



Intel(R) Thread Profiler for Linux*

Getting Started Guide

Intel® Thread Profiler helps you improve the performance of applications threaded with OpenMP* or POSIX* threads (Pthreads*). Use Thread Profiler to:

- Identify bottlenecks that limit the parallel performance of your multi-threaded application.
- Locate synchronization delays, stalled threads, excessive blocking time, and ineffective utilization of processors.
- Find the best sections of code to optimize for sequential performance and for threaded performance.
- Compare scalability across different numbers of processors or using different threading methods.

This guide presents a threaded code example and shows you how to use the command-line version of the Thread Profiler (`tprofile_cl`) to identify performance issues. After completing this guide, you will be ready to analyze and optimize your own code using Thread Profiler.

TIP: To quickly start using Thread Profiler, print this short guide and walk through the example provided.

Contents

1	Build the Sample Code.....	3
2	Collect Data	4
3	View Data	6
4	Analyze Results	7
5	Correct the Code.....	14
6	Next Steps.....	18



Disclaimer and 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. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications. 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.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECS, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Chips, Core Inside, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Pentium Inside, skool, Sound Mark, The Computer Inside., The Journey Inside, VTune, Xeon, Xeon Inside and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2007, Intel Corporation.



1 Build the Sample Code

The `primes` example is a prime number generator that uses the POSIX* threading API. The code identifies and tallies the prime numbers from one to 100,000 by testing whether odd numbers are evenly divisible by smaller odd factors. The code generates four threads to do the work.

1.1 To Build the Sample Code:

Copy the `primes` directory to a convenient workspace. By default, the samples code directory is installed in: `/opt/intel/itt/tprofile/samples/primes`.

Build the `primes` executable using one of the following methods:

To build using the GNU* C/C++ compiler:

1. Go to your copy of the `/primes` directory.
2. Enter the following command:

```
% make
```

This command builds the executable files `primes.gcc`, `primesBalanced.gcc`, and `primesOpt.gcc` using the switch `-g` to turn on debug information. Thread Profiler performs *binary instrumentation* on the executable to enable monitoring of POSIX* threads API calls.

To build using the Intel® C++ Compiler for Linux*:

1. At the shell command prompt, set up the Intel compiler environment by issuing:

```
% source <path_to_tprofile_bin_directory>/iccvars.csh
```

Or

```
% . <path_to_tprofile_bin_directory>/iccvars.sh
```

2. Go to the `primes` directory and enter the command:

```
% make icc
```

The compiler builds executable files `primes.icc`, `primesBalanced.icc` and `primesOpt.icc`. The switch `-g` turns on debug information.

NOTE: To analyze OpenMP* applications, the Intel compilers must be used.



2 Collect Data

Set up the Intel(R) Thread Profiler environment by issuing:

```
% source <path_to_tprofile_bin_directory>/tprofilevars.csh
```

or

```
% . <path_to_tprofile_bin_directory>/tprofilevars.sh
```

NOTE: In addition to modifying the PATH and LD_LIBRARY_PATH environment variables, the required variables ISM_INST_DIR and VTUNE_GLOBAL_DIR are also set.

The executable used in this example is compiled with the GNU* C/C++ compiler. Start the command line tool by entering:

```
% tprofile_cl primes.gcc
```

tprofile_cl begins by performing binary instrumentation of the executable primes.gcc and its associated modules. For large applications, this process can take many minutes.

During instrumentation, each instrumented module has a line with an estimate of the instrumentation completion time and the level of instrumentation. By default, user executables and libraries are instrumented at the **API Imports** level. This permits all calls to the POSIX* API to be monitored. For most system libraries and other pre-compiled libraries without debug information, the **Minimal** instrumentation level is used. This permits detection of module loading for proper reporting and symbol resolution.

After completing the Acitivity run, you should see output similar to the following:

```
Building project
Instrumenting
 14% primes.gcc      ( API Imports ):...
 42% libc-2.3.2.so   ( Minimal ):...
 57% libgcc_s-3.2.3-20040414.so.1 ( API Imports ):...
 71% libm-2.3.2.so   ( API Imports ):...
 85% libpthread-0.60.so ( Minimal ):...
100% libstdc++.so.5 ( Minimal ):... Building project
Running: <path>/primes.gcc
Determining primes from 1 - 100000
Found 9592 primes
Application finished
Thread Profiler      Summary Report 3.1
application:         <path>/primes.gcc
collection:          <current date and time>
```



```
runtime:                2.76127s
# of processors:        4
# of threads:           5
# of waits:             11
wait frequency:        3.98368
average concurrency:   2.32811
Concurrency:
  0 [.....] 0.00424% 0.000116708
  1 [###.....] 28.1% 0.774734
  2 [###.....] 26.6% 0.733568
  3 [###.....] 29.1% 0.801588
  4 [##.....] 16.1% 0.443197
```

For further analysis, the output in the threadprofiler directory can be viewed in the GUI (available on Microsoft* Windows*)

1. Copy the contents of the threadprofiler directory to the Windows machine, or use a network-mounted drive.
2. From the Intel(R) Thread Profiler GUI, use File->Open File and select the tprofile.<pid>.tp file.

