

Oracle® Developer Studio 12.6: Performance Analyzer Tutorials

ORACLE®

Part No: E77799
June 2017

Part No: E77799

Copyright © 2015, 2017, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Référence: E77799

Copyright © 2015, 2017, Oracle et/ou ses affiliés. Tous droits réservés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf stipulation expresse de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, accorder de licence, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est livré sous licence au Gouvernement des Etats-Unis, ou à quiconque qui aurait souscrit la licence de ce logiciel pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique :

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer un risque de dommages corporels. Si vous utilisez ce logiciel ou ce matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour des applications dangereuses.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée de The Open Group.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers, sauf mention contraire stipulée dans un contrat entre vous et Oracle. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation, sauf mention contraire stipulée dans un contrat entre vous et Oracle.

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> ou le site <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> si vous êtes malentendant.

Contents

Using This Documentation	7
Introduction to the Performance Analyzer Tutorials	9
About the Performance Analyzer Tutorials	9
Getting the Sample Code for the Tutorials	10
Setting Up Your Environment for the Tutorials	11
Introduction to C Profiling	13
About the C Profiling Tutorial	13
Setting Up the lowfruit Sample Code	14
Using Performance Analyzer to Collect Data	15
Using the Performance Analyzer to Examine the lowfruit Data	19
Introduction to Java Profiling	31
About the Java Profiling Tutorial	31
Setting Up the jlowfruit Sample Code	32
Using Performance Analyzer to Collect Data from jlowfruit	33
Using Performance Analyzer to Examine the jlowfruit Data	36
Java and Mixed Java-C++ Profiling	49
About the Java-C++ Profiling Tutorial	49
Setting Up the jsynprog Sample Code	50
Collecting the Data From jsynprog	51
Examining the jsynprog Data	52
Examining Mixed Java and C++ Code	56
Understanding the JVM Behavior	60
Understanding the Java Garbage Collector Behavior	64

Understanding the Java HotSpot Compiler Behavior	70
Hardware Counter Profiling on a Multithreaded Program	75
About the Hardware Counter Profiling Tutorial	75
Setting Up the mttest Sample Code	76
Collecting Data From mttest for Hardware Counter Profiling Tutorial	77
Examining the Hardware Counter Profiling Experiment for mttest	78
Exploring Clock-Profiling Data	80
Understanding Hardware Counter Instruction Profiling Metrics	82
Understanding Hardware Counter CPU Cycles Profiling Metrics	84
Understanding Cache Contention and Cache Profiling Metrics	86
Detecting False Sharing	90
Synchronization Tracing on a Multithreaded Program	95
About the Synchronization Tracing Tutorial	95
About the mttest Program	96
About Synchronization Tracing	96
Setting Up the mttest Sample Code	97
Collecting Data from mttest for Synchronization Tracing Tutorial	98
Examining the Synchronization Tracing Experiment for mttest	98
Understanding Synchronization Tracing	100
Comparing Two Experiments with Synchronization Tracing	105
Exploring More in Performance Analyzer	111
Using the Remote Performance Analyzer	111
More Information	112

Using This Documentation

- **Overview** –Provides step-by-step instructions for using the Oracle Developer Studio 12.6 Performance Analyzer on sample programs.
- **Audience** – Application developers, developer, architect, support engineer
- **Required knowledge** – Programming experience, program/software development testing, aptitude to build and compile software products

Product Documentation Library

Documentation and resources for this product and related products are available at http://docs.oracle.com/cd/E60778_01.

Feedback

Provide feedback about this documentation at <http://www.oracle.com/goto/docfeedback>.

Introduction to the Performance Analyzer Tutorials

Performance Analyzer is the Oracle Developer Studio tool for examining performance of your Java, C, C++, and Fortran applications. You can use it to understand how well your application is performing and find problem areas. These tutorials show how to use Performance Analyzer on sample programs using step-by-step instructions.

About the Performance Analyzer Tutorials

This document features several tutorials that show how you can use Performance Analyzer to profile various types of programs. Each tutorial provides steps for using Performance Analyzer with the source files including screen shots at most steps in the tutorial.

The source code for all the tutorials is included in a single distribution. See [“Getting the Sample Code for the Tutorials” on page 10](#) for information about obtaining the sample source code.

The tutorials include the following:

- [“Introduction to C Profiling”](#)

This introductory tutorial uses a target code named `lowfruit`, written in C. The `lowfruit` program is very simple and includes code for two programming tasks which are each implemented in an efficient way and an inefficient way. The tutorial shows how to collect a performance experiment on the C target program and how to use the various data views in Performance Analyzer. You examine the two implementations of each task and see how Performance Analyzer shows which task is efficient and which is not.

- [“Introduction to Java Profiling”](#)

This introductory tutorial uses a target code named `jlowfruit`, written in Java. Similar to the code used in the C profiling tutorial, the `jlowfruit` program is very simple and includes code for two programming tasks which are each implemented in an efficient way and an inefficient way. The tutorial shows how to collect a performance experiment on the Java target and how to use the various data views in Performance Analyzer. You examine the two implementations of each task, and see how Performance Analyzer shows which task is efficient and which is not.

- [“Java and Mixed Java-C++ Profiling”](#)

This tutorial is based on a Java code named `jsynprog` that performs a number of programming operations one after another. Some operations do arithmetic, one triggers garbage collection, and several use a dynamically loaded C++ shared object, and call from Java to native code and back again. In this tutorial you see how the various operations are implemented, and how Performance Analyzer shows you the performance data about the program.

- [“Hardware Counter Profiling on a Multithreaded Program”](#)

This tutorial is based on a multithreaded program named `mttest` that runs a number of tasks, spawning threads for each one, and uses different synchronization techniques for each task. In this tutorial, you see the performance differences between the computations in the tasks, and use hardware counter profiling to examine and understand an unexpected performance difference between two functions.

- [“Synchronization Tracing on a Multithreaded Program”](#)

This tutorial is also based on the multithreaded program named `mttest` that runs a number of tasks, spawning threads for each one, and uses different synchronization techniques for each task. In this tutorial, you examine the performance differences between the synchronization techniques.

Getting the Sample Code for the Tutorials

The programs used in the Performance Analyzer tutorials are included in a distribution that includes code used for all the Oracle Developer Studio tools. Use the following instructions to obtain the sample code if you have not previously downloaded it.

1. Go to the Oracle Developer Studio 12.6 Sample Applications page at the Oracle Developer Studio web page <http://www.oracle.com/technetwork/server-storage/developerstudio>.
2. Navigate to the downloads section of the Oracle Developer Studio web page.
3. Select the Sample Applications link under the "Other Downloads for Current Release" subsection.
4. Read the license from the link on the page and accept by selecting Accept.
5. Download the zip file by clicking its link and unzip using instructions on the download page.

After you download and unpack the sample files, you can find the samples in the `OracleDeveloperStudio12.6-Samples/PerformanceAnalyzer` directory.

Setting Up Your Environment for the Tutorials

Before you try the tutorials, make sure that you have the Oracle Developer Studio bin directory on your path and have an appropriate Java version in your path as described in Step 1 of [“How to Test Your Installation”](#) in *Oracle Developer Studio 12.6: Installation Guide*.

The make or gmake command must also be on your path so you can build the programs.

Note - The sample code for this tutorial was run on an Oracle Solaris system. The images in this tutorial might differ from what you see on your own environment.

Introduction to C Profiling

This chapter covers the following topics.

- [“About the C Profiling Tutorial” on page 13](#)
- [“Setting Up the `lowfruit` Sample Code” on page 14](#)
- [“Using Performance Analyzer to Collect Data” on page 15](#)
- [“Using the Performance Analyzer to Examine the `lowfruit` Data” on page 19](#)

About the C Profiling Tutorial

This tutorial shows the simplest example of profiling with Oracle Developer Studio Performance Analyzer and demonstrates how to use Performance Analyzer to collect and examine a performance experiment. You use the Overview, Functions view, Source view, and Timeline in this tutorial.

The program `lowfruit` is a simple program that executes two different tasks, one for initializing in a loop and one for inserting numbers into an ordered list. Each task is performed twice, in an inefficient way and in a more efficient way.

Tip - The [“Introduction to Java Profiling”](#) tutorial uses an equivalent Java program and shows similar activities with Performance Analyzer.

The data you see in the experiment that you record will be different from that shown here. The experiment used for the screen-shots in the tutorial was recorded on a SPARC T5 system running Oracle Solaris 11.3. The data from an x86 system running Oracle Solaris or Linux will be different. Furthermore, data collection is statistical in nature and varies from experiment to experiment, even when run on the same system and OS.

The Performance Analyzer window configuration that you see might not precisely match the screen shots. Performance Analyzer enables you to drag separator bars between components of the window, collapse components, and resize the window. Performance Analyzer records its configuration and uses the same configuration the next time it runs. Many configuration changes were made in the course of capturing the screen shots shown in the tutorial.

This tutorial is run locally on a system where Oracle Developer Studio is installed. You can also run remotely as described in [“Using the Remote Performance Analyzer” on page 111](#).

Setting Up the lowfruit Sample Code

Before You Begin:

See the following information about obtaining the code and setting up your environment.

- [“Getting the Sample Code for the Tutorials” on page 10](#)
 - [“Setting Up Your Environment for the Tutorials” on page 11](#)
1. Copy the contents of the lowfruit directory to your own private working area with the following command:

```
% cp -r OracleDeveloperStudio12.6-Samples/PerformanceAnalyzer/lowfruit directory
```

where *directory* is the working directory you are using.

2. Change to that working directory.

```
% cd directory/lowfruit
```

3. Build the target executable.

```
% make clobber
```

```
% make
```

Note - The `clobber` subcommand is only needed if you ran `make` in the directory before, but safe to use in any case.

