normal optimizations. In this way, the VTune Amplifier XE is able to analyze the realistic performance of your application.

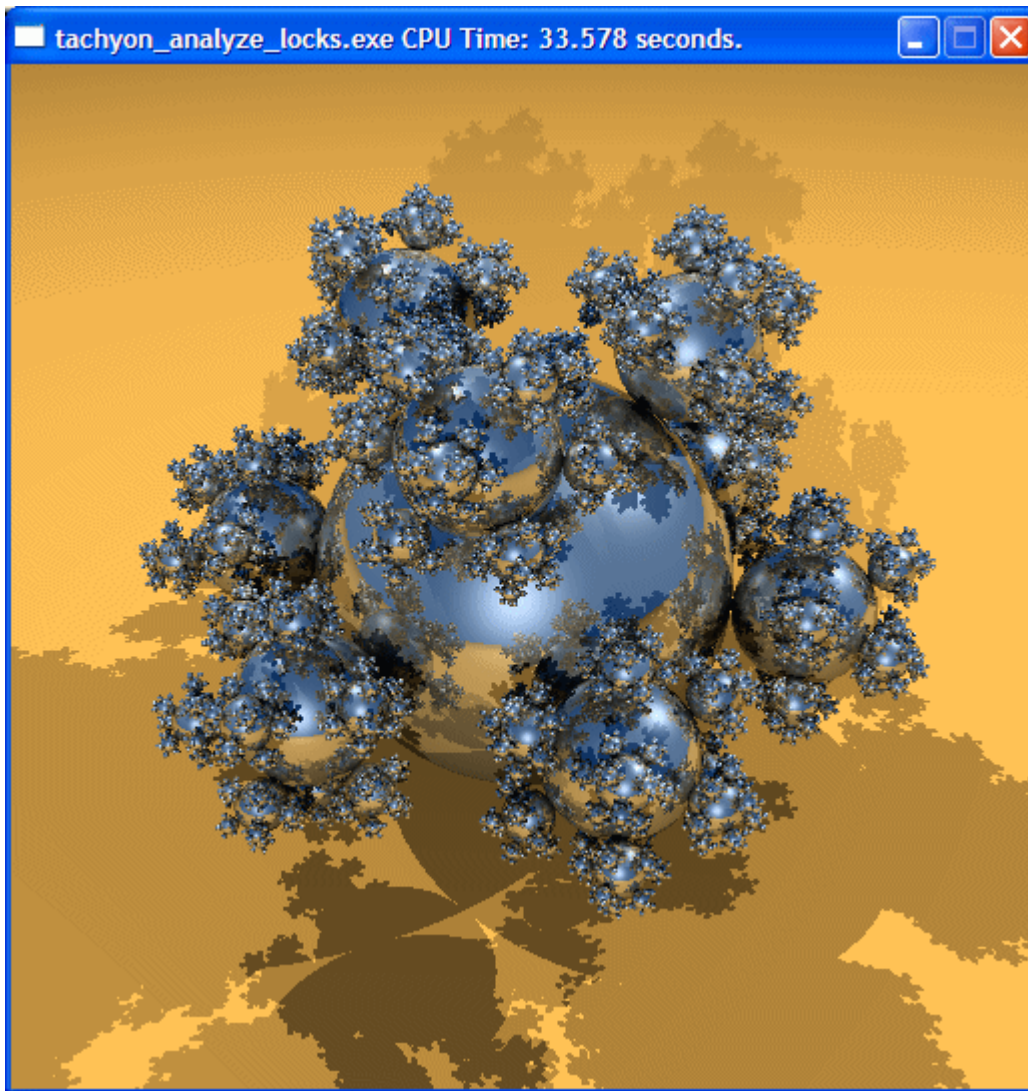
### Create a Performance Baseline

1. From the Visual Studio menu, select **Debug > Start Without Debugging**.

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. For the `tachyon_analyze_locks` executable in the figure above, the execution time is 33.578 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 selected the **analyze\_locks** project as the target for the Locks and Waits analysis.

## Key Terms and Concepts

- Term: `target`

## Next Step

Run Locks and Waits Analysis

## Standalone GUI: Build Application and Create New Project

---



Before you start analyzing your application for locks and waits, do the following:

**1. Build application.**

If you build the code in Visual Studio\*, make sure to:

- [Configure the Microsoft Visual Studio\\* environment to download the debug information for system libraries](#) so that VTune Amplifier XE can properly identify system functions and classify and attribute functions.
- [Configure Visual Studio project properties to generate the debug information for your application](#) so that VTune Amplifier XE can open the source code.
- [Build the target in the release mode with full optimizations](#), which is recommended for performance analysis.

**2. Run the application without debugging to create a performance baseline.**

**3. Create a VTune Amplifier XE project.**



---

**NOTE** The steps below are provided for Microsoft Visual Studio\* 2005. Steps for other versions of Visual Studio IDE may differ slightly.

---


### Enable Downloading the Debug Information for System Libraries

**1. Go to Tools > Options....**

The **Options** dialog box opens.

**2. From the left pane, select Debugging > Symbols.**

**3.**

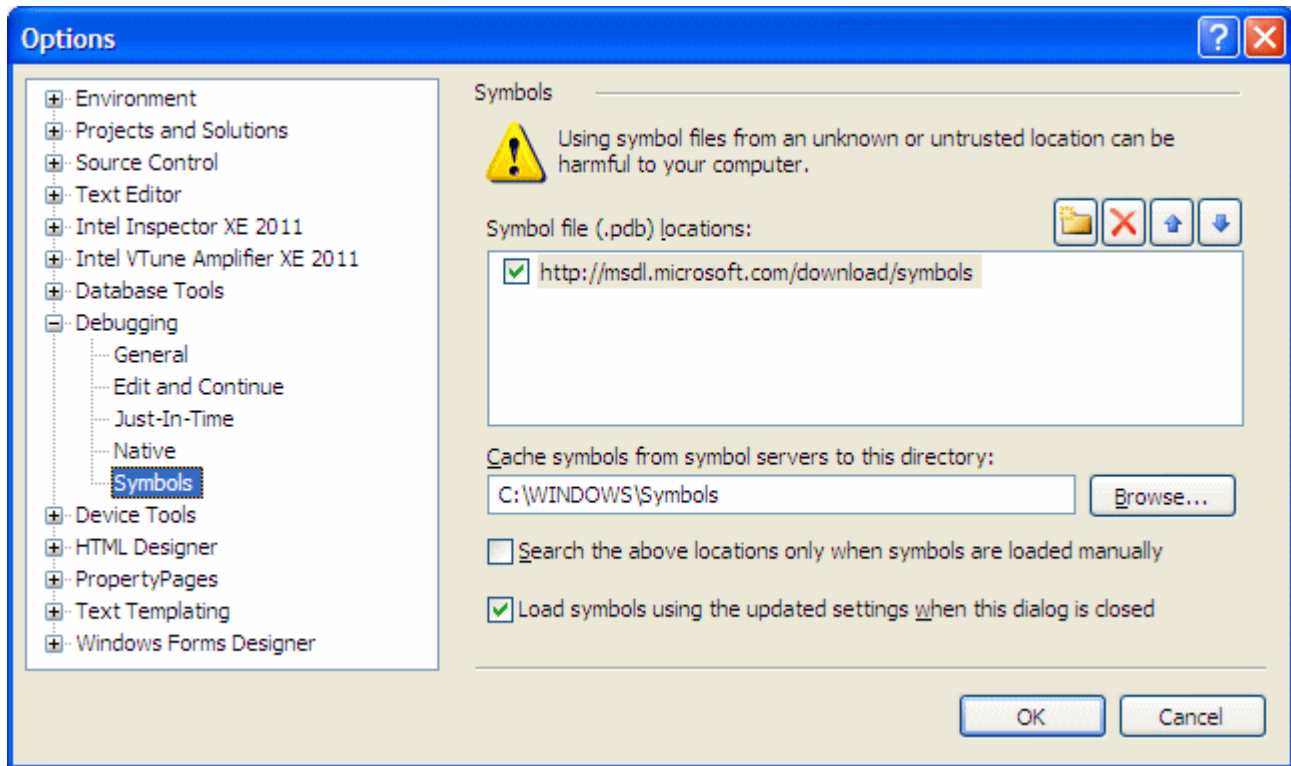
In the **Symbol file (.pdb) locations** field, click the  button and specify the following address: `http://msdl.microsoft.com/download/symbols`.

**4. Make sure the added address is checked.**

**5. In the Cache symbols from symbol servers to this directory field, specify a directory where the downloaded symbol files will be stored.**

**6. For Microsoft\* Visual Studio\* 2005, check the Load symbols using the updated settings when this dialog is closed box.**