* Other names and brands are the property of their respective owners.

Intel(R) Thread Profiler creates a directory for temporary data storage. By default, the name of the directory is /tmp/<login_name>_tp_cl_cache. To specify an alternate directory use the -d option. For example:

```
% tprofile_cl -d /home/sample_data primes.gcc
```

NOTE: The arguments for tprofile_cl must appear before the specification of the executable. Any command line arguments that appear after the specification are passed to the executable.

The summary report file lists the time for the instrumented run. The wait frequency is the number of waits per second, that is, the total number of waits divided by the runtime. The average concurrency is computed by taking the sum of the times at each concurrency level, times the concurrency level and dividing that sum by the runtime.

The concurrency level 0 represents time when all threads in the application are either sleeping or are blocked. The tiny amount of time between the termination of the last worker thread and the resumption of the main thread making a join call is one such occurrence. Any time spent by one thread waiting for blocking system I/O also contributes to this level if no other threads are active. The main thread starts at level 1. Creating new threads creates the opportunity for two or more threads to run



concurrently, leading to higher concurrency levels. The statistics may vary from run to run due to non-deterministic scheduling differences.


Intel(R) Thread Profiler also creates a directory named `threadprofiler` in the current working directory. This directory holds the output data files: `tprofile.<pid>.tp`, `trprofile.<pid>.tpd`, and `tprofile.<pid>.tps`. The binary instrumentation data file `bistro.tp` is also present. To set a different results directory, use the `-r` option. For example:

```
% tprofile_cl -r primes_gcc primes.gcc
```

In this example, the `primes_gcc` directory is created in the current working directory.

3 View the Data

The data files in the results directory (`threadprofiler` by default) contain a great deal more information than is presented in the summary report. This data can be viewed with the Intel® Thread Profiler for Windows*. To do so, perform the following:

1. Copy the contents of the results directory, the executable, and source file to a directory on a Windows* machine.
2. Start the Intel(R) Thread Profiler from, for example, **Start > All Programs > Intel(R) Software Development Tools > Intel(R) Thread Profiler >  Intel(R) Thread Profiler.**
3. In the **Easy Start** dialog box, click **Close**. You want to view a data file, not start a Thread Profiler Activity.
4. Select **File > Open file...** and browse to the directory with your data file. Select the `tprofile.<pid>.tp` file and click **Open**.
5. Thread Profiler loads the file. If the executable or source files are not in the current directory, Thread Profiler asks for the location of those files.

The exact results vary depending on the Linux system configuration. The number of processors available is the most significant detail. You should see two color charts, one with vertical bars and one with horizontal bars. Congratulations! You are now ready to identify and locate bottlenecks that are limiting the parallel performance of the sample software.



4 Analyze Results

In this section, you will walk through Intel® Thread Profiler’s main views to identify performance issues related to threading. You will then consider ways to improve the performance of the sample code.

4.1 Profile View

By default, Thread Profiler displays **Profile** and **Timeline** views as shown in Figure 1.