After you run `make` the directory contains the target program to be used in the tutorial, an executable named `lowfruit`.

The next section shows how to use Performance Analyzer to collect data from the `lowfruit` program and create an experiment.

Tip - If you prefer, you can edit the `Makefile` to do any of the following: use the GNU compilers rather than the default of the Oracle Developer Studio compilers; build in 32-bits rather than the default of 64-bits; and add different compiler flags.

Using Performance Analyzer to Collect Data

This section describes how to use the Profile Application feature of Performance Analyzer to collect data in an experiment.

Tip - If you prefer not to follow these steps to see how to profile applications, you can record an experiment with a make target included in the Makefile for `lowfruit`:

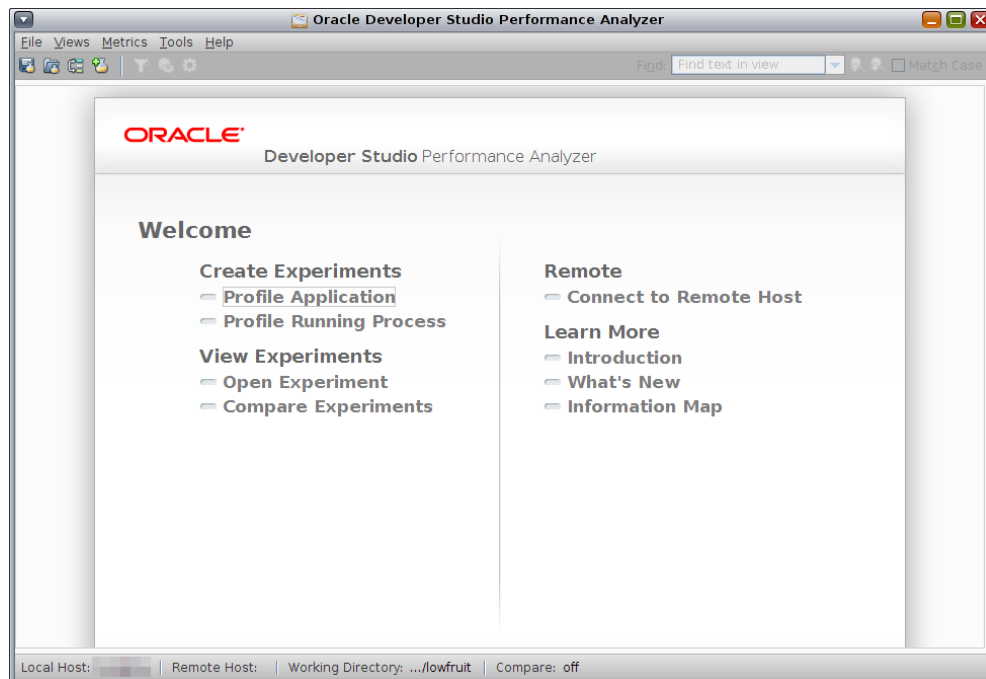
```
make collect
```

The `collect` target launches a `collect` command and records an experiment just like the one that you create using Performance Analyzer in this section. You could then skip to [“Using the Performance Analyzer to Examine the `lowfruit` Data” on page 19.](#)

1. While still in the `lowfruit` directory start, Performance Analyzer:

```
% analyzer
```

Performance Analyzer starts and displays the Welcome page.

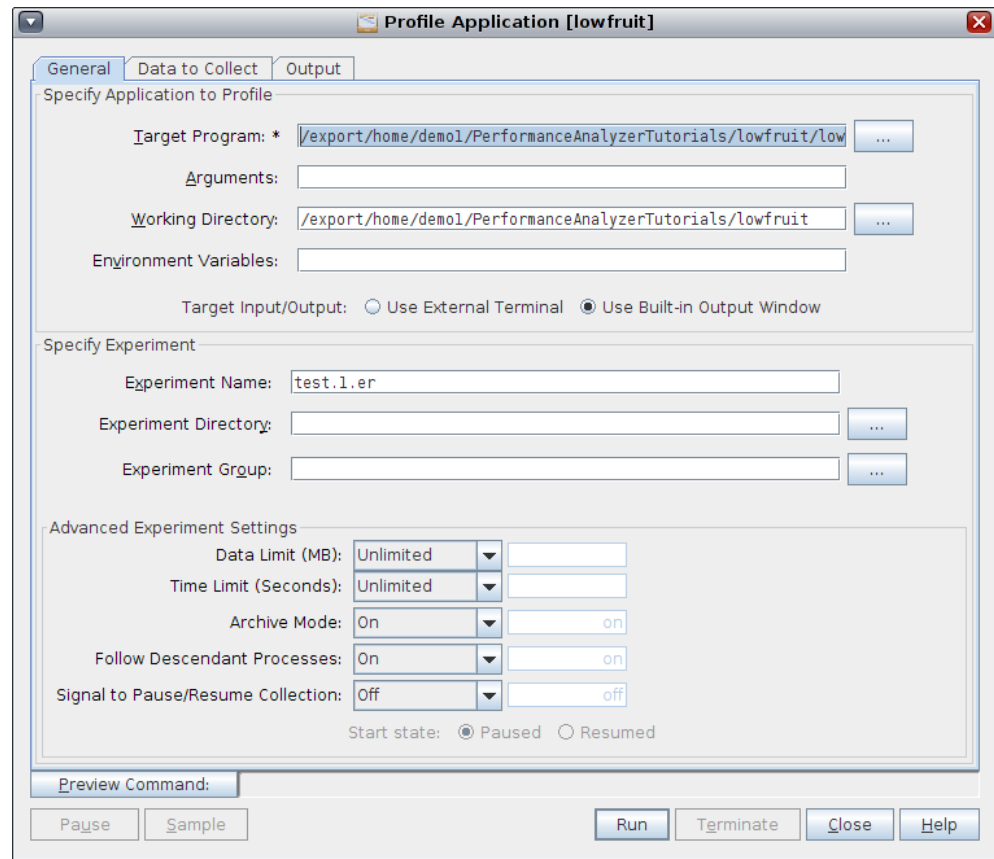


If this is the first time you have used Performance Analyzer, no recent experiments are shown below the Open Experiment item. If you have used it before, you see a list of the experiments you recently opened from the system where you are currently running Performance Analyzer.

2. Click the Profile Application link under Create Experiments in the Welcome page.

The Profile Application dialog box opens with the General tab selected. On this page options are organized into several areas: Specify Application to Profile, Specify Experiment, and Advanced Experiment Settings.

3. In the Target Program field, type the program name lowfruit.



Tip - You could start Performance Analyzer and open this dialog box directly with the program name already entered by specifying the target name when starting Performance Analyzer with the command `analyzer lowfruit`. This method only works when running Performance Analyzer locally.

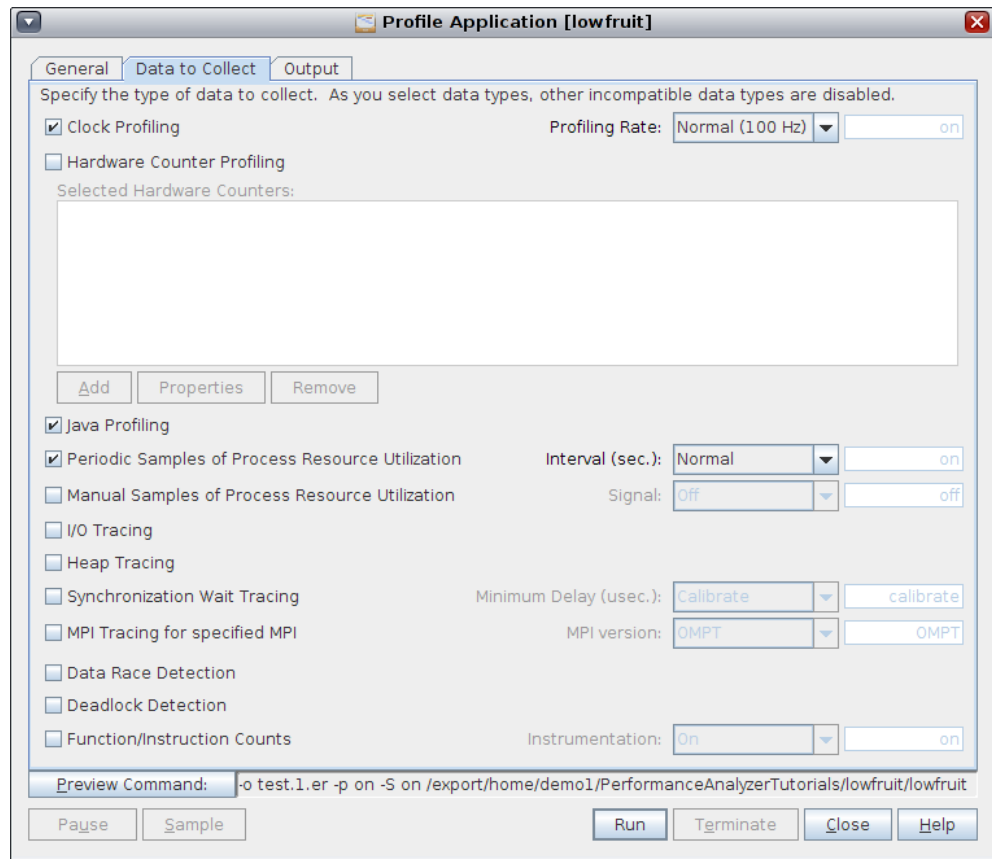
4. For the Target Input/Output option located at the bottom of the Specify Application to Profile panel, select Use Built-in Output Window.