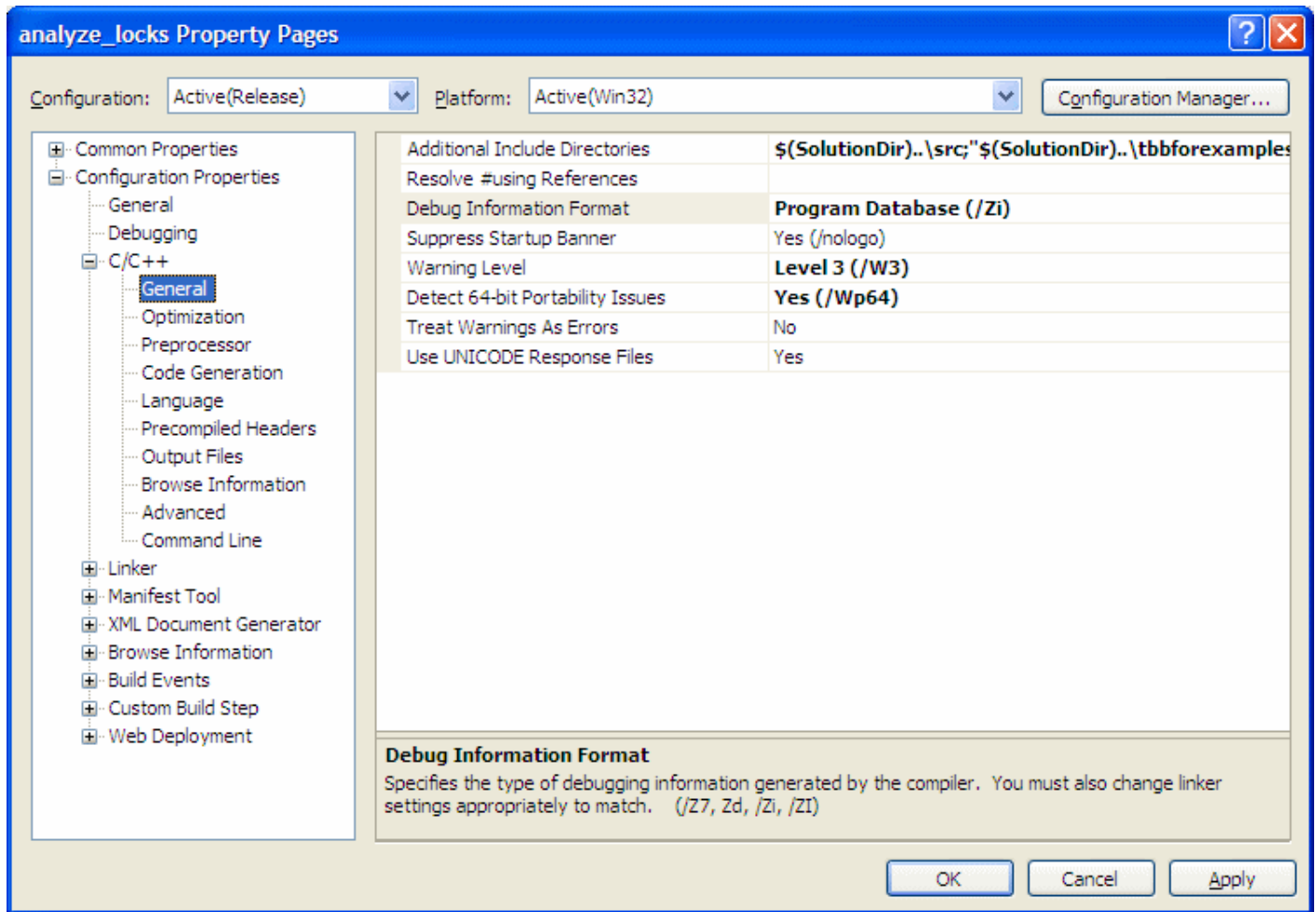




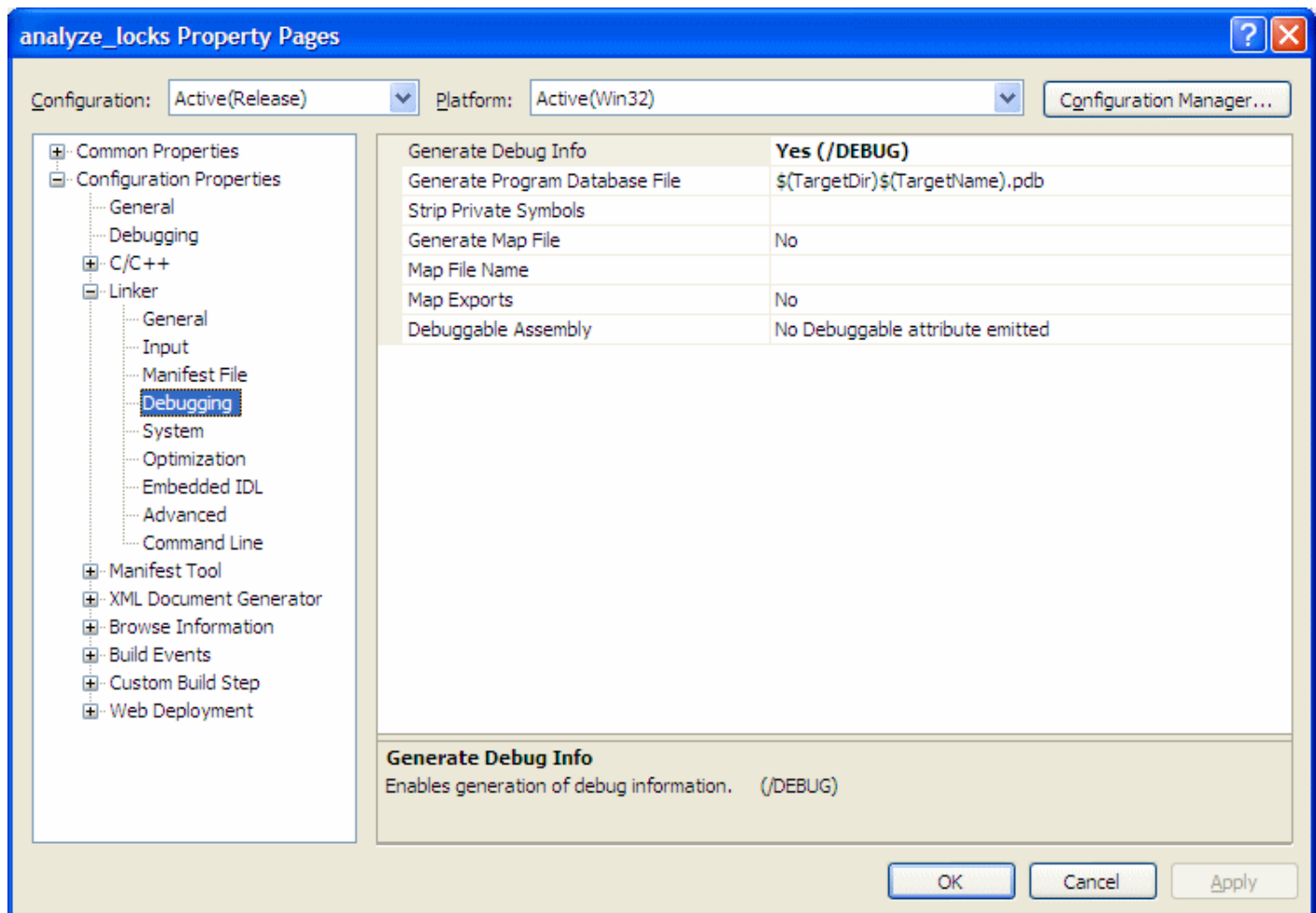
7. Click **Ok**.

### Enable Generating Debug Information for Your Binary Files

1. Select the **analyze\_locks** project and go to **Project > Properties**.
2. From the **analyze\_locks Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **analyze\_locks Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.



- From the **analyze\_locks Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



### Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build analyze\_locks**.

The `tachyon_analyze_locks` application is built.



**NOTE** The build configuration for `tachyon` may initially be set to Debug, which is typically used for development. When analyzing performance issues with the VTune Amplifier XE, you are recommended to use the Release build with normal optimizations. In this way, the VTune Amplifier XE is able to analyze the realistic performance of your application.

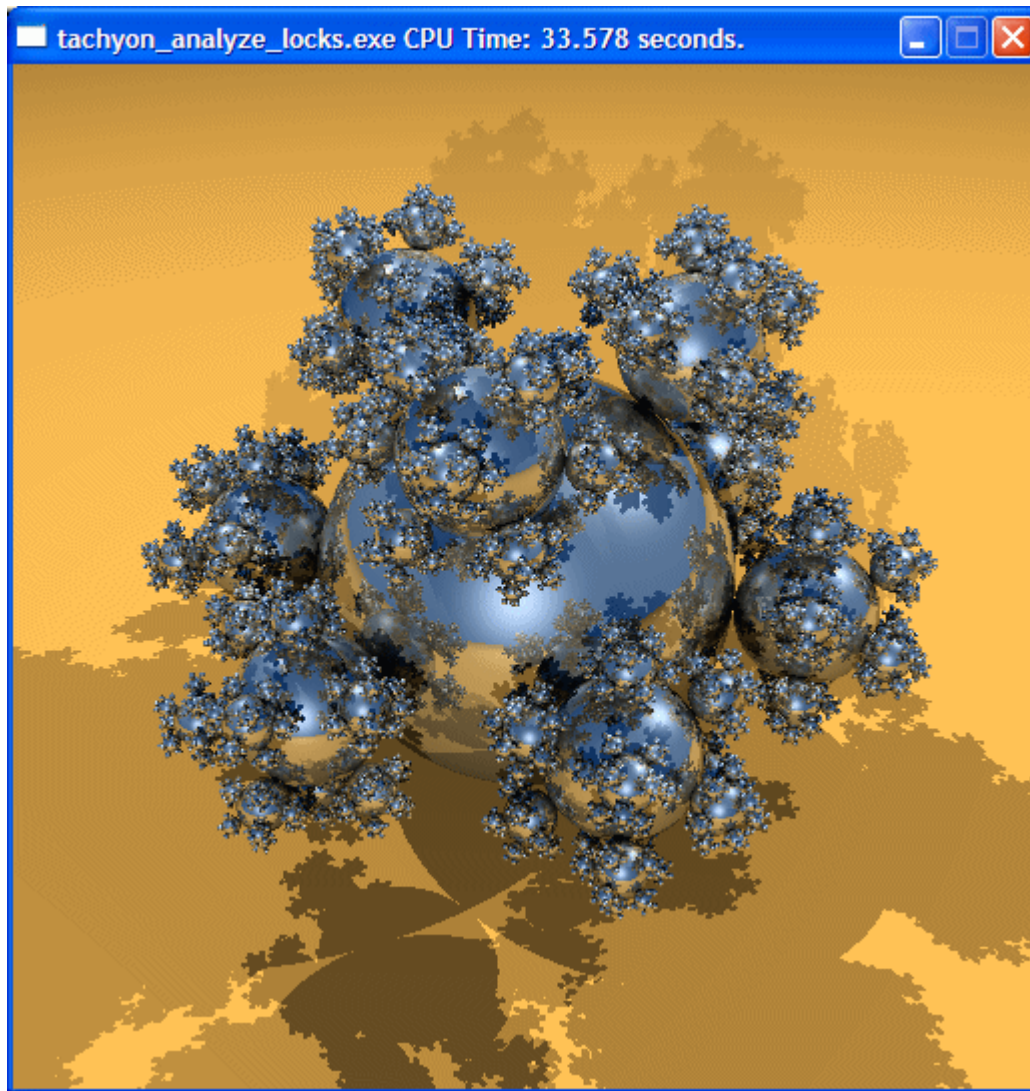
### Create a Performance Baseline

1. From the Visual Studio menu, select **Debug > Start Without Debugging**.

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. For the `tachyon_analyze_locks` executable in the figure above, the execution time is 33.578 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. From the **Start** menu select **Intel Parallel Studio XE 2011 > Intel VTune Amplifier XE 2011** to launch the VTune Amplifier XE standalone GUI.
2. Create a new project via **File > New > Project...**

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

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

VTune Amplifier XE creates a project directory under the `%USERPROFILE%\My Documents\My Amplifier XE Projects` directory and opens the **Project Properties: Target** dialog box.

4. 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>\analyze_locks.exe`.

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

## Recap

You set up your environment to enable generating symbol information for system libraries and your binary files, 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 and Concepts

- Term: [target](#), [baseline](#)
- Concept: [Locks and Waits Analysis](#)

## Next Step


[Run Locks and Waits Analysis](#)

# Run Locks and Waits Analysis



Before running an analysis, choose a configuration level to define the Intel® VTune™ 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:

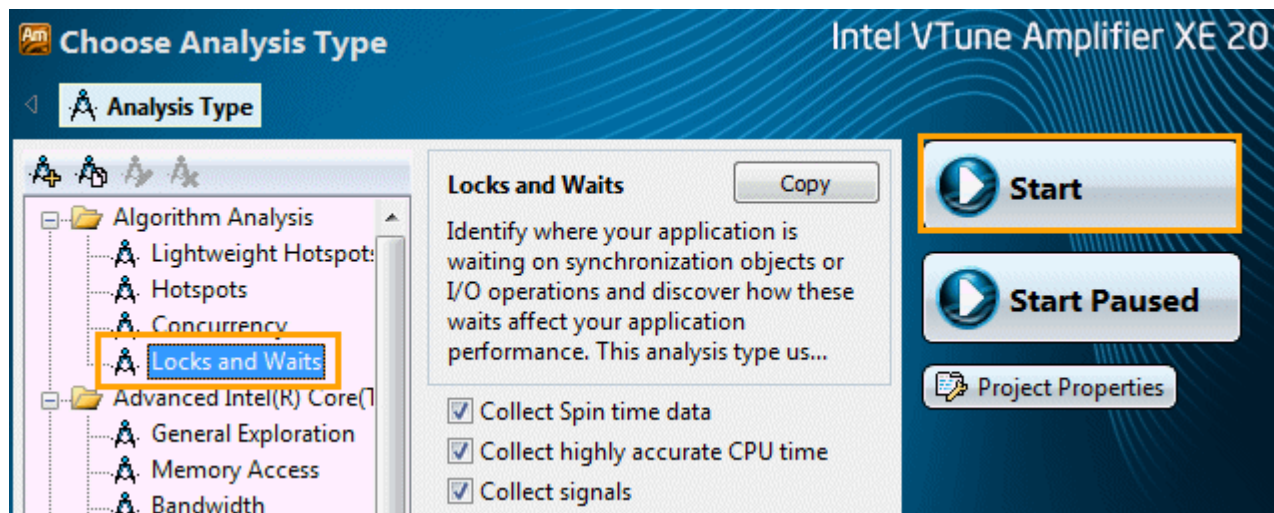
1. From the VTune Amplifier XE toolbar, analysis type from the drop-down menu click the  **New Analysis** button.

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.



**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](#)

## Next Step

[Interpret Result Data](#)

## Interpret Result Data



When the sample application exits, the Intel® VTune™ 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 the basic performance metrics](#) provided by the Locks and Waits analysis.
- [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:** ⓘ **49.170s** ⓘ

Wait Time: ⓘ	139.527s
Wait Count: ⓘ	3,298
CPU Time: ⓘ	37.006s
Total Thread Count:	4
Spin Time: ⓘ	3.169s

- 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.

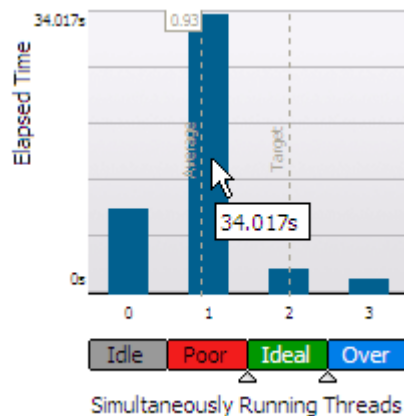
### Top Waiting Objects

This section lists the objects that spent the most time waiting synchronizations. A significant amount of Wait time associate

Sync Object	Wait Time <sup>Ⓢ</sup>	Wait Count <sup>Ⓢ</sup>
Auto Reset Event 0xc6df4909	40.674s	1,321
Multiple Objects	37.767s	279
Critical Section 0x6b3f2e9f	29.455s	438
TBB Scheduler	13.229s	0
Sleep	10.186s	977
[Others]	8.216s	283

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.



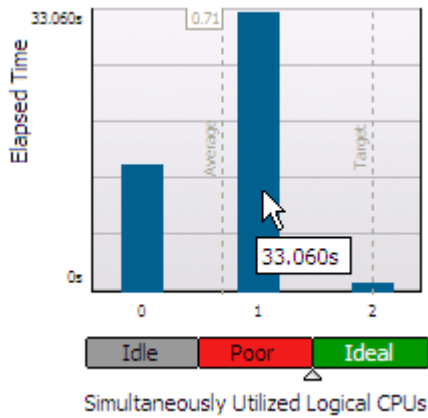
Note the **Target** value. By default, this number is equal to the number of physical cores. Consider this number as your optimization goal.

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 four threads on a machine with four cores. But it is not using available cores effectively. The Average CPU Usage on the chart is about 0.7 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 15 seconds, which is classified as Poor concurrency.