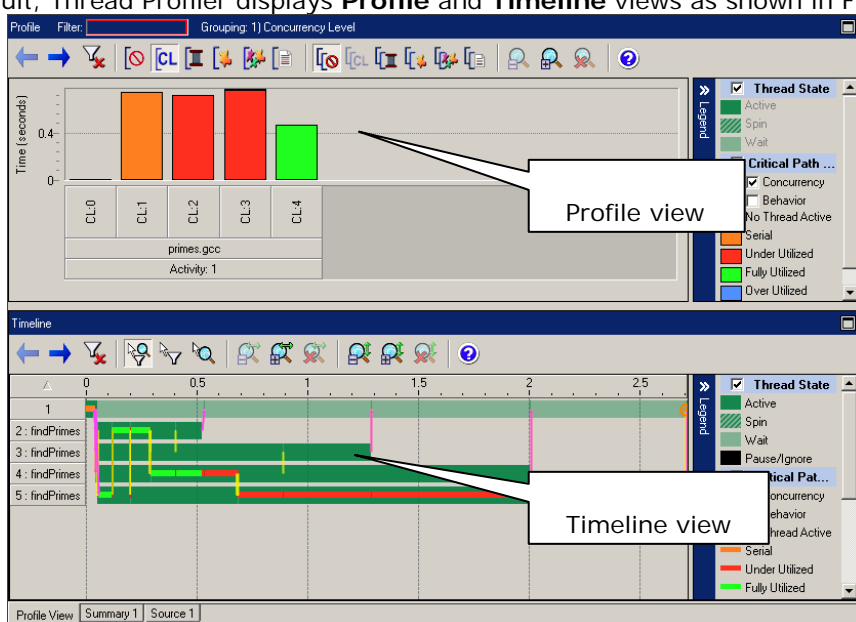


Figure 1: Profile and Timeline views are the default views for Intel(R) Thread Profiler Activity results.


The **Profile** view (on top) displays a high-level summary of the time spent on the critical path, decomposed into time categories. The **Timeline** view (on bottom) illustrates the behavior of your program over time.

By default, **Profile** view initially shows results grouped by **Concurrency Level**, the number of active threads executing at the same time on the critical path. It includes threads which are currently running or are queued but are not waiting at a defined waiting or blocking API.



Double-click the tallest red bar in **Profile** view to “drill down” to group by **Objects**.



Now the **Objects** button,  , on the **Profile** view’s toolbar is selected and you can compare times due to different software objects as shown in Figure 2.

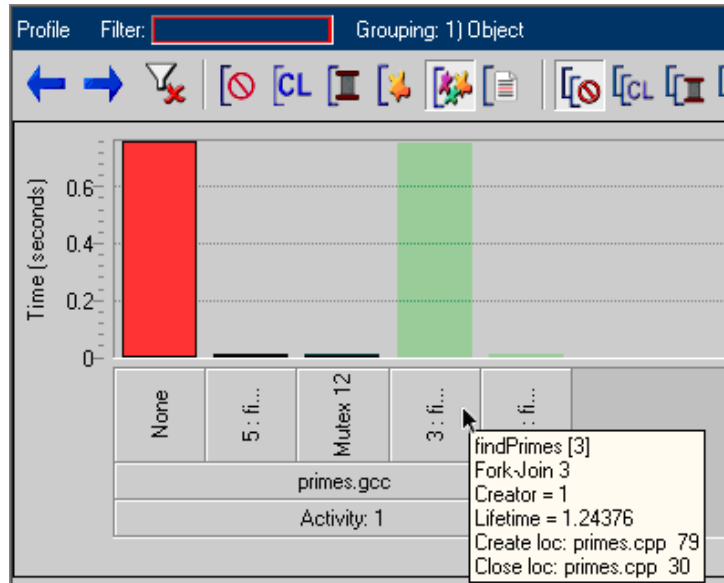
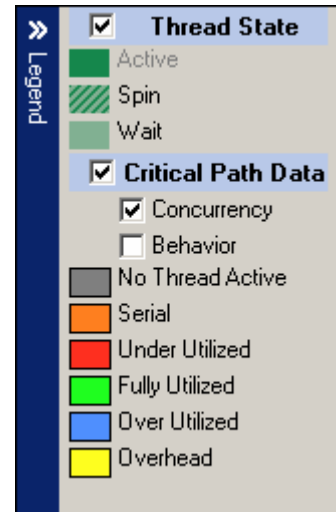


Figure 2: Profile view grouped by Objects

In this case, the majority of time was spent in Under Utilized (red) time. The next longest time was spent during a Wait (green halo). Hover your mouse over a column to see more detailed data.

Thread Profiler shows columns corresponding to objects that cause contention on the critical path. The color scheme is given in the **Legend** as shown in the opposite figure.

The Profile view legend displays the color coding used. Hover your mouse over the Legend entries to show definition tooltips.





In this example, the largest contributor to time on the critical path is not attributed to any blocking object, hence the label “None”. For this contributor no threads were blocked by a mutex, semaphore, critical section, fork-join, sleep, or message queue.