Target Input/Output option specifies the window to which the target program `stdout` and `stderr` will be redirected. The default value is Use External Terminal, but in this tutorial the Target Input/Output option was changed to Use Built-in Output Window to keep all the activity in the Performance Analyzer window. With this option the `stdout` and `stderr` is shown in the Output tab in the Profile Application dialog box.

If you are running remotely, the Target Input/Output option is absent because only the built-in output window is supported.

5. For the Experiment Name option, the default experiment name is `test.1.er` but you can change it to a different name as long as the name ends in `.er`, and is not already in use.
6. Click the Data to Collect tab.

The Data to Collect enables you to select the type of data to collect, and shows the defaults already selected.



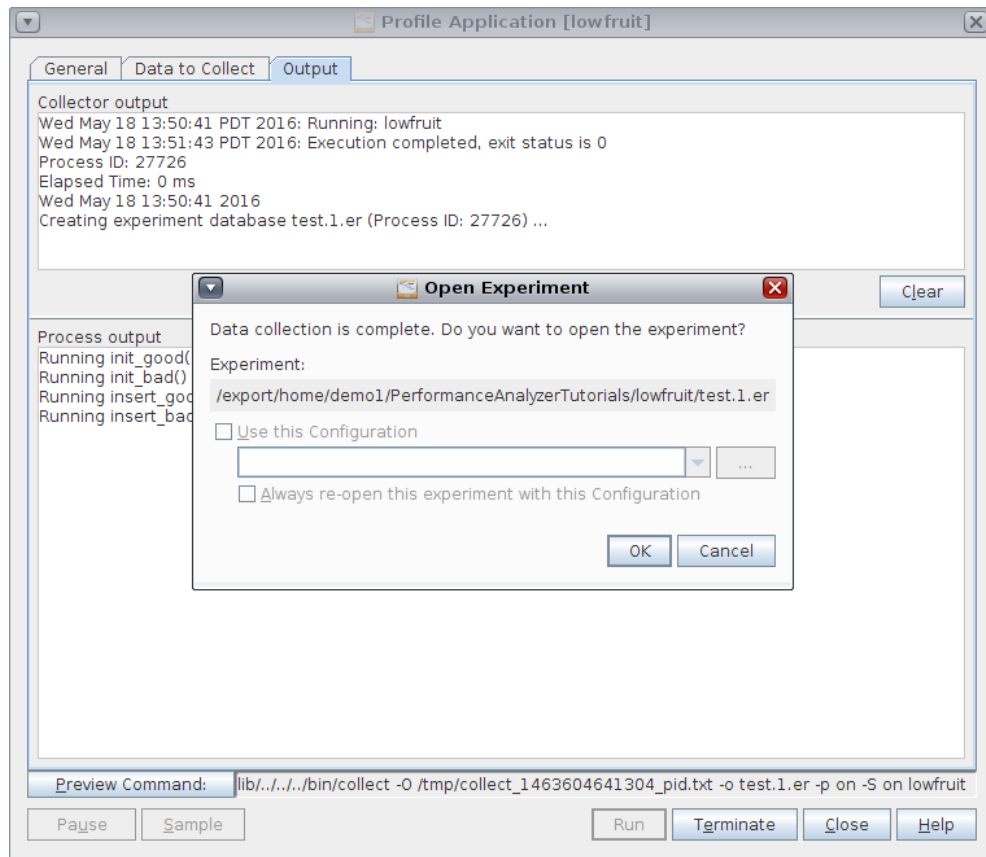
Java profiling is enabled by default as you can see in the screen shot, but it is ignored for a non-Java target such as lowfruit.

You can optionally click the Preview Command button and see the collect command that will be run when you start profiling.

7. Click the Run button.

The Profile Application dialog box displays the Output tab and shows the program output as it runs in the Process Output panel.

After the program completes, a dialog box asks if you want to open the experiment just recorded.



8. Click OK in the dialog box.

The experiment opens. The next section shows how to examine the data.

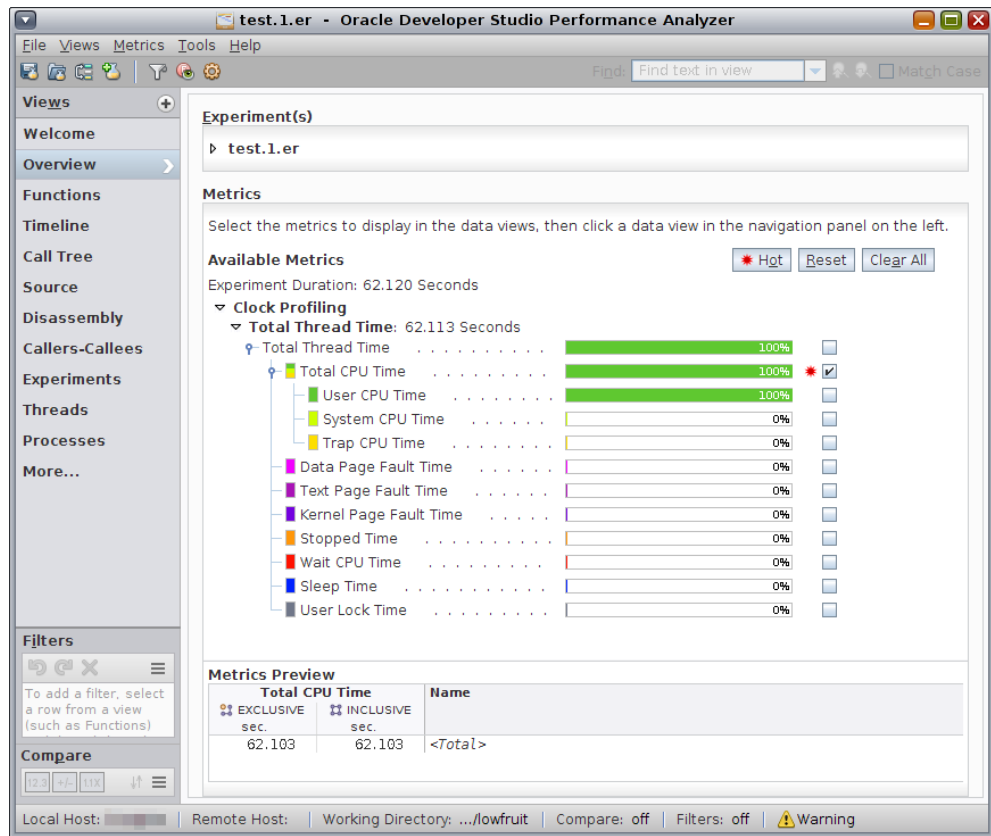
Using the Performance Analyzer to Examine the lowfruit Data

This section shows how to explore the data in the experiment created from the lowfruit sample code.

1. If the experiment you created in the previous section is not already open, you can start Performance Analyzer from the lowfruit directory and load the experiment as follows:

```
% analyzer test.1.er
```

When the experiment opens, Performance Analyzer shows the Overview screen.



In this experiment the Overview shows essentially 100% User CPU time. The program is single-threaded and that one thread is CPU-bound. The experiment was recorded on an Oracle Solaris system, and the Overview shows twelve metrics recorded but only Total CPU Time is enabled by default.

The metrics with colored indicators are the times spent in the ten microstates defined by Oracle Solaris. These metrics include User CPU Time, System CPU Time, and Trap CPU

Time which together are equal to Total CPU Time, as well as various wait times. Total Thread Time is the sum over all of the microstates.

On a Linux machine, only Total CPU Time is recorded because Linux does not support microstate accounting.

By default, both Inclusive and Exclusive Total CPU Time are previewed. *Inclusive* for any metric refers to the metric value in that function or method, including metrics accumulated in all the functions or methods that it calls. *Exclusive* refers only to the metric accumulated within that function or method.

- Click on the Functions view in the Views navigation bar on the left side, or select it using Views → Functions from the menu bar.

The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window displays a table of functions with columns for Exclusive and Inclusive Total CPU Time. The 'init_staticRoutine' function is selected and highlighted. The 'Called-by / Calls' view shows that 'init_staticRoutine' is called by 'init_good' and 'init_bad'. The 'Selection Details' view on the right provides metadata for the selected function, including its name, PC address, size, source file, and various performance metrics like Total Thread Time, Total CPU Time, User CPU Time, System CPU Time, Trap CPU Time, Data Page Fault Time, Text Page Fault Time, Kernel Page Fault Time, Stopped Time, Wait CPU Time, Sleep Time, and User Lock Time.

	Exclusive	Inclusive
Total Thread Time:	39.808 (64.09%)	39.808 (64.09%)
Total CPU Time:	39.798 (64.08%)	39.798 (64.08%)
User CPU Time:	39.798 (64.08%)	39.798 (64.08%)
System CPU Time:	0. (0. %)	0. (0. %)
Trap CPU Time:	0. (0. %)	0. (0. %)
Data Page Fault Time:	0. (0. %)	0. (0. %)
Text Page Fault Time:	0. (0. %)	0. (0. %)
Kernel Page Fault Time:	0. (0. %)	0. (0. %)
Stopped Time:	0. (0. %)	0. (0. %)
Wait CPU Time:	0.010 (100.00%)	0.010 (100.00%)
Sleep Time:	0. (0. %)	0. (0. %)
User Lock Time:	0. (0. %)	0. (0. %)

The Functions view shows the list of functions in the application, with performance metrics for each function. The list is initially sorted by the Exclusive Total CPU Time spent in each function. The list includes all functions from the target application and any shared objects the program uses. The top-most function, the most expensive one, is selected by default.

The Selection Details window on the right shows all the recorded metrics for the selected function.

The Called-by/Calls panel below the functions list provides more information about the selected function and is split into two lists. The Called-by list shows the callers of the selected function and the metric values show the attribution of the total metric for the function to its callers. The Calls list shows the callees of the selected function and shows how the Inclusive metric of its callees contributed to the total metric of the selected function. If you double-click a function in either list in the Called-by/Calls panel, the function becomes the selected function in the main Functions view.