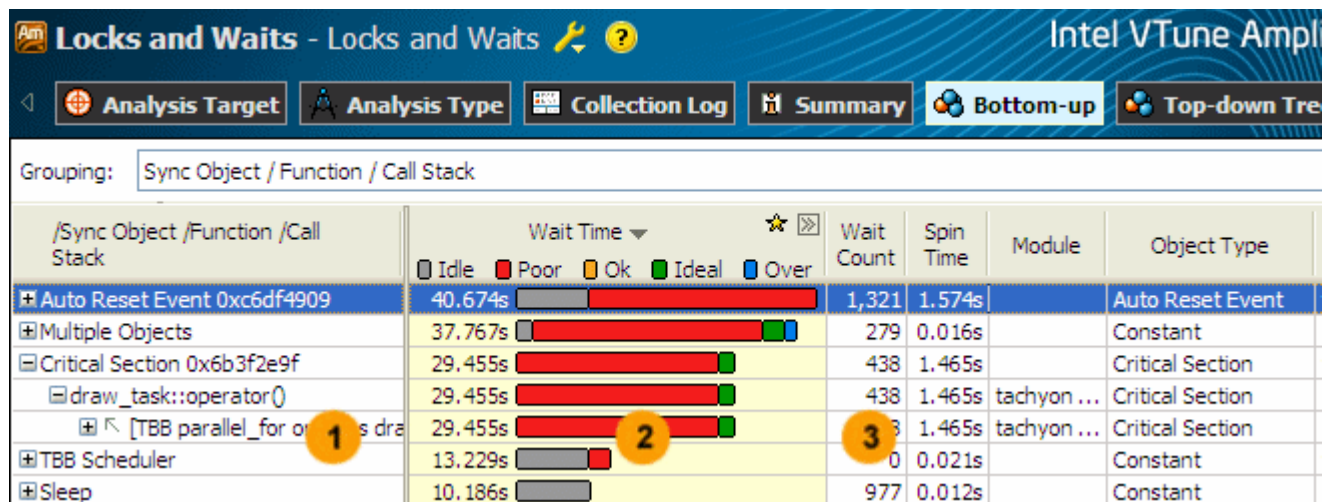
The **CPU Usage Histogram** represents the Elapsed time and usage level for the logical CPUs. Ideally, the highest bar of your chart should be within the Ok or Ideal utilization range.



The `tachyon_analyze_locks` application ran mostly on one logical CPU. If you hover over the second bar, you see that it spent 16.603 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.



**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.

For Intel® Threading Building Blocks (Intel® 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.


**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.

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.

From the code knowledge, you may understand that the Manual and Auto Reset Event objects are most likely related to the join where the main program is waiting for the worker threads to finish. This should not be a problem. Consider the third item in the Bottom-up pane that is more interesting. It is a Critical Section that shows much serial time and is causing a wait. Click the plus sign  at the object name to expand the node and see the `draw_task` wait function that contains this critical section and call stack. Double-click the Critical Section 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](#), [Wait time](#)
- Concept: [Locks and Waits Analysis](#), [CPU Usage](#), [Data of Interest](#)

## Next Step

[Analyze Code](#)


# Analyze Code



You identified the critical section 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® VTune™ 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

Line	Source	Wait Time			Wait Count	Spin Time
		Idle	Poor	Ok		
168						
169	// Enter Critical Section to protect pixel calculation from multith					
170	EnterCriticalSection(&rgb_critical_section);	29.455s			438	1.465s
171						
172	for (int x = startx; x < stopx; x++) {					
173	color_t c = render_one_pixel (x, y, local_mbox, serial, startx,					
174	drawing.put_pixel(c);					
175	}					
176						
177	// Exit from the critical section					
178	LeaveCriticalSection(&rgb_critical_section);					
179						


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.
- 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 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 170 entering the critical section `rgb_critical_section` 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 438 times.

The `rgb_critical_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.

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 Usage](#), [Locks and Waits Analysis](#), [Data of Interest](#)

### Next Step

[Remove Lock](#)

---


## Remove Lock



In the Source window, you located the critical section 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:

```


draw_task operator (const tbb::blocked_range<int> &r) cor
161 unsigned int serial = 1;
162 unsigned int mboxsize = sizeof(unsigned int)*(max_objectid() + 2
163 unsigned int * local_mbox = (unsigned int *) alloca(mboxsize);
164 memset(local_mbox,0,mboxsize);
165
166 for (int y=r.begin(); y!=r.end(); ++y) {
167     drawing_area drawing(startx, totaly-y, stopx-startx, 1);
168
169     // Enter Critical Section to protect pixel calculation from
170     EnterCriticalSection(&rgb_critical_section);
171
172     for (int x = startx; x < stopx; x++) {
173         color_t c = render_one_pixel (x, y, local_mbox, serial,
174         drawing.put_pixel(c);
175     }
176
177     // Exit from the critical section
178     LeaveCriticalSection(&rgb_critical_section);
179
180     if(!video->next_frame()) tbb::task::self().cancel_group_exec
...

```

## Remove the Lock

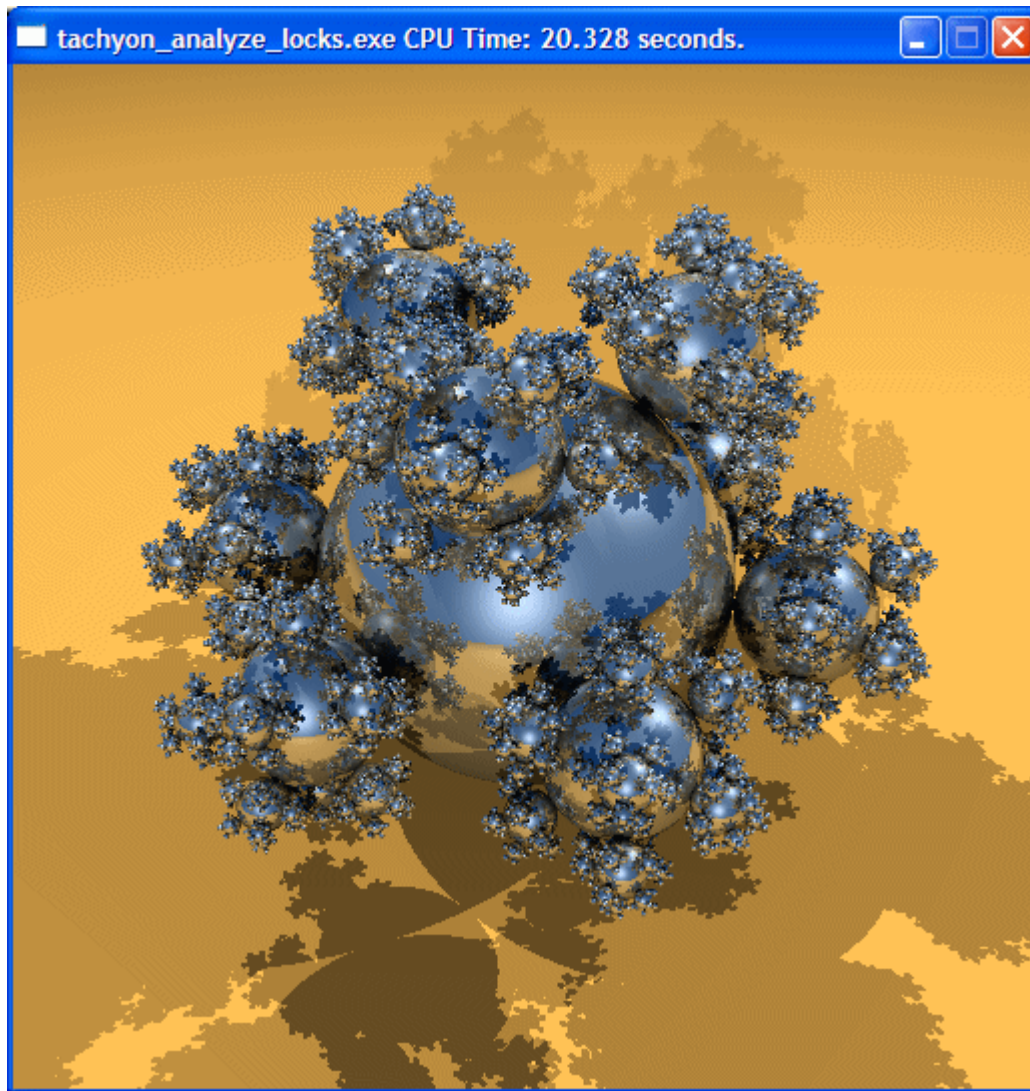
The `rgb_critical_section` was introduced to protect calculation from multithreaded access. The brief analysis shows that the code is thread safe and the critical section is not really needed.

To resolve this issue:

 **NOTE** The steps below are provided for Microsoft Visual Studio\* 2005. Steps for other versions of Visual Studio IDE or for the standalone version of the VTune Amplifier XE may slightly differ.

1. Comment out code lines 170 and 178 to disable the critical section.
2. From Solution Explorer, select the **analyze\_locks** project.
3. From Visual Studio menu, select **Build > Rebuild analyze\_locks**.  
The project is rebuilt.
4. From Visual Studio menu, select **Debug > Start Without Debugging** to run the application.

Visual Studio runs the `tachyon_analyze_locks.exe`. Note that execution time reduced from 33.578 seconds to 20.328 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](#)

## Next Step

[Compare with Previous Result](#)


## Compare with Previous Result



You made sure that removing the critical section gave you 13 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:

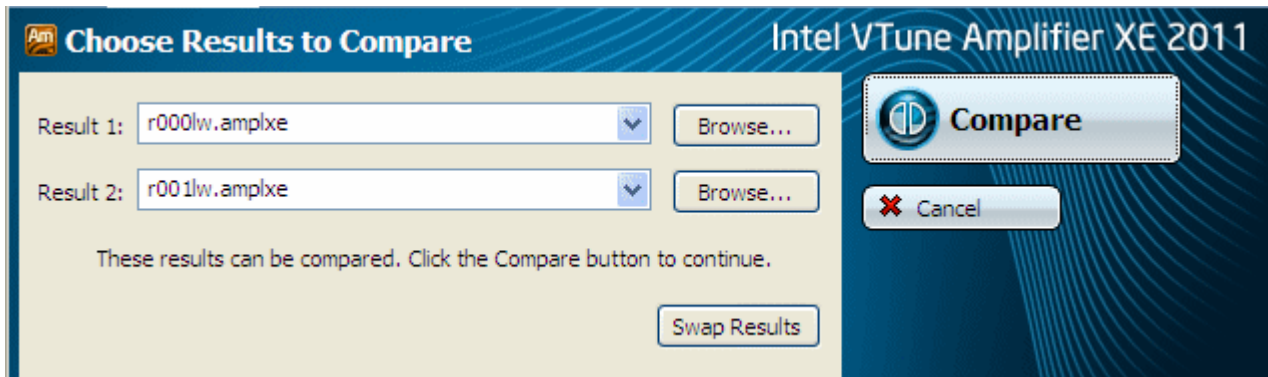
- Compare results before and after optimization.
- Identify the performance gain.

## Compare Results Before and After Optimization

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

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:

Sync Object /Function	Wait Time:Difference				Wait Time: r000lw			Wait Time: r001lw			Wait Count:Diff ...	Wait Count: r000lw	Wait Count: r001lw
	Idle	Poor	Ideal	Over	Idle	Poor	Ok	Idle	Poor	Ok			
Critical Section 0x6b3f2e9f	0s	27.264s	2.176s	0.015s	29.455s			0s			438	438	0
Auto Reset Event 0xc6df4909	0.064s	30.283s	-7.160s	-0.000s	40.674s			17.487s			245	1,321	1,076
Multiple Objects	-0.963s	30.383s	-5.341s	-11.129s	37.767s			24.817s			100	279	179
TBB Scheduler	0.074s	3.007s	0.078s	-0.000s	13.229s			10.071s			-1	0	1
Manual Reset Event 0x6d8bec	1.051s	0.007s	-0.037s	0s	7.994s			6.972s			0	2	2
Stream ..\dat\balls.dat 0xd79	0.113s	0s	0s	0s	0.113s			0.001s			-2	3	5
Sleep	0.057s	0.006s	0.003s	0s	10.186s			10.120s			-14	977	991
Unknown 0x0524f060	0.029s	-0.000s	0s	0s	0.036s			0.008s			10	71	61
Thread 0x0e51a306	0.014s	0.001s	0s	0s	0.028s			0.013s			0	2	2

- 1 Difference in Wait time per utilization level between the two results in the following format:  $\langle \text{Difference Wait Time} \rangle = \langle \text{Result 1 Wait Time} \rangle - \langle \text{Result 2 Wait Time} \rangle$ . 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.
- 2 Wait time and CPU utilization for the initial version of the code.
- 3 Wait time and CPU utilization for the optimized version of the code.
- 4 Difference in Wait count between the two results in the following format:  $\langle \text{Difference Wait Count} \rangle = \langle \text{Results 1 Wait Count} \rangle - \langle \text{Result 2 Wait Count} \rangle$ .



- 5 Wait count for the initial version of the code.
- 6 Wait count for the optimized 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.