The bar is red, indicating that the application under-utilized the available processors for this time on the critical path. This example has up to five threads running on a machine with four processors. The example clearly does not fully utilize the available processors

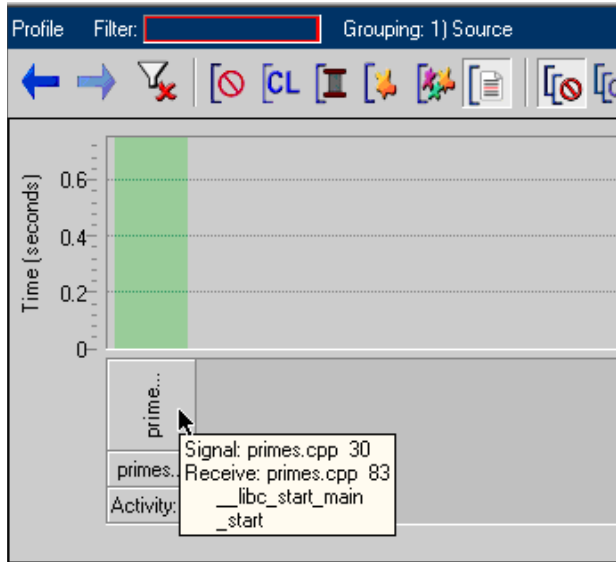
The next highest bar has a light green halo, indicating a thread is waiting. The object responsible for the wait is the fork-Join of thread number 3. The main thread is blocked waiting for this thread to complete its work and join.

In **Profile** view, you can also:

- Double-click any bar to “drill down” to the next logical level of grouping.
- Right-click any bar and select **Filter Selection** to filter data by the current selection.
- Right-click a bar and select **Filter and Group by >** to filter data and group by **Concurrency Level, Thread, Object, Object Type, or Source Stack**. You can also use the corresponding toolbar buttons to group data by the same categories.
- Apply second-level grouping using the secondary set of toolbar buttons:



- For certain types of data, you can right-click and select **Transition Source View** or **Creation/Entry Source View**.



Double-click an Object bar to group the data by Source.

Hovering your mouse over the Source label brings a tool tip with additional information.

Figure 3: Information on the green halo Source bar

Figure 3 shows the information available for the green Source bar. In this example, when the wait ended, thread 3 was executing a function at line 30. The waiting thread, the main thread, was blocked at line 83.

Double-click the Object bar to display the source code for the start and end of the transition involved. See Figure 4. Thread 1 spawned a worker thread which began running function `findPrimes`. Thread 1 is now waiting for that thread to terminate and join.



The figure displays two screenshots of the Source View window, showing the start and end of a transition. The top screenshot shows the start of the transition, and the bottom screenshot shows the end of the transition.

Top Screenshot (Signal):

- Signal: Thread 1
- Receive: Thread 1
- Sync Object: 3 : findPrimes
- Stack:
 - findPrimes(void *)
 - 'primes.cpp': 30
 - Path: Unknown

Address	Line	Source
	24	
	25	long primes[MAX_NUMBERS];
	26	long primeCount;
	27	pthread_mutex_t cs;
	28	
	29	void * findPrimes(void * arg)
0x5FD	30	{
0x5F8	31	long threadnum = *(long *) arg;
	32	long blocksize, start, end, stride;
	33	long number, factor;
	34	
	35	blocksize = MAX_NUMBERS/NUM_THREADS;
	36	

Bottom Screenshot (Receive):

- Signal: Thread 1
- Receive: Thread 1
- Sync Object: 3 : findPrimes
- Stack:
 - main
 - 'primes.cpp': 83
 - Path: Unknown
 - __libc_start_main
 - Address: 0x2c747
 - Module: libc.so.6
 - Path: /tmp/dlbeckma_tc_cl_cache
 - _start
 - Address: 0x560
 - Module: primes.gcc

Address	Line	Source
	77	{
0x6EC	78	tnums[i] = i;
0x6D7	79	pthread_create(&h[i], 0, findPrimes, (void *)
	80);
	81	}
0x6FF	82	for (i = 0; i < NUM_THREADS; ++i)
0x701	83	pthread_join (h[i], 0);
	84	
0x718	85	pthread_mutex_destroy(&cs);
	86	
0x725	87	printf("Found %d primes\n", primeCount);
	88	
	89	return 0;

Figure 4: View source code of start and end of the transition

4.2 Timeline View

Timeline view shows the contribution of each thread to the total program, whether on the default critical path or not.