3. Experiment with selecting the various functions to see how the windows in the Functions view update with the changing selection.

The Selection Details window shows you that most of the functions come from the `lowfruit` executable as indicated in the Load Object field.

You can also experiment with clicking on the column headers to change the sort from Exclusive Total CPU Time to Inclusive Total CPU Time, or by Name.

4. In the Functions view compare the two versions of the initialization task, `init_bad()` and `init_good()`.

You can see that the two functions have roughly the same Exclusive Total CPU Time but very different Inclusive times. The `init_bad()` function is slower due to time it spends in a callee. Both functions call the same callee, but they spend very different amounts of time in that routine. You can see why by examining the source of the two routines.

5. Select the function `init_good()` and then click the Source view or choose Views → Source from the menu bar.
6. Adjust the window to allow more space for the code: Collapse the Called-by/Calls panel by clicking the down arrow in the upper margin, and collapse the Selection Details panel by clicking the right-arrow in the side margin.

Note - You might have to re-expand and re-collapse these panels as needed for the rest of the tutorial.

You should scroll up a little to see the source for both `init_bad()` and `init_good()`. The Source view should look similar to the following screen shot.

Function	Total CPU Time (sec)
init_bad(int imax)	36.185
init_good(int imax)	3.613

```

45. init_bad(int imax)
46. {
47.     int i,j,k;
48.     volatile double x;
49.
50.     for(i = 0; i < imax; i++) {
51.         init_static_routine(); /* inside loop */
52.         for(j= 0; j < 50; j++) {
53.             x = 0.0;
54.             for(k=0; k<1000000; k++) {
55.                 x = x + 1.0;
56.             }
57.         }
58.     }
59. }
60.
61. void
62. init_good(int imax)
63. {
64.     int i,j,k;
65.     volatile double x;
66.
67.     init_static_routine(); /* outside loop */
68.     for(i = 0; i < imax; i++) {
69.         for(j= 0; j < 50; j++) {
70.             x = 0.0;
71.             for(k=0; k<1000000; k++) {
72.                 x = x + 1.0;
73.             }
74.         }
75.     }
76. }
77.
78. void

```

Notice that the call to `init_static_routine()` is outside of the loop in `init_good()`, while `init_bad()` has the call to `init_static_routine()` inside the loop. The bad version takes about ten times longer (corresponding to the loop count) than in the good version.

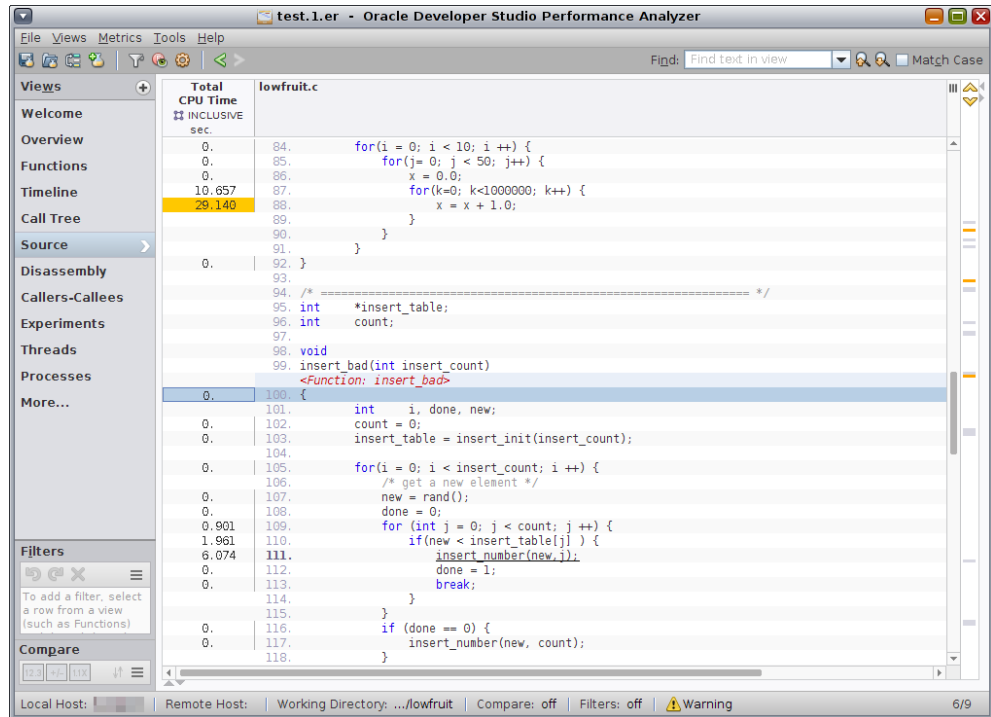
This example is not as silly as it might appear. It is based on a real code that produces a table with an icon for each table row. While it is easy to see that the initialization should not be inside the loop in this example, in the real code the initialization was embedded in a library routine and was not obvious.

The toolkit that was used to implement that code had two library calls (APIs) available. The first API added an icon to a table row, and second API added a vector of icons to the entire table. While it is easier to code using the first API, each time an icon was added, the toolkit recomputed the height of all rows in order to set the correct value for the whole table. When the code used the alternative API to add all icons at once, the recomputation of height was done only once.

- Now go back to the Functions view and look at the two versions of the insert task, `insert_bad()` and `insert_good()`.

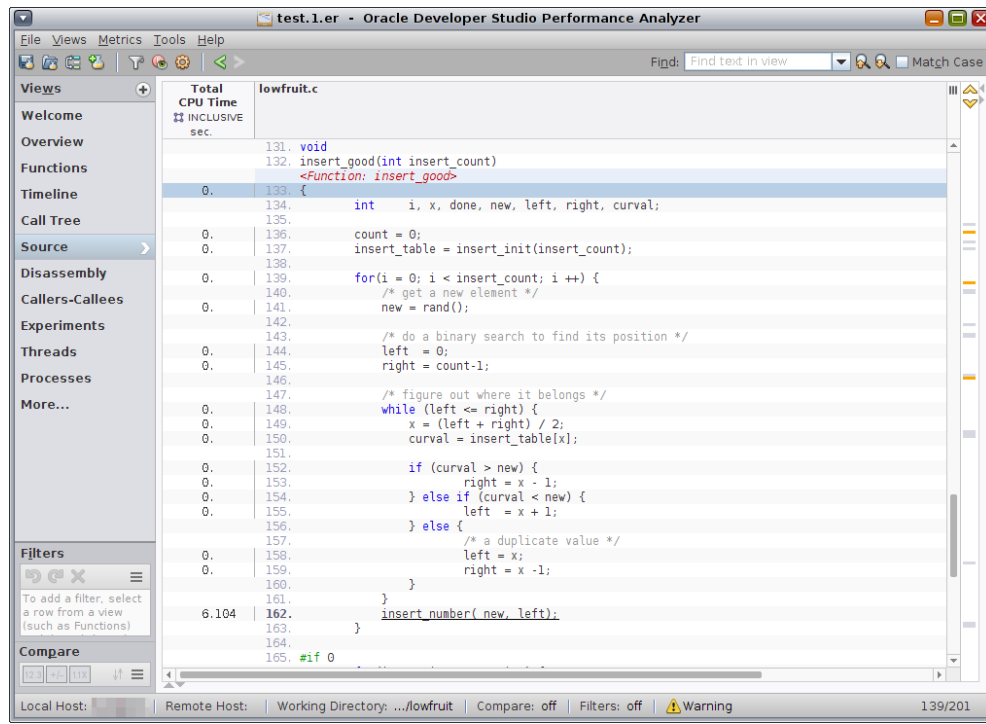
Note that the Exclusive Total CPU time is significant for `insert_bad()`, but negligible for `insert_good()`. The difference between Inclusive and Exclusive time for each version, representing the time in the function `insert_number()` called to insert each entry into the list, is the same. You can see why by examining the source.

8. Select `insert_bad()` and switch to the Source view:



Notice that the time, excluding the call to `insert_number()`, is spent in a loop looking with a linear search for the right place to insert the new number.

9. Now scroll down to look at `insert_good()`.



Note that the code is more complicated because it is doing a binary search to find the right place to insert, but the total time spent, excluding the call to `insert_number()`, is much less than in `insert_bad()`. This example illustrates that binary search can be more efficient than linear search.


You can also see the differences in the routines graphically in the Timeline view.

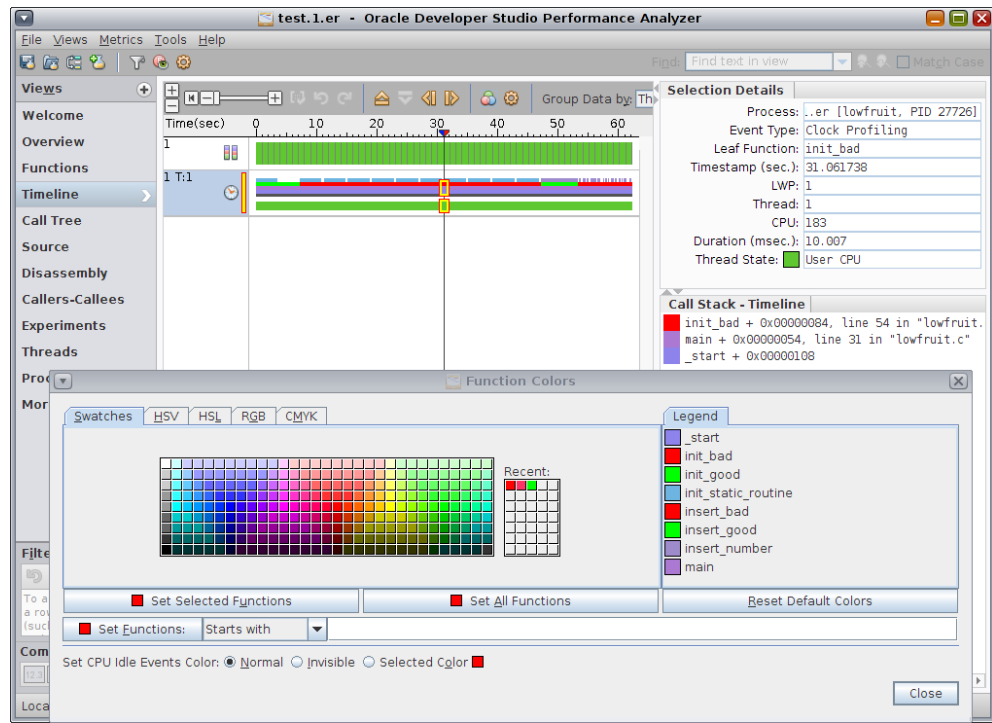
- Click on the Timeline view or choose Views → Timeline from the menu bar.