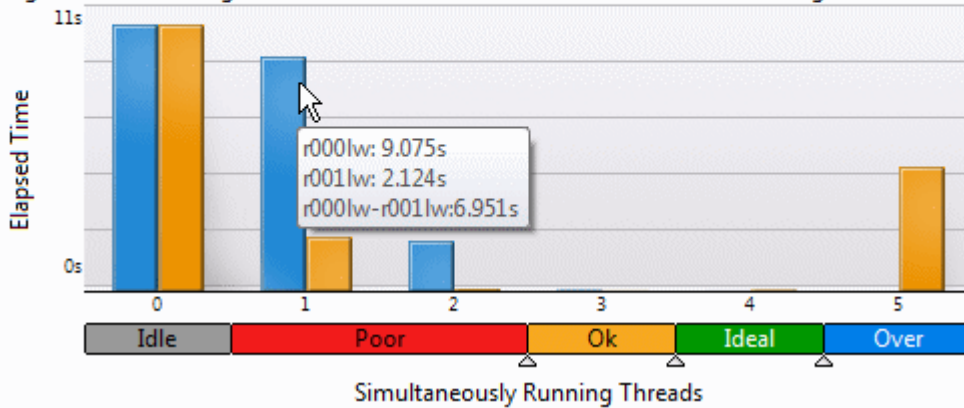
**Elapsed Time:** 21.395s - 17.373s = 4.022s

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 v higher than CPU usage if threads are in the runnable state and not consuming CPU time.



In the Bottom-up pane, locate the Critical Section you identified as a bottleneck in your code. Since you removed it during optimization, the optimized result r001lw 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 29 seconds of optimization in Wait time.

Sync Object /Function	Wait Time:Difference	Wait Time: r000lw			Wait Time: r001lw		
		Idle	Poor	Ok	Idle	Poor	Ok
⊕ Critical Section 0x6b3f2e9f	29.455s						0s
⊕ Auto Reset Event 0xc6df4909	23.188s						
⊕ Multiple Objects	12.950s						
⊕ TBB Scheduler	3.158s						
⊕ Manual Reset Event 0x6d8bec	1.022s						
⊕ Stream ..\dat\balls.dat 0xd79	0.113s						0.001s
⊕ Sleep	0.066s						10.120s

## 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 (r001lw 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-c1` 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 Usage](#)

## Next Step

Read [Summary](#)

# Summary

---



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

## Step 1. Choose and Build Your Target

- Configure the Microsoft\* symbol server and your project properties to get the most accurate results for system and 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. For Visual Studio\* projects, the analysis target settings are inherited automatically.



## 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-c1` 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 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.

## 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.



# Tutorial: Identifying Hardware Issues

---

# 3



## Learning Objectives

---



This tutorial shows how to use the General Exploration analysis of the Intel® VTune™ Amplifier XE to identify the hardware-related issues in the sample application.

Estimated completion time: 15 minutes.

Sample application: `matrix`.

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

- Choose an analysis target.
- Run the General Exploration analysis for Intel® microarchitecture code name Nehalem.
- Understand the event-based performance metrics.
- Identify the types of the most critical hardware issues for the application as a whole.
- Identify the modules/functions that caused the most critical hardware issues.
- Analyze the source code to locate the most critical code lines.
- Identify the next steps of the performance analysis to get more detailed results.

### Start Here

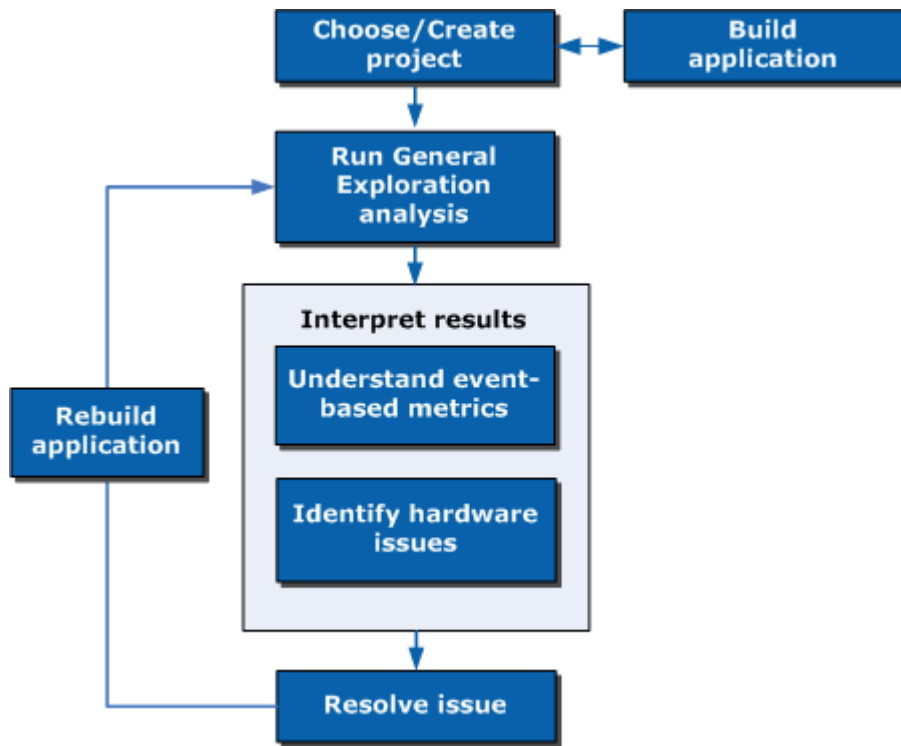
[Workflow Steps to Identify Hardware Issues](#)

## Workflow Steps to Identify Hardware Issues

---



You can use an advanced event-based sampling analysis of the Intel® VTune™ Amplifier XE to identify the most significant hardware issues that affect the performance of your application. This tutorial guides you through these workflow steps running the General Exploration analysis type on a sample `matrix` application.



1. Do one of the following:
  - Visual Studio\* IDE: Choose a project, verify settings, and build application.
  - Standalone GUI: Build an application to analyze for hardware issues and create a new VTune Amplifier XE project.
2. Choose and run the General Exploration analysis.
3. Interpret the result data.
4. View and analyze code of the performance-critical functions.
5. Modify the code to resolve the detected performance issues and rebuild the code.

## Visual Studio\* IDE: Choose Project and Build Application



Before you start analyzing hardware issues affecting the performance of your application, do the following:

1. Choose a project with the analysis target in the Visual Studio IDE.
2. Configure the Microsoft Visual Studio\* environment to download the debug information for system libraries so that the VTune Amplifier XE can properly identify system functions and classify and attribute functions.
3. Configure Visual Studio project properties to generate the debug information for your application so that the VTune Amplifier XE can open the source code.
4. Build the target in the release mode with full optimizations, which is recommended for performance analysis.

For this tutorial, your target is the `matrix` application that calculates matrix transformations. To learn how to install and set up the sample code, see [Prerequisites](#).



- The steps below are provided for Microsoft Visual Studio\* 2005. Steps for other versions of Visual Studio IDE or for the standalone version of the Intel® VTune™ Amplifier XE 2011 may slightly differ.

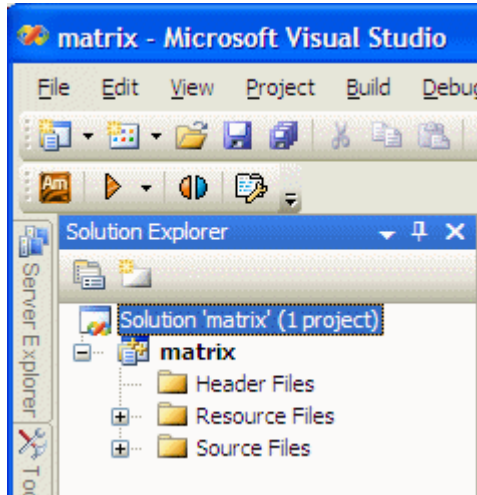
- Steps provided by this tutorial are generic and applicable to any application. You may choose to follow the proposed workflow using your own application.

## Choose a Project

1. From the Visual Studio menu, select **File > Open > Project/Solution....**

The **Open Project** dialog box opens.

2. In the **Open Project** dialog box, browse to the location where you extracted the `matrix_vtune_amp_xe.zip` file and select the `matrix.sln` file.



The solution is added to Visual Studio and shows up in the Solution Explorer. VTune Amplifier XE automatically inherits Visual Studio settings and uses the currently opened project as a target project for performance analysis.

When you choose a project in Visual Studio IDE, the VTune Amplifier XE automatically creates the `config.amplxproj` project file and sets the `matrix` application as an analysis target in the project properties.


## Enable Downloading the Debug Information for System Libraries

1. Go to **Tools > Options....**

The **Options** dialog box opens.

2. From the left pane, select **Debugging > Symbols.**

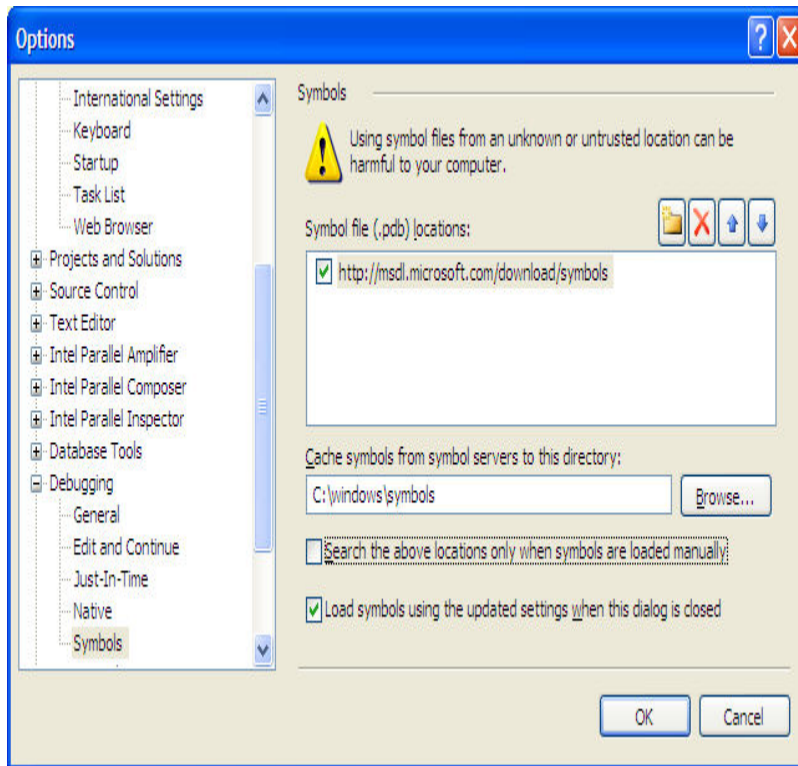
3.

In the **Symbol file (.pdb) locations** field, click the  button and specify the following address: <http://msdl.microsoft.com/download/symbols>.

4. Make sure the added address is checked.

5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.

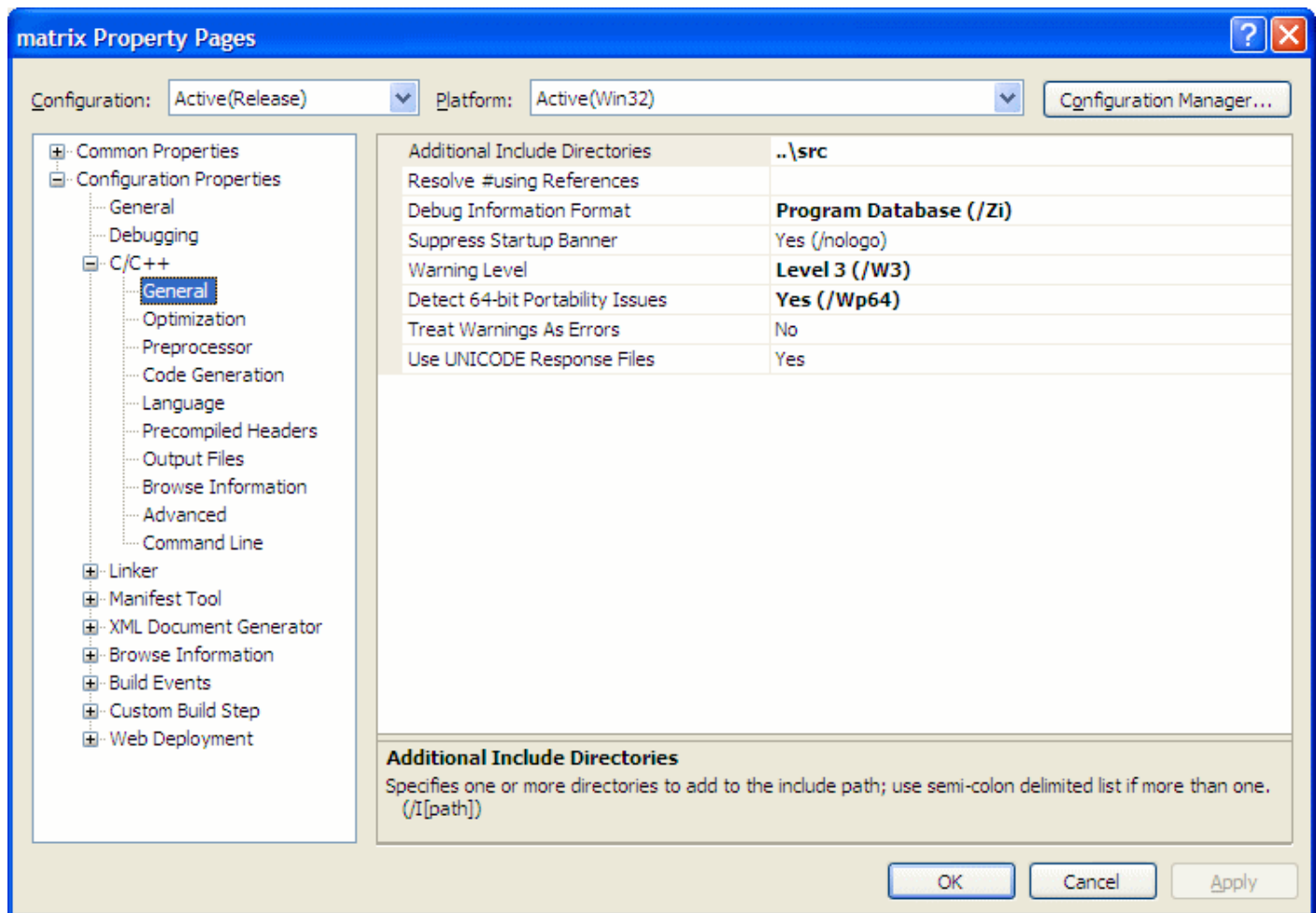
6. For Microsoft Visual Studio\* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.