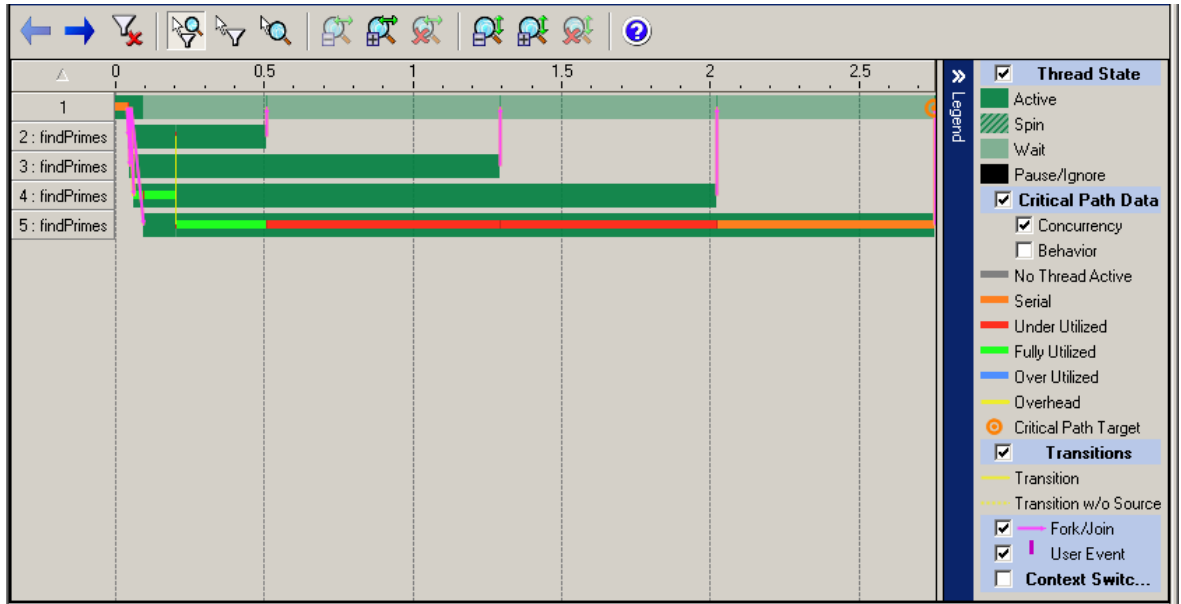


Figure 5: Timeline view shows the behavior of a program over time and across threads

Figure 5 shows that the four threads take increasingly longer to run. The range of numbers to test has been partitioned into four groups. A group is assigned to each thread. Thread 2, represented by the next-to top bar, works on a set of small numbers that take little time to check. Thread 3 checks a set of larger numbers, taking longer. Threads 4 and 5 check even larger numbers, taking longer to complete. Though the program accomplishes its task, the loading is clearly unbalanced. In the next section, you will explore ways to improve the parallelism in this code example. Meanwhile, take a closer look at the **Timeline** view.

Thread Profiler tracks the flow of all threads in the application. The critical path is the continuous path from the beginning of execution to the critical path target. By default, the critical path target is the end of program execution.

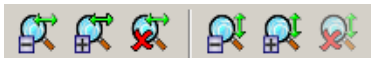
In **Timeline** view, you can use the **Legend** to hide or show the corresponding time categories that appear in the **Profile** view. You can also hide or show **Transitions**, **Forks** and **Joins**, and **User** events that you defined. When you mouse over the different elements in the **Timeline** view's **Legend**, the corresponding elements flash in the graph, helping you to locate or identify different elements in the graph.

In Timeline view you can also:

- Drag your mouse over a section of the **Timeline** view and release to zoom in on that section.



- Use the horizontal and vertical zoom buttons on the toolbar,



to help you focus on a particular section of the graph. This feature is particularly useful when you have vast quantities of densely displayed data.


- Double-click a bar to “drill-down” to **Source** view to see where an event is happening in your code.

4.3 Time Categories

Thread Profiler breaks time into different **Time Categories**, represented in the graphs and **Legends** by different colors. In this example, the critical path is composed of the following time categories:

- **Serial** times (shades of orange) indicate serial portions of the code.
- **Under Utilized** times (shades of red) indicate that the code is not fully utilizing all processors.
- **Fully Utilized** times (shades of green) indicate that portions of the code demonstrate good processor utilization.
- **Over Utilized** times (shades of blue) indicate that portions of the code use more threads than there are processors.

Ideally, on a multi-processor system, your code should be characterized by a predominance of **Fully Utilized** time (the sum of all the greens). All other time categories indicate opportunities for improving performance.

For more information on a certain view or legend, press F1 or click **Help** . To view the complete Thread Profiler help, choose **Help > Contents** from the menu. Figure 6 shows a sample help topic.