Tip - If the Selection Details panel is not visible on the right side of the screen, restore it by clicking the small left-arrow in the right margin.

The profiling data is recorded as a series of events, one for every tick of the profiling clock for every thread. The Timeline view shows each individual event with the callstack recorded in that event. The callstack is shown as a list of the frames in the callstack, with the leaf PC (the instruction next to execute at the instant of the event) at the top, and the call

site calling it next, and so forth. For the main thread of the program, the top of the callstack is always `_start`.

11. In the Timeline tool bar, click the Call Stack Function Colors icon  for coloring functions or choose Tools → Function Colors from the menu bar and see the dialog box as shown below.



The function colors were changed to distinguish the good and bad versions of the functions more clearly for the screen shot. The `init_bad()` and `insert_bad()` functions are both now red and the `init_good()` and `insert_good()` are both bright green.

12. To make your Timeline view look similar, do the following in the Function Colors dialog box:
 - Scroll down the list of methods in the Legend to find the `init_bad()` method.
 - Select the `init_bad()` method, click on a red color square in Swatches, and click Set Selected Functions button.
 - Select the `insert_bad()` method, click on a red color square in Swatches, and click Set Selected Functions button.

- Select the `init_good()` method, click on a green color square in Swatches, and click Set Selected Functions button.
- Select the `insert_good()` method, click on a green color square in Swatches, and click Set Selected Functions button.

13. Look at the top bar of the Timeline.

The top bar of the Timeline is the Process-Wide Resource-Utilization Samples bar, as you can see in the tool tip if you move your mouse cursor over the first column. Each segment of the Process-Wide Resource-Utilization Samples bar represents a one-second interval showing the resource usage of the target during that second of execution.

In this example, all the segments are green because all the intervals were spent accumulating User CPU Time. The Selection Details window shows the mapping of colors to microstate although it is not visible in the screen shot.

14. Look at the second bar of the Timeline.

The second bar is the Clock Profiling Call Stacks bar, labeled "1 T:1" which means Process 1 and Thread 1, the only thread in the example. The Clock Profiling Call Stacks bar shows color-coded representations of the callstack. For applications profiled on Oracle Solaris, an additional bar is placed just below the callstack which shows the thread state for each event. In this example, the thread state was always User CPU, so the bar shows a solid green line.

If you click anywhere within that Clock Profiling Call Stacks bar you select the nearest event and the details for that event are shown in the Selection Details window. From the pattern of the call stacks, you can see that the time in the `init_good()` and `insert_good()` routines shown in bright green in the screen shot is considerably shorter than the corresponding time in the `init_bad()` and `insert_bad()` routines shown in red.

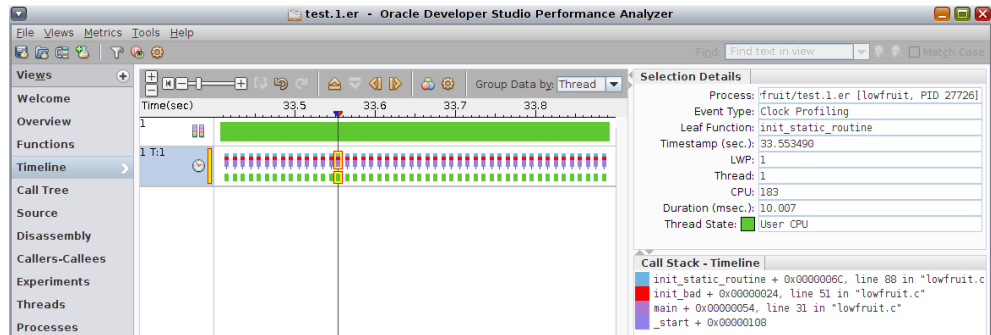
15. Select events in the regions corresponding to the good and bad routines in the timeline and look at the call stacks in the Call Stack - Timeline window below the Selection Details window.

You can select any frame in the Call Stack window, and then select the Source view on the Views navigation bar, and go to the source for that source line. You can also double-click a frame in a call stack to go to the Source view or right-click the frame in the call stack and select from a pop-up menu.

16. Zoom in on the events by using one of the following methods:

- Double-click on the area of interest.
- Drag the cursor in the ruler to adjust vertical time markers, then press Enter.
- Use the + or - icons in the toolbar.
- Press the plus (+) and minus (-) keys to further adjust the zoom.

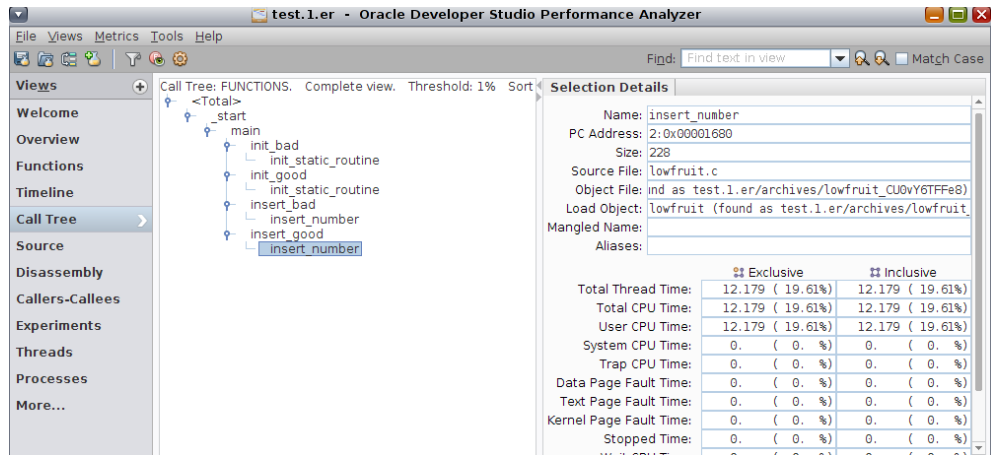
If you zoom in enough you can see that the data shown is not continuous but consists of discrete events, one for each profile tick, which is about 10 ms in this example.



Press the F1 key to see the Help for more information about the Timeline view.

- Click on the Call Tree view or choose Views → Call Tree to see the structure of your program.

The Call Tree view shows a dynamic call graph of the program, with the Selection Details panel showing performance information.



Performance Analyzer has many additional views of the data, such as the Caller-Callees view which enables you to navigate through the program structure, and the Experiments view which shows you details of the recorded experiment. For this simple example, the Threads and Processes views are not very interesting.

By clicking on the + button on the Views list you can add other views to the navigation bar. If you are an assembly-language programmer, you might want to look at the Disassembly. Try exploring the other views.

Performance Analyzer also has a very powerful filtering capability. You can filter by time, thread, function, source line, instruction, call stack-fragment, and any combination of them. The use of filtering is outside the scope of this tutorial, since the sample code is so simple that filtering is not needed.

Introduction to Java Profiling

This chapter covers the following topics.

- [“About the Java Profiling Tutorial” on page 31](#)
- [“Setting Up the `javafruit` Sample Code” on page 32](#)
- [“Using Performance Analyzer to Collect Data from `javafruit`” on page 33](#)
- [“Using Performance Analyzer to Examine the `javafruit` Data” on page 36](#)

About the Java Profiling Tutorial

This tutorial shows the simplest example of profiling with Oracle Developer Studio Performance Analyzer and demonstrates how to use Performance Analyzer to collect and examine a performance experiment. You use the Overview, Functions view, Source view, Timeline view, and Call Tree view in this tutorial.

The program `javafruit` is a simple program that executes two different tasks, one for initializing in a loop and one for inserting numbers into an ordered list. Each task is performed twice, in an inefficient way and in a more efficient way.

Tip - The [“Introduction to C Profiling”](#) tutorial uses an equivalent C program and shows similar activities with Performance Analyzer.

The data you see in the experiment that you record will be different from that shown here. The experiment used for the screen-shots in the tutorial was recorded on a SPARC T5 system running Oracle Solaris 11.3. The data from an x86 system running Oracle Solaris or Linux will be different. Furthermore, data collection is statistical in nature and varies from experiment to experiment, even when run on the same system and OS.

The Performance Analyzer window configuration that you see might not precisely match the screen shots. Performance Analyzer enables you to drag separator bars between components of the window, collapse components, and resize the window. Performance Analyzer records its configuration and uses the same configuration the next time it runs. Many configuration changes were made in the course of capturing the screen shots shown in the tutorial.

This tutorial is run locally on a system where Oracle Developer Studio is installed. You can also run remotely as described in [“Using the Remote Performance Analyzer” on page 111](#).

Setting Up the jlowfruit Sample Code

Before You Begin:

See the following for information about obtaining the code and setting up your environment.