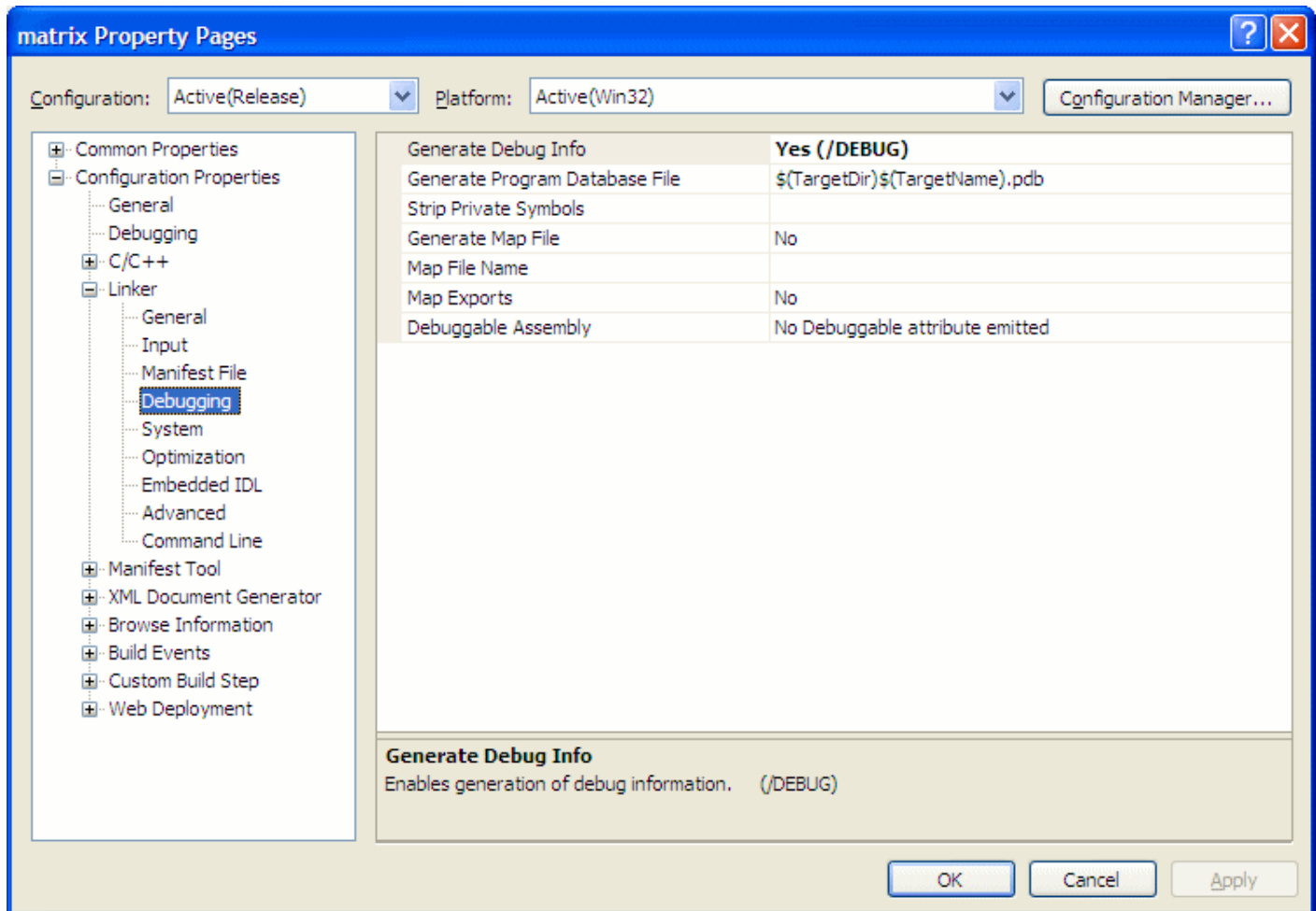
7. Click **Ok**.

## Enable Generating Debug Information for Your Binary Files

1. Select the **matrix** project and go to **Project > Properties**.
2. From the **matrix Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **matrix Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.



4. From the **matrix Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



### Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build matrix**.

The `matrix.exe` application is built.

### Recap

You selected the **matrix** project as the target for the hardware event-based sampling analysis, set up your environment to enable generating symbol information for system libraries and your binary files, and built the target in the Release mode. Your application is ready for analysis.

### Next Step

Run General Exploration Analysis

## Standalone GUI: Build Application and Create New Project



Before you start analyzing hardware issues affecting the performance of your application, do the following:

1. Build application.

If you build the code in Visual Studio\*, make sure to:

- Configure the Microsoft Visual Studio\* environment to download the debug information for system libraries so that the VTune Amplifier XE can properly identify system functions and classify and attribute functions.
- Configure Visual Studio project properties to generate the debug information for your application so that the VTune Amplifier XE can open the source code.
- Build the target in the release mode with full optimizations, which is recommended for performance analysis.

## 2. Create a VTune Amplifier XE project.



**NOTE** The steps below are provided for Microsoft Visual Studio\* 2005. Steps for other versions of Visual Studio IDE or for the standalone version of the Intel® VTune™ Amplifier XE may differ slightly.


## Enable Downloading the Debug Information for System Libraries

### 1. Go to **Tools > Options...**

The **Options** dialog box opens.

### 2. From the left pane, select **Debugging > Symbols**.

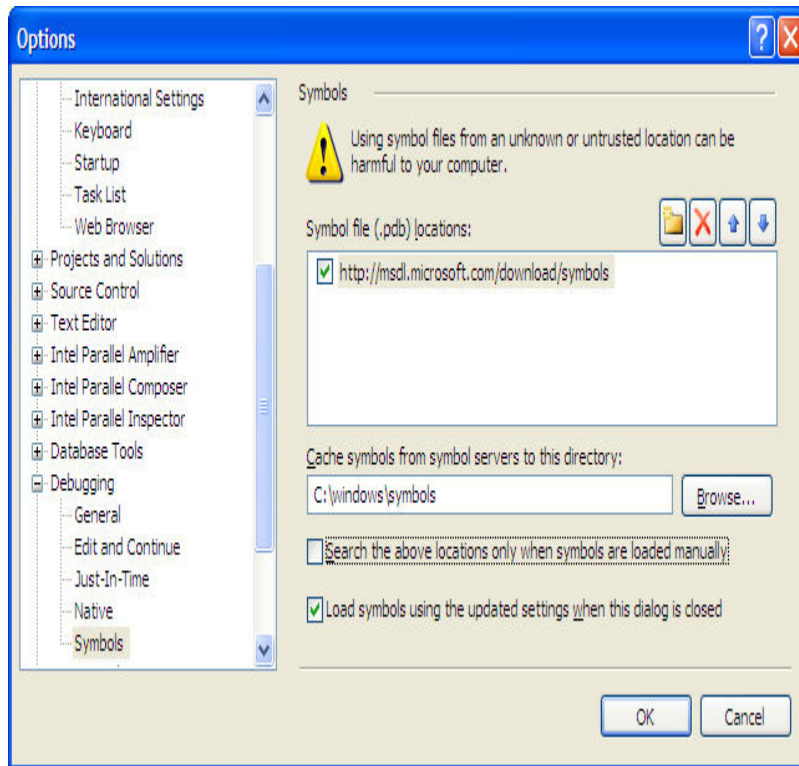
### 3.

In the **Symbol file (.pdb) locations** field, click the  button and specify the following address: <http://msdl.microsoft.com/download/symbols>.