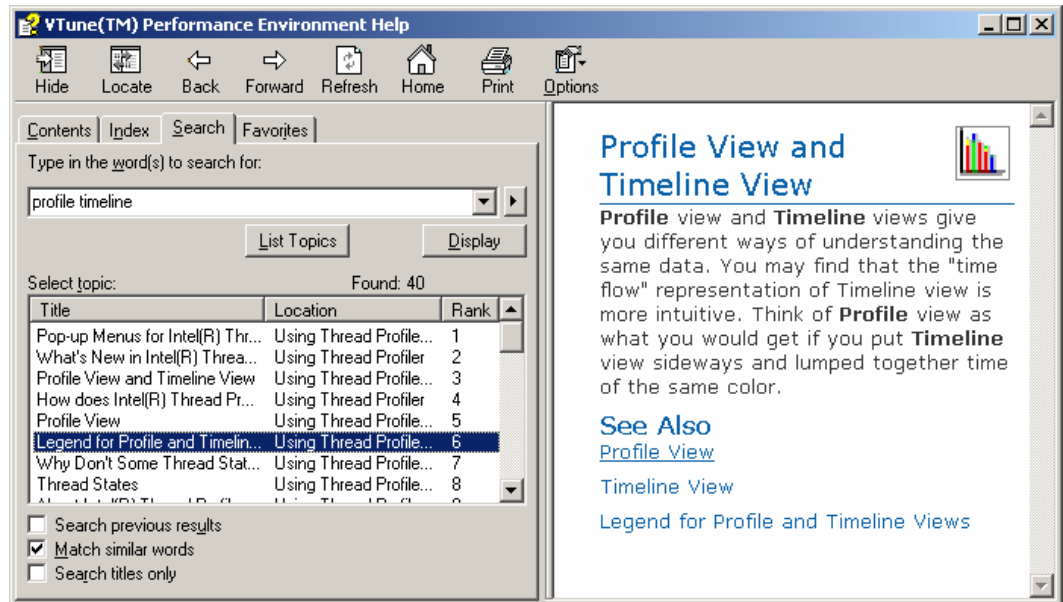



Figure 6: Press F1 to access help topics.

In the **Help** you can also:

- Click entries in the **Contents** pane to jump to a topic.
- Click the **Search** tab to open the search window to find a topic on a particular subject.
- Find tips for improving your code such as **Dealing with Impact Time** for valuable tips on ways to reduce the delays on the critical path.

- Click the locate button on the **Help** toolbar,  to browse to related topics in the **Contents**.

5 Correct the Code

To correct the imbalance found during analysis, try the solution presented in this section. A corrected version of the sample code is provided for you to check your results and note the differences.



5.1 Solution: Increase Parallel Execution Time

The revised code in the sample `primesBalanced.cpp` interleaves the numbers each thread checks so that each thread works on both small and large values. To tune to a particular machine configuration, the number of worker threads could also be changed. To verify that the imbalance problem identified by Thread Profiler as a performance issue is indeed fixed, do the following:

1. If the original run with `primes.gcc` produced the default results directory `threadprofiler`, consider renaming the directory to preserve the results. Another run using the default directory name will cause deletion of the original files.
2. Issue the command:

```
% tprofile_cl primesBalanced.gcc
```

Copy the contents of the results directory, the executable, and source files to the Windows machine.
3. Start the Intel® Thread Profiler. Select **File > Open file...** and browse to the directory with your data file. Select the `tprofile.<pid>.tp` file and click **Open**.

5.2 Analyze Results

In **Profile** view, you should see improvements. In the **Timeline** view, the threads now show improved workload balance over the original example. The goal is to have a **Profile** view with primarily **Fully Utilized** (green) execution times.