- [“Getting the Sample Code for the Tutorials” on page 10](#)
- [“Setting Up Your Environment for the Tutorials” on page 11](#)

1. Copy the contents of the jlowfruit directory to your own private working area with the following command:

```
% cp -r OracleDeveloperStudio12.6-Samples/PerformanceAnalyzer/jlowfruit directory
```

where *mydirectory* is the working directory you are using.

2. Change to that working directory copy.

```
% cd directory/jlowfruit
```

3. Build the target executable.

```
% make clobber
```

```
% make
```

Note - The clobber subcommand is only needed if you ran make in the directory before, but safe to use in any case.

After you run make the directory contains the target application to be used in the tutorial, a Java class file named jlowfruit.class.

Tip - If you are having trouble compiling the sample, check your version of javac using the following command:

```
% javac -version
```

If the output does not report at least javac 1.7, then you need to update your PATH to a JDK of 7 or higher.

The next section shows how to use Performance Analyzer to collect data from the jlowfruit program and create an experiment.

Using Performance Analyzer to Collect Data from jlowfruit

This section describes how to use the Profile Application feature of Performance Analyzer to collect data in an experiment on a Java application.

Tip - If you prefer not to follow these steps to see how to profile applications from Performance Analyzer, you can record an experiment with a make target included in the Makefile for jlowfruit:

```
% make collect
```

The collect target launches a collect command and records an experiment just like the one that you create using Performance Analyzer in this section. You could then skip to [“Using Performance Analyzer to Examine the jlowfruit Data”](#) on page 36.

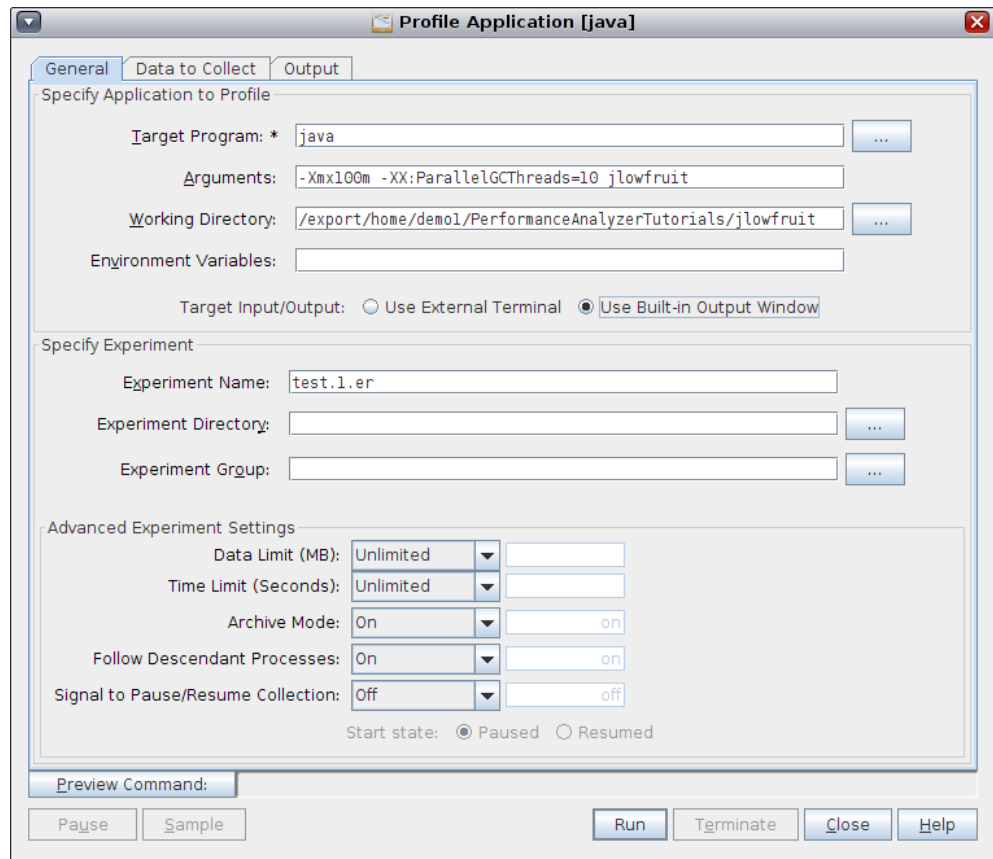
1. While still in the jlowfruit directory start Performance Analyzer with the target java and its arguments:

```
% analyzer java -Xmx100m -XX:ParallelGCThreads=10 jlowfruit
```

The Profile Application dialog box opens with the General tab selected and several options already filled out using information you provided with the analyzer command.

Target Program is set to java and Arguments is set to

```
-Xmx100m -XX:ParallelGCThreads=10 jlowfruit
```



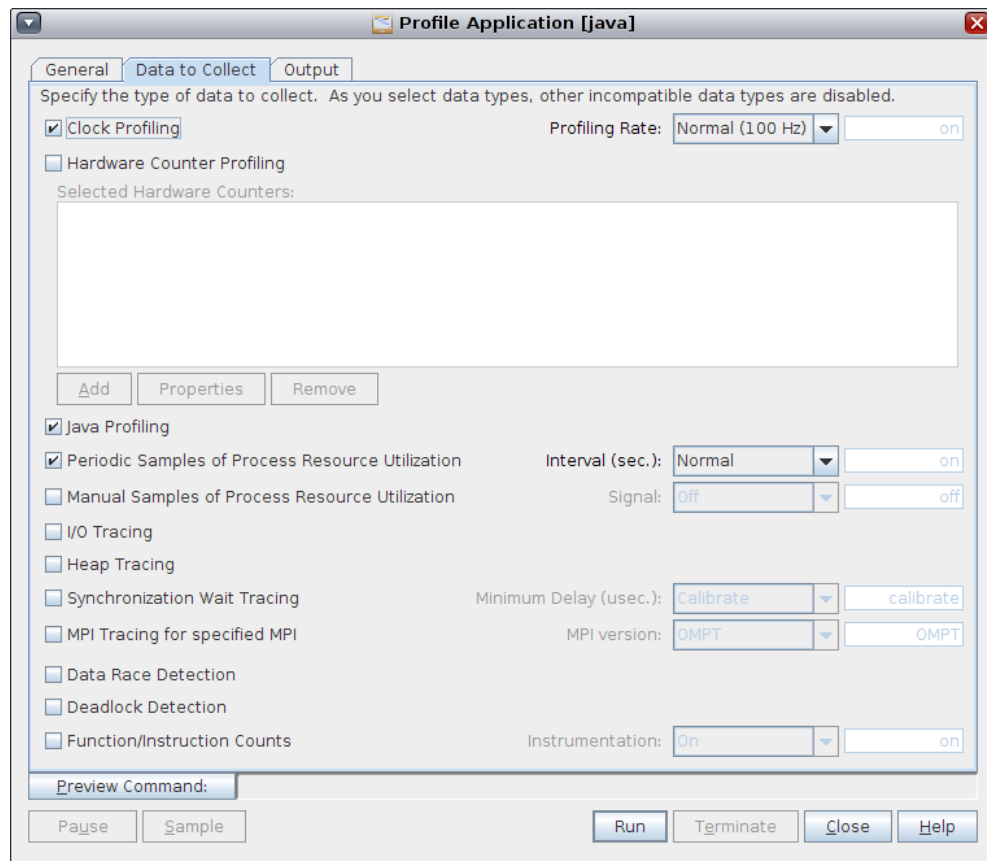
2. For the Target Input/Output option, select Use Built-in Output Window.

Target Input/Output option specifies the window to which the target program `stdout` and `stderr` will be redirected. The default value is Use External Terminal, but in this tutorial you should change the Target Input/Output option to Use Built-in Output Window to keep all the activity in the Performance Analyzer window. With this option the `stdout` and `stderr` is shown in the Output tab in the Profile Application dialog box.

If you are running remotely, the Target Input/Output option is absent because only the built-in output window is supported.

3. For the Experiment Name option, the default experiment name is `test.1.er` but you can change it to a different name as long as the name ends in `.er`, and is not already in use.
4. Click the Data to Collect tab.

The Data to Collect tab enables you to select the type of data to collect, and shows the defaults already selected.



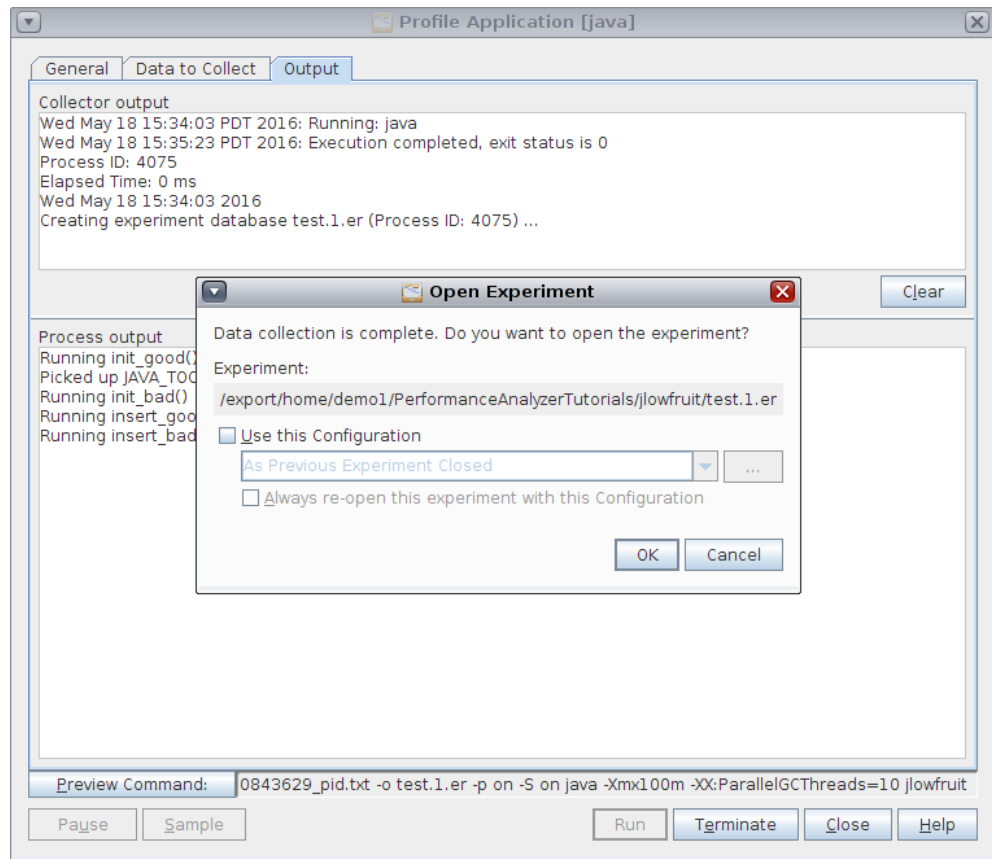
Java profiling is enabled by default as you can see in the screen shot.