### 4. Make sure the added address is checked.

### 5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.

### 6. For Microsoft Visual Studio\* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.

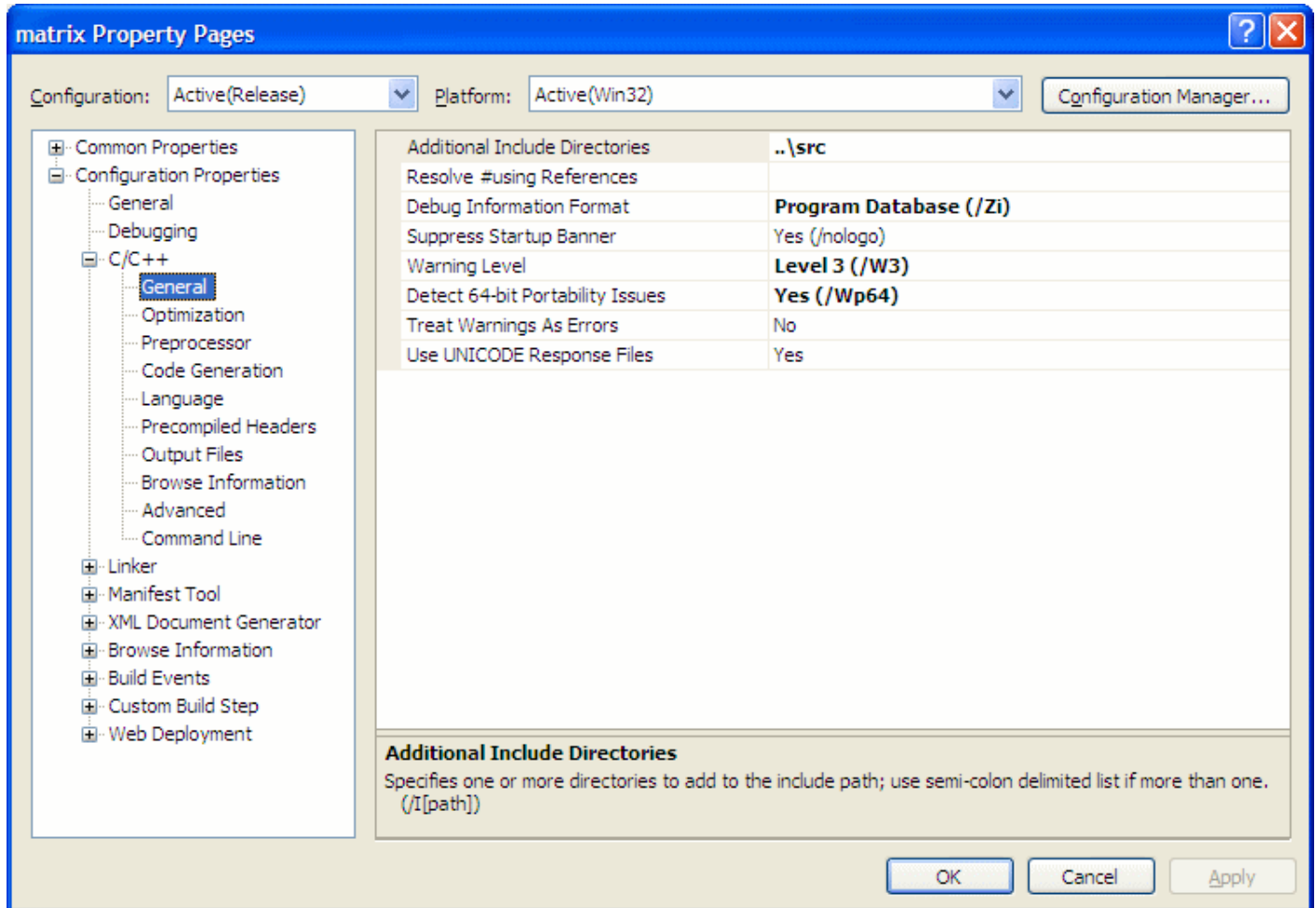


### 7. Click **Ok**.

## Enable Generating Debug Information for Your Binary Files

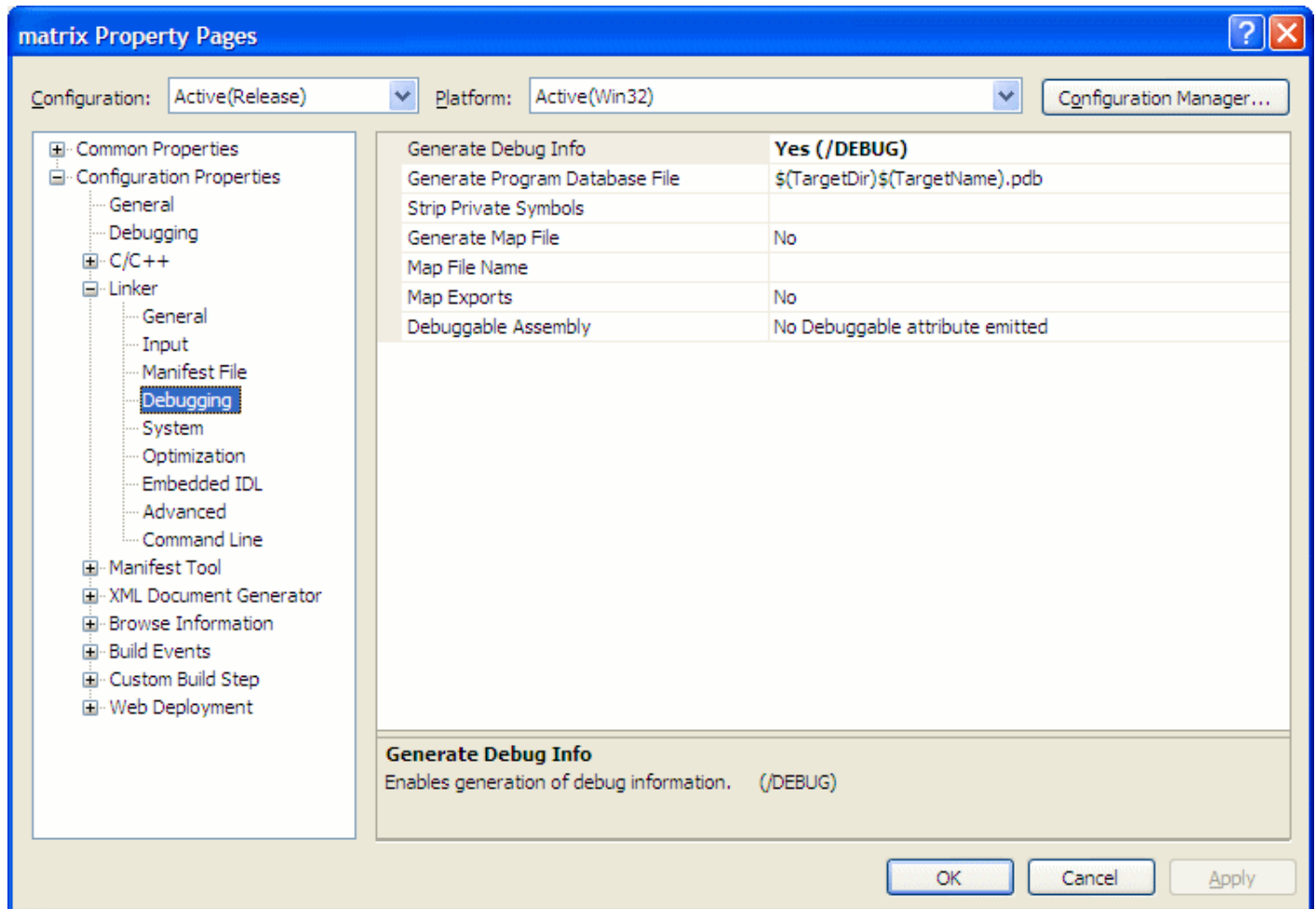
### 1. Select the **matrix** project and go to **Project > Properties**.

2. From the **matrix Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **matrix Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.



4. From the **matrix Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.





### Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build matrix**.

The `matrix.exe` application is built.

### Create a Project

1. From the **Start** menu select **Intel Parallel Studio XE 2011 > Intel VTune Amplifier XE 2011** to launch the VTune Amplifier XE GUI client.
2. Create a new project via **File > New > Project...**

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

3. Specify the project name `matrix` that will be used as the project directory name and click the **Create Project** button.

By default, the VTune Amplifier XE creates a project directory under the `%USERPROFILE%\My Documents \My Amplifier XE Projects` directory and opens the **Project Properties: Target** dialog box.

4. In the **Target: Application to Launch** pane, browse to the `matrix.exe` application and click **OK**.

### Recap

You set up your environment to enable generating symbol information for system libraries and your binary files, built the target in the Release mode, and created the VTune Amplifier XE project for your analysis target. Your application is ready for analysis.

## Key Terms and Concepts

- Term: **target**
- Concept: **Event-based Sampling Analysis**

## Next Step

Run General Exploration Analysis

# Run General Exploration Analysis




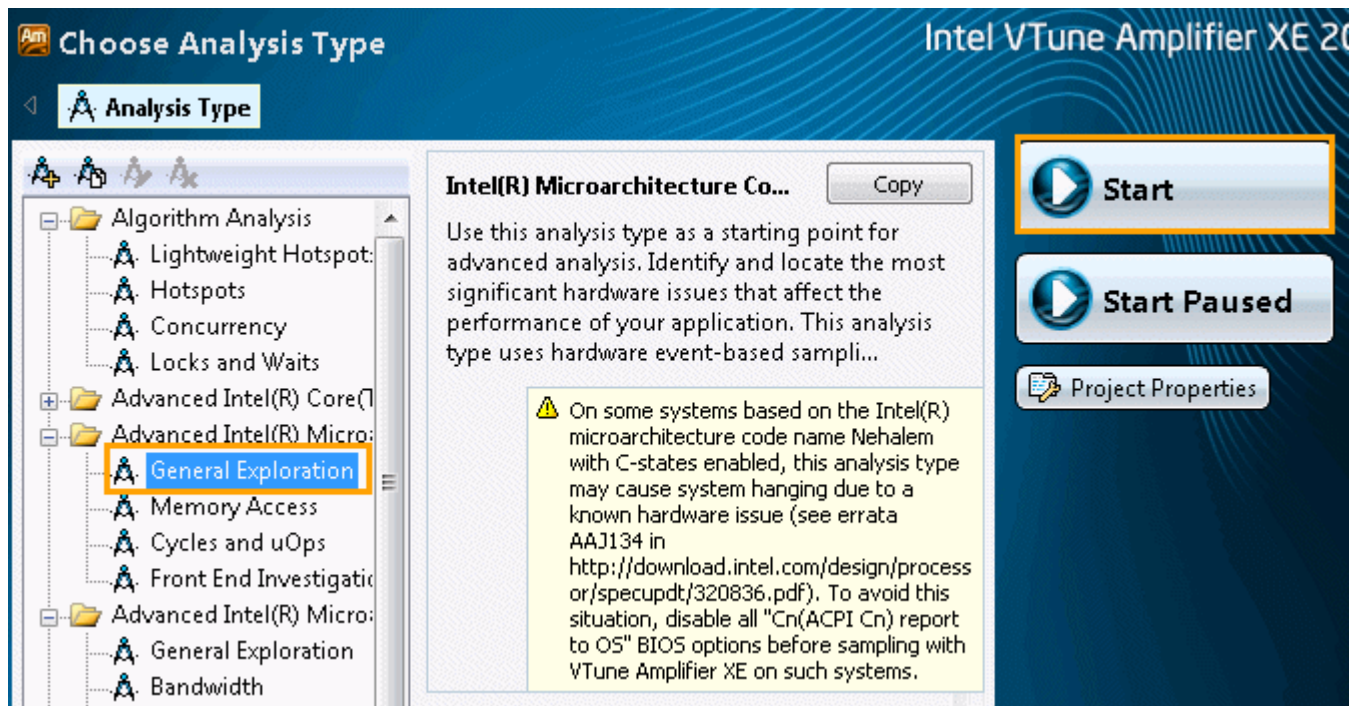
After building the target, you can run it with the Intel® VTune™ Amplifier XE to analyze its performance. In this tutorial, you run the General Exploration analysis on the Intel® Core™ i7 processor based on the Intel® microarchitecture code name Nehalem. The General Exploration analysis type helps identify the widest scope of hardware issues that affect the application performance. This analysis type is based on the hardware event-based sampling collection.



**NOTE** The steps below are provided for Microsoft Visual Studio\* 2005. Steps for other versions of Visual Studio IDE or for the standalone version of the VTune Amplifier XE may slightly differ.

### To run the analysis:

1. From the VTune Amplifier XE toolbar, click the  **New Analysis** button.  
The **New Amplifier XE Result** tab opens with the **Analysis Type** configuration window active.
2. From the analysis tree on the left, select the **Advanced Intel(R) Microarchitecture Code Name Nehalem Analysis > General Exploration** analysis type.
3. Click the **Start** button on the right to run the analysis.



The VTune Amplifier XE launches the `matrix` application that calculates matrix transformations and exits. The VTune Amplifier XE finalizes the collected data and opens the results in the Hardware Issues 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 General Exploration analysis that monitors how your application performs against a set of event-based hardware metrics. To see the list of processor events used for this analysis type, see the **Details** section of the General Exploration configuration pane.

## Key Terms and Concepts

- Term: [viewpoint](#)
- Concept: [Event-based Sampling Analysis](#), [Finalization](#)

## Next Step

[Interpret Results](#)

# Interpret Results

---



When the application exits, the Intel® VTune™ Amplifier XE finalizes the results and opens the Hardware Issues viewpoint that consists of the Summary window, Bottom-up window, and Timeline pane. To interpret the collected data and understand where you should focus your tuning efforts for the specific hardware, do the following:

- [Understand the event-based metrics](#)
- [Identify the hardware issues that affect the performance of your application](#)



**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.

---

## Understand the Event-based Metrics

Click the **Summary** tab to explore the data provided in the Summary window for the whole application performance.

Elapsed Time: 56.740s 1

INST_RETIRED.ANY:	43,616,000,000
CPU_CLK_UNHALTED.THREAD:	276,892,000,000
CPI Rate: ⓘ	6.348
The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or...	
Retire Stalls: ⓘ	0.912
A high number of retire stalls is detected. This may result from branch misprediction, instruction starvation, long latency...	
LLC Miss: ⓘ	3.292
A high number of CPU cycles spent waiting for LLC load misses to be serviced. Possible optimizations are to reduce data working...	
LLC Load Misses Serviced By Remote DRAM: ⓘ	1.020
A significant amount of time is spent servicing memory requests from remote DRAM. Wherever possible, try to consistently use...	
Contested Accesses: ⓘ	0.000
Instruction Starvation: ⓘ	-0.007
Branch Mispredict: ⓘ	0.000
Execution Stalls: ⓘ	0.755
The percentage of cycles with no micro-operations executed is high. Look for long-latency operations at code regions with high...	
Data Sharing: ⓘ	0.103
Significant data sharing by different cores is detected. First, examine the Contested Accesses metric to determine whether the..	

Significant data sharing by different cores is detected. First, examine the Contested Accesses metric to determine whether the major component of data sharing is due to contested accesses or simple read sharing. Read sharing is a lower priority than Contested Accesses or issues such as LLC Misses and Remote Accesses. If simple read sharing is a performance bottleneck, consider changing data layout across threads or rearranging computation. However, this type of tuning may not be straightforward and could bring more serious performance issues back.

- Elapsed time is the wall time from the beginning to the end of the collection. Treat this metric as your basic performance *baseline* against which you will compare subsequent runs of the application. The goal of your optimization is to reduce the value of this metric.
- Event-based performance metrics. Each metric is an event ratio provided by Intel architects. Mouse over the yellow icon ⓘ to see the metric description and formula used for the metric calculation.
- Values calculated for each metric based on the event count. VTune Amplifier XE highlights those values that exceed the threshold set for the corresponding metric. Such a value highlighted in pink signifies an application-level hardware issue. The text below a metric with the detected hardware issue describes the issue, potential cause and recommendations on the next steps, and displays a threshold formula used for calculation. Mouse over the truncated text to read a full description.

Quick look at the summary results discovers that the `matrix` application has the following issues:

- CPI (Clockticks per Instructions Retired) Rate
- Retire Stalls
- LLC Miss
- LLC Load Misses Serviced by Remote DRAM
- Execution Stalls
- Data Sharing

## Identify the Hardware Issues

Click the **Bottom-up** tab to open the Bottom-up window and see how each program unit performs against the event-based metrics. Each row represents a program unit and percentage of the CPU cycles used by this unit. Program units that take more than 5% of the CPU time are considered hotspots. This means that by resolving a hardware issue that, for example, took about 20% of the CPU cycles, you can obtain 20% optimization for the hotspot.

By default, the VTune Amplifier XE sorts data in the descending order by Clockticks and provides the hotspots at the top of the list.

The screenshot shows the Intel VTune Amplifier XE 2011 interface. The 'Bottom-up' tab is active, displaying a table of hardware event counts and metrics for various functions. The 'multiply1' function is highlighted as a hotspot. A tooltip is visible over the 'CPI Rate' cell, explaining that a high CPI rate can be caused by memory stalls, instruction starvation, branch misprediction, or long latency instructions. Below the table, there is a timeline view showing threads and hardware events over time.