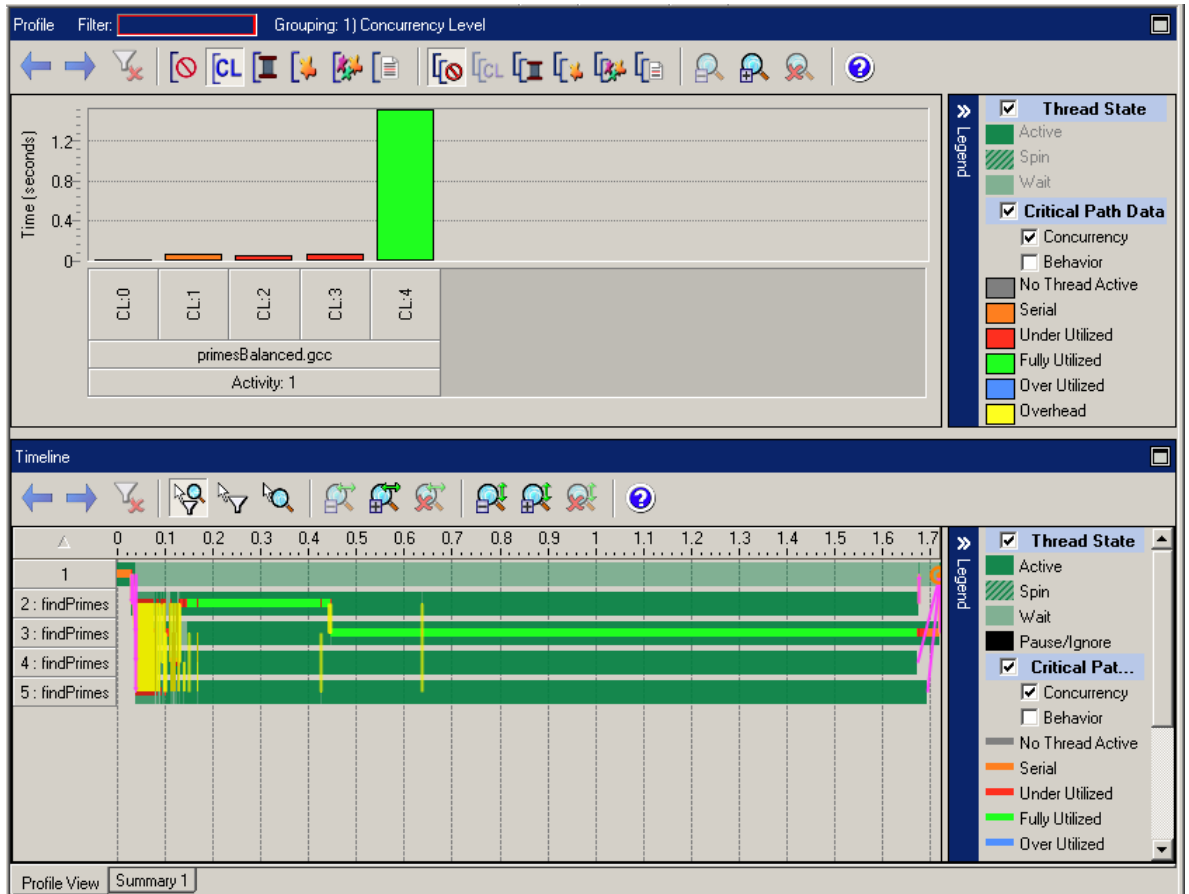


Figure 7: Revised code primesBalanced.gcc. The processors are more fully utilized, but added overhead is evident.

In the balanced example, each worker thread evaluates every fourth number in the range. A single thread works on both small and large numbers so the work is distributed more evenly. On the four processor machine used here, the four worker threads fully utilize the resources.

The improvement in the naive algorithm reveals another problem. As each prime number is discovered, a mutex is acquired before writing the result to global variables. Early on, all threads are finding primes and a great number of transitions occur. Unneeded overhead is introduced as a result. Zoom the **Timeline** view horizontally to expose the excessive activity.

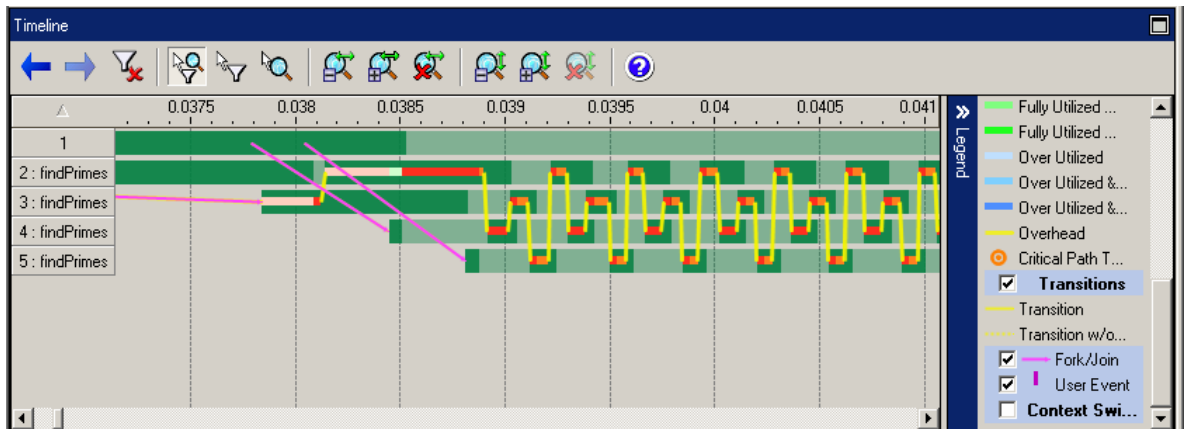


Figure 8: Horizontal zoom of the Timeline View reveals excessive transitions.

5.3 Solution: Decrease Overhead

The final version of the example, `primesOpt.gcc`, removes the transition overhead. Each thread stores its results locally. Only when it has completed searching its range does it acquire the mutex and report the results to the global variables. The mutex is only acquired four times during the entire run. Figure 10 shows that the optimized example is not only balanced, but unnecessary overhead has been eliminated.