You can optionally click the Preview Command button and see the collect command that will be run when you start profiling.

5. Click the Run button.

The Profile Application dialog box displays the Output tab and shows the program output in the Process Output panel as the program runs.

After the program completes, a dialog box asks if you want to open the experiment just recorded.



6. Click OK in the dialog box.

The experiment opens. The next section shows how to examine the data.

Using Performance Analyzer to Examine the jlowfruit Data

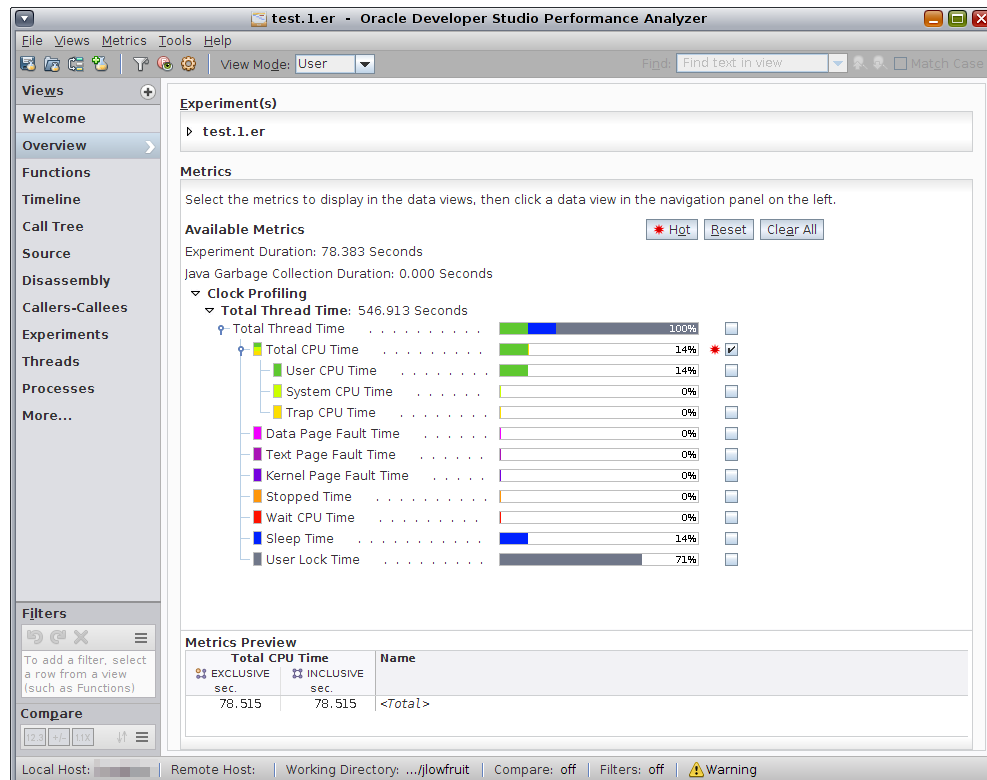
This section shows how to explore the data in the experiment created from the jlowfruit sample code.

1. If the experiment you created in the previous section is not already open, you can start Performance Analyzer from the jlowfruit directory and load the experiment as follows:

% analyzer test.1.er

When the experiment opens, Performance Analyzer shows the Overview page.

2. Notice the Overview page shows a summary of the metric values and enables you to select metrics.



In this experiment the Overview shows about 14% Total CPU Time which was all User CPU Time, plus about 14% Sleep Time and 71% User Lock Time. The user Java code jlowfruit is single-threaded and that one thread is CPU-bound, but all Java programs use multiple threads including a number of system threads. The number of those threads depends on the choice of JVM options, including the Garbage Collector parameters and the size of the machine on which the program was run.

The experiment was recorded on an Oracle Solaris system, and the Overview shows twelve metrics recorded but only Total CPU Time is enabled by default.

The metrics with colored indicators are the times spent in the ten microstates defined by Oracle Solaris. These metrics include User CPU Time, System CPU Time, and Trap CPU Time which together are equal to Total CPU Time, as well as various wait times. Total Thread Time is the sum over all of the microstates.

On a Linux machine, only Total CPU Time is recorded because Linux does not support microstate accounting.

By default, both Inclusive and Exclusive Total CPU Time are previewed. *Inclusive* for any metric refers to the metric value in that function or method, including metrics accumulated in all the functions or methods that it calls. *Exclusive* refers only to the metric accumulated within that function or method.

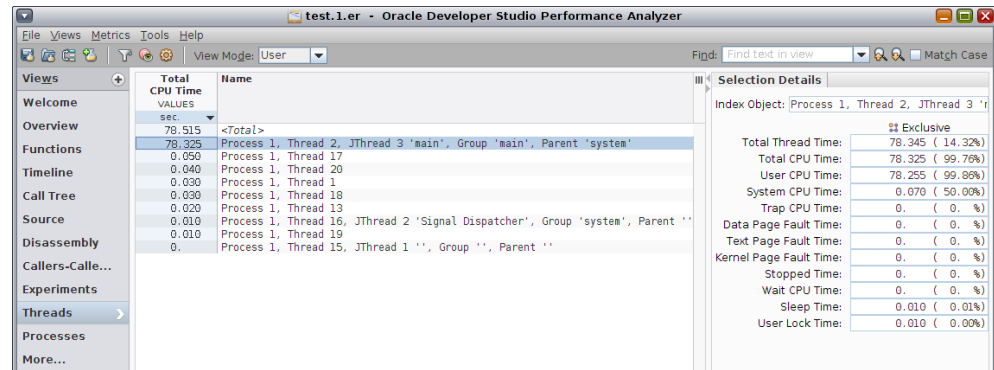
3.



Click the Hot button to select metrics with high values to show them in the data views.

The Metrics Preview panel at the bottom is updated to show you how the metrics will be displayed in the data views that present table-formatted data. You will next look to see which threads are responsible for which metrics.

4. Now switch to the Threads view by clicking its name in the Views navigation panel or choosing Views → Threads from the menu bar.



The thread with almost all of the Total CPU Time is Thread 2, which is the only user Java thread in this simple application.

Thread 15 is most likely a user thread even though it is actually created internally by the JVM. It is only active during start-up and has very little time accumulated. In your experiment, a second thread similar to thread 15 might be created.

Thread 1 spends its entire time sleeping.

The remaining threads spend their time waiting for a lock, which is how the JVM synchronizes itself internally. Those threads include those used for HotSpot compilation and for Garbage Collection. This tutorial does not explore the behavior of the JVM system, but that is explored in another tutorial, “[Java and Mixed Java-C++ Profiling](#)”.

- Click on the Functions view in the Views navigation panel, or choose Views → Functions from the menu bar.

The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window is titled 'test.1.er - Oracle Developer Studio Performance Analyzer'. The 'Views' panel on the left has 'Functions' selected. The main area displays a table of functions with columns for 'EXCLUSIVE' and 'INCLUSIVE' CPU Time in seconds. The top function is '<Total>' with 78,515 seconds of exclusive time. Other functions include 'jlowfruit.init_static_routine()', 'jlowfruit.insert_number(int, int)', 'jlowfruit.insert_bad(int)', 'jlowfruit.init_bad(int)', 'jlowfruit.init_good(int)', '<JVM-System>', 'jlowfruit.insert_good(int)', 'java.util.Random.nextInt()', 'java.util.concurrent.atomic.AtomicLong.compareAndSet(long, long)', 'java.util.Random.next(int)', 'java.net.URLClassLoader.getResources(java.lang.String)', 'java.util.Formatter.access\$000(java.util.Formatter)', 'sun.misc.URLClassPath\$JarLoader\$1.run()', 'Interpreter', 'InterpreterRuntime::new(JavaThread*, constantPoolDesc*, int)', 'InterpreterRuntime::resolve_invoke(JavaThread*, Bytecodes::Code)', 'JVM_MonitorWait', 'JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments)', 'JavaCalls::call_virtual(JavaValue*, Handle, KlassHandle, Symbol*, Symbol*)', 'JavaThread::run()', 'JavaThread::thread_main_inner()', 'Java_java_io_UnixFileSystem.canonicalize0', 'Java_java_lang_ClassLoader.defineClass1', 'Java_java_security_AccessController.doPrivileged_Ljava_security_Java_java_security_AccessController.doPrivileged_Ljava_security_AccessController.doPrivileged_Ljava_security_LinkResolver::resolve_invoke(CallInfo6, Handle, constantPoolHandle, LinkResolver::resolve_invokestatic(CallInfo6, constantPoolHandle, LinkResolver::resolve_static_call(CallInfo6, KlassHandle, Symbol*))