/Function	CPU_CLK_...	INST_RETIRE...	CPI Rate	Retire Stalls	LLC Miss	LLC Loa...	Con...	Instr...	Bran...	Exec...	Data Shar...	Module
multiply1	1,281,854,000,000	239,542,000,000	5.351	0.902	0.592	0.340	0.000	-0.024	0.000	0.543	0.026	matrix.exe mu
HalReturnToFirmware	294,000,000	6,000,000	49.000	0.884	0.265	0.000	0.000	0.354	0.000	0.884	0.000	hal.dll Hal

Tooltip for CPI Rate: The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

Threshold: ( Clockticks where Hardware Event Type is CPU\_CLK\_UNHALTED.THREAD / Hardware Event Count where Hardware Event Type is INST\_RETIRED.ANY > 1.0 )  
 Clockticks where Hardware Event Type is CPU\_CLK\_UNHALTED.THREAD / query all Clockticks where Hardware Event Type is CPU\_CLK\_UNHALTED.THREAD > 0.05 )

Selected 1 row(s): 1,281,854,000,000 239,542,000,000

You see that the `multiply1` function is the most obvious hotspot in the `matrix` application. It has the highest event count (Clockticks and Instructions Retired events) and most of the hardware issues were also detected during execution of this function.

**NOTE** Mouse over a column header with an event-based metric name to see the metric description. Mouse over a highlighted cell to read the description of the hardware issue detected for the program unit.

For the `multiply1` function, the VTune Amplifier XE highlights the same issues (except for the Data Sharing issue) that were detected as the issues affecting the performance of the whole application:

- **CPI Rate** is high (>1). Potential causes are memory stalls, instruction starvation, branch misprediction, or long-latency instruction. To define the cause for your code, explore other metrics in the Bottom-up window.
- The **Retire Stalls** metric shows that during the execution of the `multiply1` function, about 90% (0.902) of CPU cycles were waiting for data to arrive. This may result from branch misprediction, instruction starvation, long latency operations, and other issues. Once you have located the stalled instructions in your code, analyze metrics such as LLC Miss, Execution Stalls, Remote Accesses, Data Sharing, and Contested Accesses. You can also look for long-latency instructions like divisions and string operations to understand the cause.



- **LLC misses** metric shows that about 60% (0.592) of CPU cycles were spent waiting for LLC load misses to be serviced. Possible optimizations are to reduce data working set size, improve data access locality, blocking and consuming data in chunks that fit in the LLC, or better exploit hardware prefetchers. Consider using software prefetchers but beware that they can increase latency by interfering with normal loads and can increase pressure on the memory system.
- **LLC Load Misses Serviced by Remote DRAM** metric shows that 34% (0.340) of cycles were spent servicing memory requests from remote DRAM. Wherever possible, try to consistently use data on the same core or at least the same package, as it was allocated on.
- **Execution Stalls** metric shows that 54% (0.543) of cycles were spent with no micro-operations executed. Look for long-latency operations at code regions with high execution stalls and try to use alternative methods or lower latency operations. For example, consider replacing `div` operations with right-shifts or try to reduce the latency of memory accesses.

## Recap

You analyzed the data provided in the Hardware Issues viewpoint, explored the event-based metrics, and identified the areas where your sample application had hardware issues. Also, you were able to identify the exact function with poor performance per metrics and that could be a good candidate for further analysis.

## Key Terms and Concepts

- Term: [viewpoint](#), [baseline](#), [Elapsed time](#)
- Concept: [Event-based Sampling Analysis](#), [Event-based Metrics](#)

## Next Step

[Analyze Code](#)

## Analyze Code



You identified a hotspot function with a number of hardware issues. Double-click the `multiply1` function in the Bottom-up window to open the source code:

Line	Source	INST_RETIRED. ANY by Package	CPU_CLK_U... THREAD by Package	CPU_CLK_U... REF by Package	OF A t
31	<code>void multiply1(int msize, int tidx, int numt, TYPE a[][NUM]</code>				
32	<code>{</code>				
33	<code>int i,j,k;</code>				
34	<code></code>				
35	<code>// Naive implementation</code>				
36	<code>for(i=tidx; i&lt;msize; i=i+numt) {</code>		2	29	29
37	<code>for(j=0; j&lt;msize; j++) {</code>	735	8,079	9,129	
38	<code>for(k=0; k&lt;msize; k++) {</code>				
39	<code>c[i][j] = c[i][j] + a[i][k] * b[k][j];</code>	38,795	329,350	430,403	
40	<code>}</code>				
41	<code>}</code>				
42	<code>}</code>				

The table below explains some of the features available in the Source pane when viewing the event-based sampling analysis data.

- 1 Source pane displaying the source code of the application, which is available if the function symbol information is available. The code line that took the highest number of Clockticks samples is highlighted. The source code in the Source pane is not editable.
- 2 Values per hardware event attributed to a particular code line. By default, the data is sorted by the Clockticks event count. Focus on the events that constitute the metrics identified as performance-critical in the Bottom-up window. To identify these events, mouse over the metric column header in the Bottom-up window. Drag-and-drop the columns to organize the view for your convenience. VTune Amplifier XE remembers your settings and restores them each time you open the viewpoint.
- 3 Hotspot navigation buttons to switch between code lines that took a long time to execute.
- 4 Source file editor button to open and edit your code in the default editor.
- 5 Assembly button to toggle in the Assembly pane that displays assembly instructions for the selected function.

In the Source pane for the `multiply1` function, you see that line 39 took the most of the Clockticks event samples during execution. This code section multiplies matrices in the loop but ineffectively accesses the memory. Focus on this section and try to reduce the memory issues.

## Recap

You analyzed the code for the hotspot function identified in the Bottom-up window and located the hotspot line that generated a high number of CPU Clockticks.

## Key Terms and Concepts

- Concept: [Event Skid](#)

## Next Step

[Resolve Issue](#)

# Resolve Issue



In the Source pane, you identified that in the `multiply1` function the code line 39 resulted in the highest values for the Clockticks event. To solve this issue, do the following:

- [Change the multiplication algorithm](#) and, if using the Intel® compiler, enable vectorization.
- [Re-run the analysis to verify optimization](#).

## Change Algorithm



**NOTE** The proposed solution is one of the multiple ways to optimize the memory access and is used for demonstration purposes only.

1. Open the `matrix.c` file from the **Source Files** of the **matrix** project.

For this sample, the `matrix.c` file is used to initialize the functions used in the `multiply.c` file.

2. In line 90, replace the `multiply1` function name with the `multiply2` function.

This new function uses the loop interchange mechanism that optimizes the memory access in the code.

```

matrix.c
77 #ifdef WIN32
78 DWORD WINAPI ThreadFunction(LPVOID ptr)
79 #else
80 void *ThreadFunction(void *ptr)
81 #endif
82 {
83     _tparam* par = (_tparam*)ptr;
84     assert(par->numt > 0);
85     assert(par->a != NULL);
86     assert(par->b != NULL);
87     assert(par->c != NULL);
88     assert((par->msize % par->numt) == 0);
89
90     multiply2(par->msize,
91             par->tidx,
92             par->numt,
93             par->a,
94             par->b,
95             par->c);
96
97 #ifdef WIN32
98     return 0;
99 #else
100 //printf("exit thread function\n")
101 pthread_exit((void *)0);
102 #endif

```

```

multiply.c
36     for(i=tidx; i<msize; i=i+numt) {
37         for(j=0; j<msize; j++) {
38             for(k=0; k<msize; k++) {
39                 c[i][j] = c[i][j] + a[i][k];
40             }
41         }
42     }
43 }
44 void multiply2(int msize, int tidx, int numt, int *a, int *b, int *c)
45 {
46     int i,j,k;
47
48     // Loop interchange
49     for(i=tidx; i<msize; i=i+numt) {
50         for(k=0; k<msize; k++) {
51             #pragma ivdep
52             for(j=0; j<msize; j++) {
53                 c[i][j] = c[i][j] + a[i][k];
54             }
55         }
56     }
57 }
58 void multiply3(int msize, int tidx, int numt, int *a, int *b, int *c)
59 {
60     int i,j,k,i0,j0,k0,ibeg,ibound,istep;
61

```

The proposed optimization assumes you may use the Intel® C++ Compiler to build the code. Intel compiler helps vectorize the data, which means that it uses SIMD instructions that can work with several data elements simultaneously. If only one source file is used, the Intel compiler enables vectorization automatically. The current sample uses several source files, that is why the `multiply2` function uses `#pragma ivdep` to instruct the compiler to ignore assumed vector dependencies. This information lets the compiler enable the Supplemental Streaming SIMD Extensions (SSSE).

### 3. Save files and rebuild the project using the compiler of your choice.

If you have the Intel® Composer XE installed, you may use it to build the project with the Intel® C++ Compiler XE. To do this, select **Intel Composer XE > Use Intel C++...** from the Visual Studio **Project** menu and then **Build > Rebuild matrix**.

## Verify Optimization

### 1. From the VTune Amplifier XE toolbar, click the **New Analysis** button and select **Quick Intel(R) Microarchitecture Code Name Nehalem - General Exploration Analysis**.

VTune Amplifier XE reruns the General Exploration analysis for the updated `matrix` target and creates a new result, `r001ge`, that opens automatically.

### 2. In the `r001ge` result, click the **Summary** tab to see the Elapsed time value for the optimized code:



**Elapsed Time:** 9.122s

Hardware Event Count:	601,158,000,000
CPU_CLK_UNHALTED.THREAD:	3.37658e+011
INST_RETIRED.ANY:	2.635e+011
CPI Rate:	1.281
The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or...	
Retire Stalls:	0.721
A high number of retire stalls is detected. This may result from branch misprediction, instruction starvation, long latency...	
LLC Miss:	0.001
LLC Load Misses Serviced By Remote DRAM:	0.000
Contested Accesses:	0.000
Instruction Starvation:	0.130
Branch Mispredict:	0.006
Execution Stalls:	0.008
Data Sharing:	0.000
Paused Time:	0s

You see that the Elapsed time has reduced from 56.740 seconds to 9.122 seconds and the VTune Amplifier XE now identifies only two types of issues for the application performance: high CPI Rate and Retire Stalls.

## Recap

You solved the memory access issue for the sample application by interchanging the loops and sped up the execution time. You also considered using the Intel compiler to enable instruction vectorization.

## Key Terms and Concepts

- Concept: [Event-based Sampling Analysis](#)

## Next Step

[Resolve Next Issue](#)

### 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

## Resolve Next Issue



You got a significant performance boost by optimizing the memory access for the `multiply1` function. According to the data provided in the Summary window for your updated result, `r001ge`, you still have high CPI rate and Retire Stalls issues. You can try to optimize your code further following the steps below:

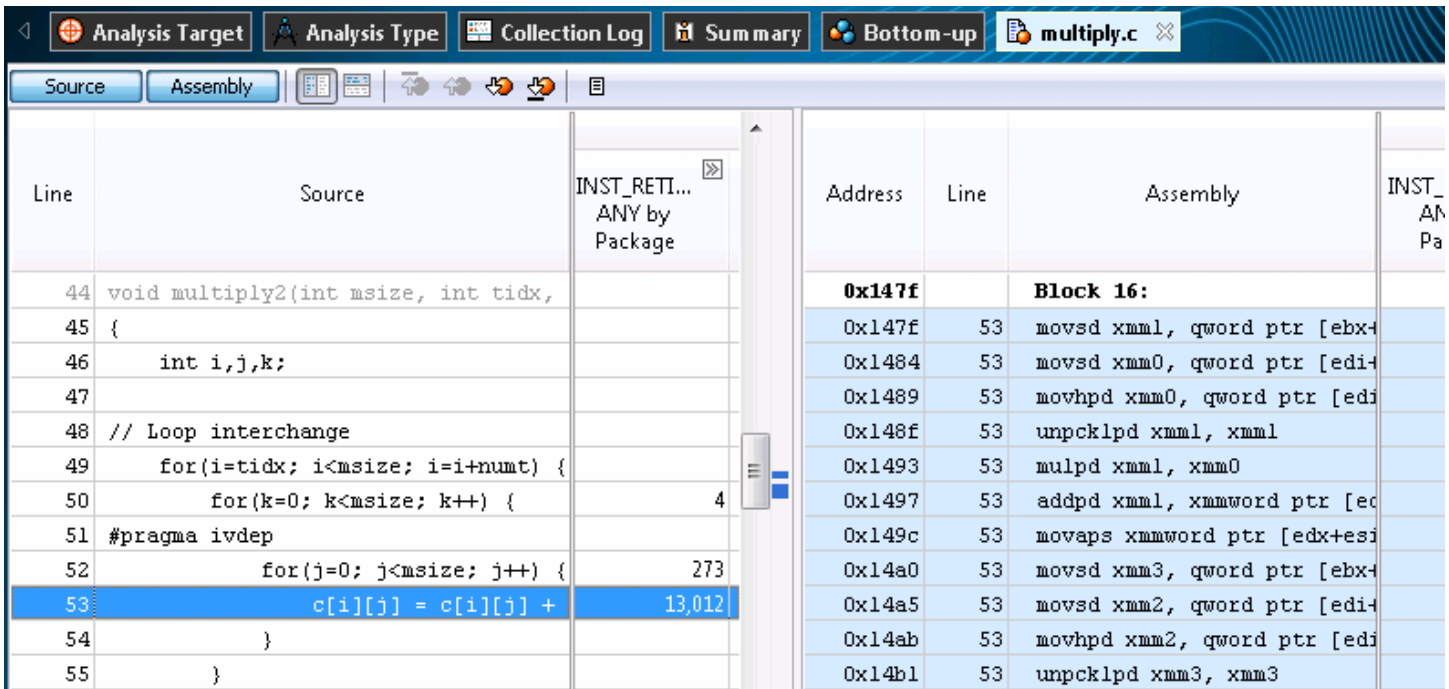
- Analyze results after optimization
- Use more advanced algorithms
- Verify optimization

### Analyze Results after Optimization

To get more details on the issues that still affect the performance of the `matrix` application, switch to the Bottom-up window:

Function	CPU_CLK...	INST_RETIRE...	CPI Rate	Retire Stalls	LLC Miss	LLC Load Misses ...	Con... Acc...	Instr... Star...	Bran... Mis...
multiply2	330,080,000,000	256,324,000,000	1.288	0.698	0.006	0.002	0.000	0.140	0.000
init_arr	120,000,000	142,000,000	0.845	2.167	0.000	0.000	0.000	0.000	0.000
HalReturnToFirmware	112,000,000	6,000,000	18.667	2.321	0.232	0.000	0.000	0.232	0.000
memset	90,000,000	2,000,000	45.000	0.000	0.000	0.000	0.000	0.000	0.000
KeUpdateRunTime	36,000,000	8,000,000	4.500	0.722	0.722	0.000	0.000	0.722	0.000
Selected 1 row(s):	330,080,000,000	256,324,000,000							

You see that the `multiply2` function (in fact, updated `multiply1` function) is still a hotspot. Double-click this function to view the source code and click both the **Source** and **Assembly** buttons on the toolbar to enable the Source and Assembly panes.