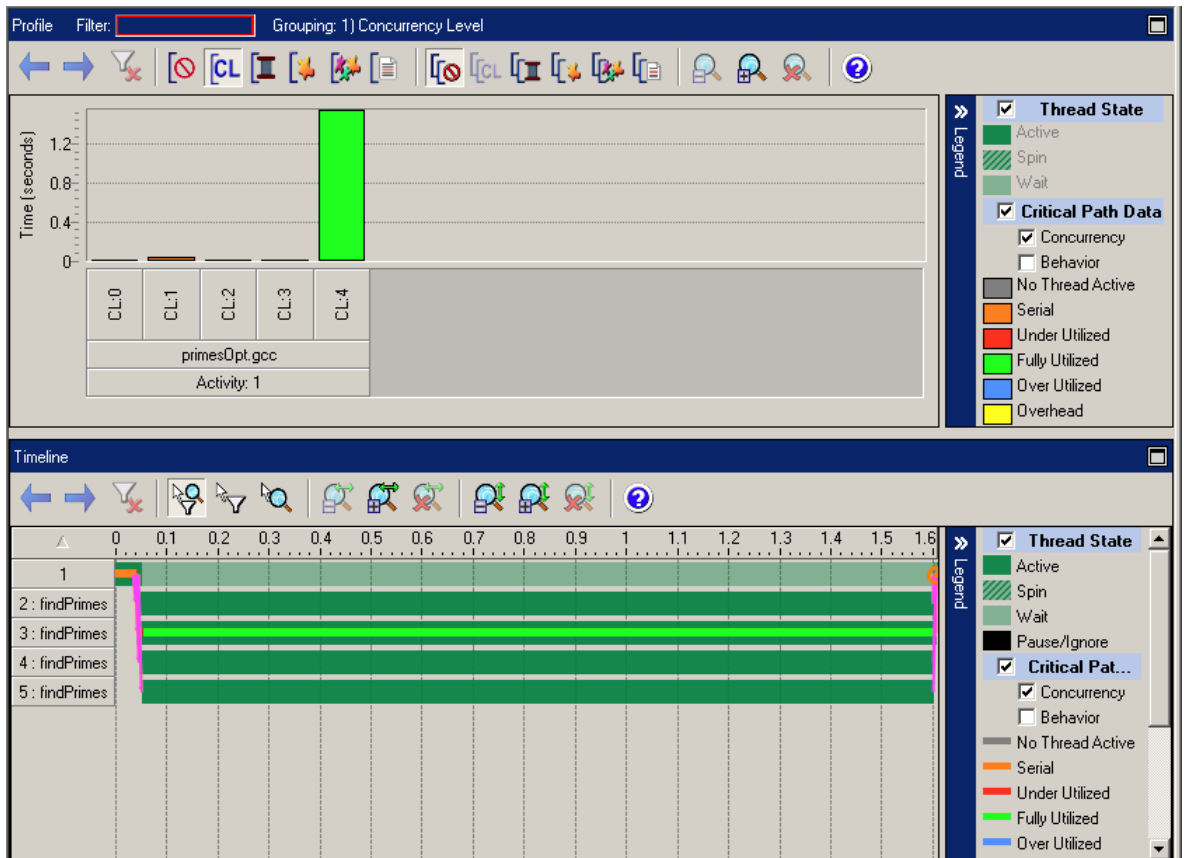


Figure 9: Revised code primesOpt.gcc. The added transition overhead has been eliminated.

6 Next Steps

When you make any revisions to your code, make sure to use the **Intel® Thread Checker** to verify that your code is free of conditions that could lead to inconsistent results. Find details about Thread Checker and other Intel software development products at: <http://www.intel.com/software/products/>

To get the most out of Thread Profiler, explore the following resources:

- **Command line help** is available by running:
`tprofile_cl -help`
Options not covered in this introductory document are described.
- **Online Help** is the Windows* product's complete user's guide. Use **Help** to learn about features not mentioned in this Guide. Open **Help** by pressing the **F1** key or clicking the **Help** buttons in the **Profile** or **Timeline** views.



- **Samples.** Explore additional code examples in `tprofile/samples`. Use them to learn to identify and resolve other types of threading issues.
- **Release Notes.** See `tprofile/doc/Release_Notes.txt` for updated information on requirements, technical support, and known limitations.
- **FAQ.** See `tprofile/doc/tprofileFAQ.htm` for additional information on procedures for driving Thread Profiler for Linux* directly from a Windows* installation.