In the Source pane, the VTune Amplifier XE highlights line 53 that took the highest number of Clockticks samples. This is again the section where matrices are multiplied. The Assembly pane is automatically synchronized with the Source pane. It highlights the basic blocks corresponding to the code line highlighted in the Source pane. If you compiled the application with the Intel® Compiler, you can see that highlighted block 156 includes vectorization instructions added after your previous optimization. All vectorization instructions have the *p* (packed) postfix (for example, *mulpd*). You may use the `/Qvec-report3` option of the Intel compiler to generate the compiler optimization report and see which cycles were not vectorized and why. For more details, see the Intel compiler documentation.

### Use More Advanced Algorithms

1. Open the `matrix.c` file from the Source Files of the **matrix** project.
2. In line 90, replace the `multiply2` function name with the `multiply3` function.

This function enables uploading the matrix data by blocks.

```

matrix.c
(Global Scope)
79 #else
80 void *ThreadFunction(void *ptr)
81 #endif
82 {
83     _tparam* par = (_tparam*)ptr;
84     assert(par->numt > 0);
85     assert(par->a != NULL);
86     assert(par->b != NULL);
87     assert(par->c != NULL);
88     assert( (par->msize % par->nu
89
90     multiply3( par->msize,
91               par->tidx,
92               par->numt,
93               par->a,
94               par->b,
95               par->c
96               );
97 #ifdef WIN32
98     return 0;
99 #else
100 //printf("exit thread functio
101 pthread_exit( (void *)0 );
102 #endif
103 }
104
105 int main()
106 {
107 #ifdef WIN32
108     clock_t start=0.0, stop=0.0;
109 #else

```

```

multiply.c
(Global Scope)
53     c[i][j] = c[i][j] + a[i][k] * b[k][j];
54     }
55     }
56     }
57 }
58 void multiply3(int msize, int tidx, int numt, TYPE a[][NUM]
59 {
60     int i,j,k,i0,j0,k0,ibeg,ibound,istep,mblock;
61
62 // Step3: Cache blocking
63 istep = msize / numt;
64 ibeg = tidx * istep;
65 ibound = ibeg + istep;
66 mblock = MATRIX_BLOCK_SIZE;
67
68 for (i0 = ibeg; i0 < ibound; i0 +=mblock) {
69     for (k0 = 0; k0 < msize; k0 += mblock) {
70         for (j0 =0; j0 < msize; j0 += mblock) {
71             for (i = i0; i < i0 + mblock; i++) {
72                 for (k = k0; k < k0 + mblock; k++) {
73
74 #pragma unroll(8)
75 #pragma ivdep
76                     for (j = j0; j < j0 + mblock; j++)
77                         c[i][j] = c[i][j] + a[i][k] *
78
79
80
81
82
83 }

```

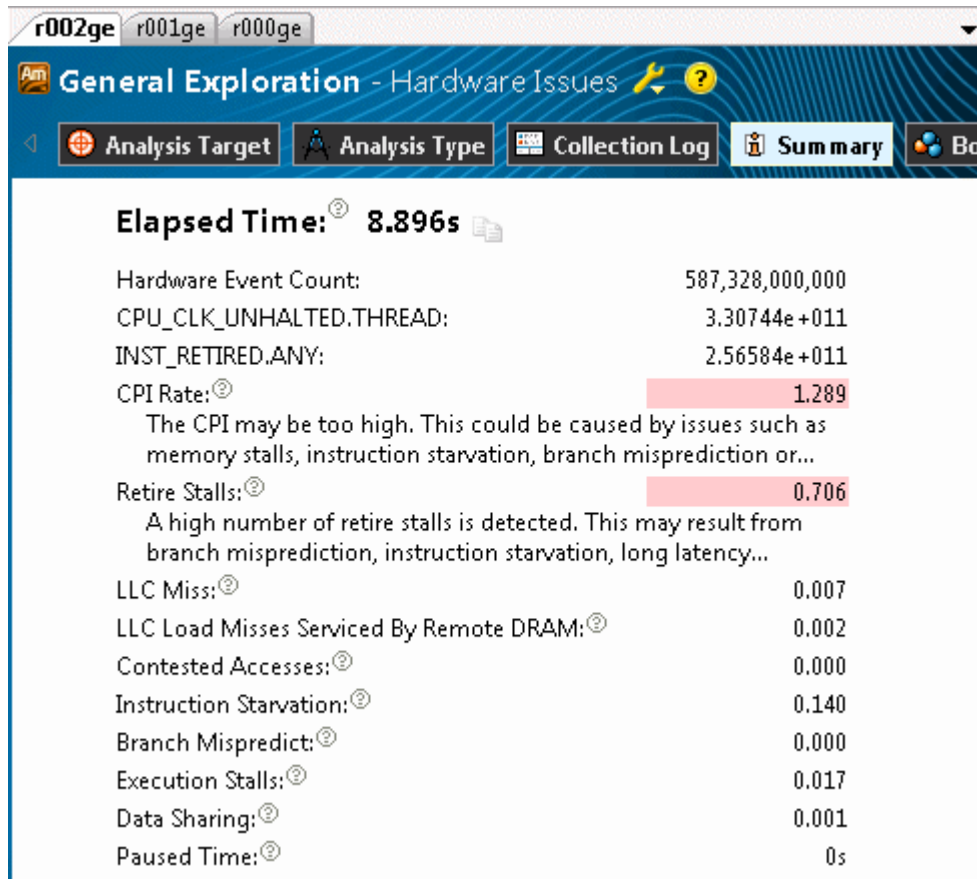
3. Save the files and rebuild the project.

## Verify Optimization

1. From the VTune Amplifier XE toolbar, click the  **New Analysis** button and select **Quick Intel(R) Microarchitecture Code Name Nehalem - General Exploration Analysis**.

VTune Amplifier XE reruns the General Exploration analysis for the updated `matrix` target and creates a new result, `r002ge`, that opens automatically.

2. In the `r002ge` result, click the **Summary** tab to see the Elapsed time value for the optimized code:



You see that the Elapsed time has reduced a little: from 9.122 seconds to 8.896 seconds but the hardware issues identified in the previous run, CPI Rate and Retire Stalls, stayed practically the same. This means that there is more room for improvement and you can try other, more effective, mechanisms of matrix multiplication.

## Recap

You tried optimizing the mechanism of matrix multiplication and obtained 0.2 seconds of optimization in the application execution time.

## Key Terms and Concepts

- Concept: [Event-based Sampling Analysis](#), [Event-based Metrics](#)

## Next Step

Read [Summary](#)

## Summary



You have completed the Identifying Hotspot Issues tutorial. Here are some important things to remember when using the Intel® VTune™ Amplifier XE to analyze your code for hardware issues:

### Step 1. Choose and Build Your Target

- Configure the Microsoft\* symbol server and your project properties to get the most accurate results for system and user binaries and to analyze the performance of your application at the code line level.

- Use the **Project Properties: Target** tab to choose and configure your analysis target. For Visual Studio\* projects, the analysis target settings are inherited automatically.

## Step 2. Run Analysis

- Use the **Analysis Type** configuration window to choose, configure, and run the analysis. You may choose between a predefined analysis type like the General Exploration type used in this tutorial, or create a new custom analysis type and add events of your choice. For more details on the custom collection, see the *Creating a New Analysis Type* topic in the product online help.

## Step 3. Interpret Results and Resolve the Issue

- Start analyzing the performance of your application from the Summary window to explore the event-based performance metrics for the whole application. Mouse over the yellow help icons to read the metric descriptions. Use the Elapsed time value as your performance baseline.
- Move to the Bottom-up window and analyze the performance per function. Focus on the *hotspots* - functions that took the highest Clockticks event count. By default, they are located at the top of the table. Analyze the hardware issues detected for the hotspot functions. Hardware issues are highlighted in pink. Mouse over a highlighted value to read the issues description and see the threshold formula.
- Double-click the hotspot function in the Bottom-up pane to open its source code at the code line that took the highest Clockticks event count.
- Consider using Intel® Compiler, part of the Intel® Composer XE, to vectorize instructions. Explore the compiler documentation for more details.

### 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

# More Resources

---

## Getting Help

---

Intel® VTune™ 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 .

From the Visual Studio user interface, select **Help > Intel VTune Amplifier XE 2011 > Getting Started Tutorials** and explore available tutorials.

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

### Browsing Help


In the Visual Studio IDE, you can browse and search for topics in different ways:

- Use **Help > Contents** to open the Contents window and browse the Table of Contents.
- To view help for the VTune Amplifier XE directly, select **Help > Intel VTune Amplifier XE 2011 Help**.
- Use **Help > Index** to open the Index window and access an index to VTune Amplifier XE topics. Either type in the keyword you are looking for, or scroll through the list of keywords.
- Use **Help > Search** to open the Search page and search the full text of topics in the help.


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

### Locating Intel Topics in the Document Explorer

To filter the documentation so that only the Intel documentation appears, select **Help > Contents** from the Visual Studio user interface. In the **Filtered by:** drop-down list, select **Intel**.

To determine where the currently displayed topic appears in the table of contents (TOC), click the  **Sync with Table of Contents** button on the Visual Studio toolbar to highlight the topic in the Contents pane.

### 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.

### Activating Intel Search Filters in the Document Explorer

With Microsoft Visual Studio 2005 and 2008, you can include Intel documentation in all search results by checking the **Intel** search filter box for the **Language**, **Technology**, and **Content Type** categories. You must check the **Intel** search box for all three categories to include Intel documentation in your searches. Unchecking all three **Intel** search boxes excludes Intel documentation from search results. The Intel search filters work in combination with other search options for each category.

### 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:

- **? Help:** In Visual Studio, click the ? button, in the upper-right corner of the dialog box or pane to get help for the dialog box or pane.
- **F1 Help:** Press F1 to get help for an active dialog box, property page, pane, or window.
- **Dynamic Help:** In Visual Studio 2005/2008, select **Help > Dynamic Help** to open the Dynamic Help window, which displays links to relevant help topics for the current window.

## Product Website and Support

---

### Product Website and Support

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

- <http://software.intel.com/en-us/articles/tools/>  
Intel® 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.



# Intel® VTune™ Amplifier XE Tutorials Troubleshooting

## 5



## Troubleshooting

### Problem: Cannot open samples

The sample projects are Visual Studio\* 2005 projects. You may have a problem opening the sample if you have a later version of Visual Studio\* software.

**Solution:** Use the conversion wizard to convert the solution/projects to the newer version.

### Problem: Product is not recognized

If you installed a new version of Visual Studio\* software, the previously installed Intel® VTune™ Amplifier XE may not appear in the new installation.


**Solution 1:** If you have the VTune Amplifier XE installation execution file, run the installation program, select **Modify**, and follow the instructions to reintegrate the VTune Amplifier XE with your new version of Visual Studio\* software.

#### Solution 2:

1. Go to **Control Panel > Add or Remove Programs**.
2. Select the VTune Amplifier XE and select **Modify**.
3. Follow the instructions to reintegrate the VTune Amplifier XE with your new version of Visual Studio\* software.

### Problem: The Project Properties function is disabled


The **Intel VTune Amplifier XE 2011 Project Properties** option does not appear on the **Project** menu,

and the  icon is disabled on the VTune Amplifier XE toolbar.

**Solution:** Make sure the item highlighted in the **Solution Explorer** is a valid project recognized by Visual Studio\* software or a VTune Amplifier XE result. (The **My Amplifier XE Results** folder is a *virtual* project.)

### 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.