The 'Selection Details' window on the right shows the following metrics for the selected function '<Total>':

	Exclusive	Inclusive
Total Thread Time:	546.913 (100.00%)	5
Total CPU Time:	78.515 (100.00%)	
User CPU Time:	78.365 (100.00%)	
System CPU Time:	0.140 (100.00%)	
Trap CPU Time:	0.010 (100.00%)	
Data Page Fault Time:	0. (0. %)	
Text Page Fault Time:	0. (0. %)	
Kernel Page Fault Time:	0. (0. %)	
Stopped Time:	0. (0. %)	
Wait CPU Time:	0. (0. %)	
Sleep Time:	78.355 (100.00%)	
User Lock Time:	390.043 (100.00%)	3

The 'Called-by / Calls' panel at the bottom shows the callers of the selected function:

Total ... ATTRIBUTED sec	<Total> is called by	<Total> calls
78.175	jlowfruit.main(java.lang.String[])	
0.330	<JVM-System>	
0.010	sun.launcher.LauncherHelper.check	
0	lwn_start	

The Functions view shows the list of functions in the application, with performance metrics for each function. The list is initially sorted by the Exclusive Total CPU Time spent in each function. There are also a number of functions from the JVM in the Functions view, but they have relatively low metrics. The list includes all functions from the target application and any shared objects the program uses. The top-most function, the most expensive one, is selected by default.

The Selection Details window on the right shows all the recorded metrics for the selected function.

The Called-by/Calls panel below the functions list provides more information about the selected function and is split into two lists. The Called-by list shows the callers of the selected function and the metric values show the attribution of the total metric for

the function to its callers. The Calls list shows the callees of the selected function and shows how the Inclusive metric of its callees contributed to the total metric of the selected function. If you double-click a function in either list in the Called-by/Calls panel, the function becomes the selected function in the main Functions view.

6. Experiment with selecting the various functions to see how the Called-by / Calls panel and Selection Details window in the Functions view update with the changing selection.

The Selection Details window shows you that most of the functions come from the `jlowfruit.class` as indicated in the Load Object field.

You can also experiment with clicking on the column headers to change the sort from Exclusive Total CPU Time to Inclusive Total CPU Time, or by Name.

7. In the Functions view compare the two versions of the initialization task, `jlowfruit.init_bad()` and `jlowfruit.init_good()`.

You can see that the two functions have roughly the same Exclusive Total CPU Time but very different Inclusive times. The `jlowfruit.init_bad()` function is slower due to time it spends in a callee. Both functions call the same callee, but they spend very different amounts of time in that routine. You can see why by examining the source of the two routines.

8. Select the function `jlowfruit.init_good()` and then click the Source view or choose Views → Source from the menu bar.
9. Adjust the window to allow more space for the code: Collapse the Called-by/Calls panel by clicking the down arrow in the upper margin, and collapse the Selection Details panel by clicking the right arrow in the side margin.

Note - You might have to re-expand and re-collapse these panels as needed for the rest of the tutorial.

You should scroll up a little to see the source for both `jlowfruit.init_bad()` and `jlowfruit.init_good()`. The Source view should look similar to the following screen shot.

Line	Code	Total CPU Time (sec)
35	/* ----- */	
36	double	
37	init_bad(int imax)	
38	{	
39	int i,j,k;	
40	double x = 0;	0.0
41	/*	
42	for(i = 0; i < imax; i++) {	
43	init_static_routine(); /* inside loop */	30.641
44	for(j = 0; j < 50; j++) {	0.0
45	x = 0.0;	0.0
46	for(k=0; k<1000000; k++) {	0.190
47	x = x + 1.0;	1.341
48	}	
49	}	
50	}	
51	return x;	0.0
52	}	
53		
54	double	
55	init_good(int imax)	
56	{	
57	int i,j,k;	
58	double x = 0;	0.0
59	/*	
60	init_static_routine(); /* outside loop */	3.072
61	for(i = 0; i < imax; i++) {	0.0
62	for(j = 0; j < 50; j++) {	0.0
63	x = 0.0;	0.130
64	for(k=0; k<1000000; k++) {	0.130
65	x = x + 1.0;	1.401
66	}	
67	}	
68	}	
69	return x;	0.0
70	}	
71		
72	double	

Notice that the call to `jlowfruit.init_static_routine()` is outside of the loop in `jlowfruit.init_good()`, while `jlowfruit.init_bad()` has the call to `jlowfruit.init_static_routine()` inside the loop. The bad version takes about ten times longer (corresponding to the loop count) than in the good version.

This example is not as silly as it might appear. It is based on a real code that produces a table with an icon for each table row. While it is easy to see that the initialization should not be inside the loop in this example, in the real code the initialization was embedded in a library routine and was not obvious.

The toolkit that was used to implement that code had two library calls (APIs) available. The first API added an icon to a table row, and second API added a vector of icons to the entire table. While it is easier to code using the first API, each time an icon was added, the toolkit recomputed the height of all rows in order to set the correct value for the whole table. When the code used the alternative API to add all icons at once, the recomputation of height was done only once.

- Now go back to the Functions view and look at the two versions of the insert task, `jlowfruit.insert_bad()` and `jlowfruit.insert_good()`.

Note that the Exclusive Total CPU time is significant for `jlowfruit.insert_bad()`, but negligible for `jlowfruit.insert_good()`. The difference between Inclusive and Exclusive time for each version, representing the time in the function `jlowfruit.insert_number()` called to insert each entry into the list, is the same. You can see why by examining the source.

11. Select `jlowfruit.insert_bad()` and switch to the Source view:

```

test.1.er - Oracle Developer Studio Performance Analyzer
File Views Metrics Tools Help
View Mode: User Find: Find text in view Match Case

Views
Welcome
Overview
Functions
Timeline
Call Tree
Source
Disassembly
Callers-Calle...
Experiments
Threads
Processes
More...

Filters
To add a filter,
select a row from a
view (such as

Compare
12.3 9.2 11K

Total CPU Time
INCLUSIVE
sec.
jlowfruit.java
92.
93. void
94. insert_bad(int insert_count)
95. {
96.     int i, done, newR;
    <Function: jlowfruit.insert_bad(int)>
97.     count = 0;
98.     insert_table = insert_init(insert_count);
99.
100.    for(i = 0; i < insert_count; i++) {
101.        /* get a new element */
102.        newR = rand();
103.        done = 0;
104.        for (int j = 0; j < count; j++) {
105.            if(newR < insert_table[j]) {
106.                insert_number(newR, j);
107.                done = 1;
108.                break;
109.            }
110.        }
111.        if (done == 0) {
112.            insert_number(newR, count);
113.        }
114.    }
115.
116.
117.    // free(insert_table);
118. }
119.
120. void
121. insert_good(int insert_count)
122. {
123.     int i, x, done, newR, left, right, curval;
124.
    <Function: jlowfruit.insert_good(int)>
125.     count = 0;
126.     insert_table = insert_init(insert_count);
127.
128.     for(i = 0; i < insert_count; i++) {
129.         /* get a new element */

```

Local Host: Remote Host: Working Directory: .../jlowfruit Compare: off Filters: off Warning 43/229

Notice that the time, excluding the call to `jlowfruit.insert_number()`, is spent in a loop looking with a linear search for the right place to insert the new number.

12. Now scroll down to look at `jlowfruit.insert_good()`.

The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window displays the source code for `jlowfruit.java`. The 'Total CPU Time' column on the left indicates the execution time for each line of code. The most significant time is spent on the call to `insert_number(newR, left)` at line 151, which takes 14.890 seconds. The code includes a binary search loop from line 136 to 149, which is much faster than a linear search.

Line	Total CPU Time (sec)	Code
121		<code>insert_good(int insert_count)</code>
122		<code>{</code>
123		<code>int i, x, done, newR, left, right, curval;</code>
124		<code><Function: jlowfruit.insert_good(int)></code>
125	0.	<code>count = 0;</code>
126	0.	<code>insert_table = insert_init(insert_count);</code>
127		<code>}</code>
128	0.	<code>for(i = 0; i < insert_count; i++) {</code>
129		<code>/* get a new element */</code>
130	0.030	<code>newR = rand();</code>
131		<code>}</code>
132		<code>/* do a binary search to find its position */</code>
133	0.	<code>left = 0;</code>
134	0.	<code>right = count-1;</code>
135		<code>}</code>
136	0.010	<code>/* figure out where it belongs */</code>
137	0.010	<code>while (left <= right) {</code>
138		<code>x = (left + right) / 2;</code>
139	0.020	<code>curval = insert_table[x];</code>
140		<code>}</code>
141	0.	<code>if (curval > newR) {</code>
142	0.030	<code>right = x - 1;</code>
143	0.	<code>} else if (curval < newR) {</code>
144	0.	<code>left = x + 1;</code>
145		<code>} else {</code>
146		<code>/* a duplicate value */</code>
147	0.	<code>left = x;</code>
148	0.	<code>right = x - 1;</code>
149		<code>}</code>
150		<code>}</code>
151	14.890	<code>insert_number(newR, left);</code>
152		<code>}</code>
153	0.	<code>}</code>
154		<code>}</code>
155		<code>int [] // int *</code>
156		<code>insert_init(int maxcount)</code>
157		<code>{</code>
158		<code>/* reset a random number seed */</code>


Note that the code is more complicated because it is doing a binary search to find the right place to insert, but the total time spent, excluding the call to `jlowfruit.insert_number()`, is much less than in `jlowfruit.insert_bad()`. This example illustrates that binary search can be more efficient than linear search.

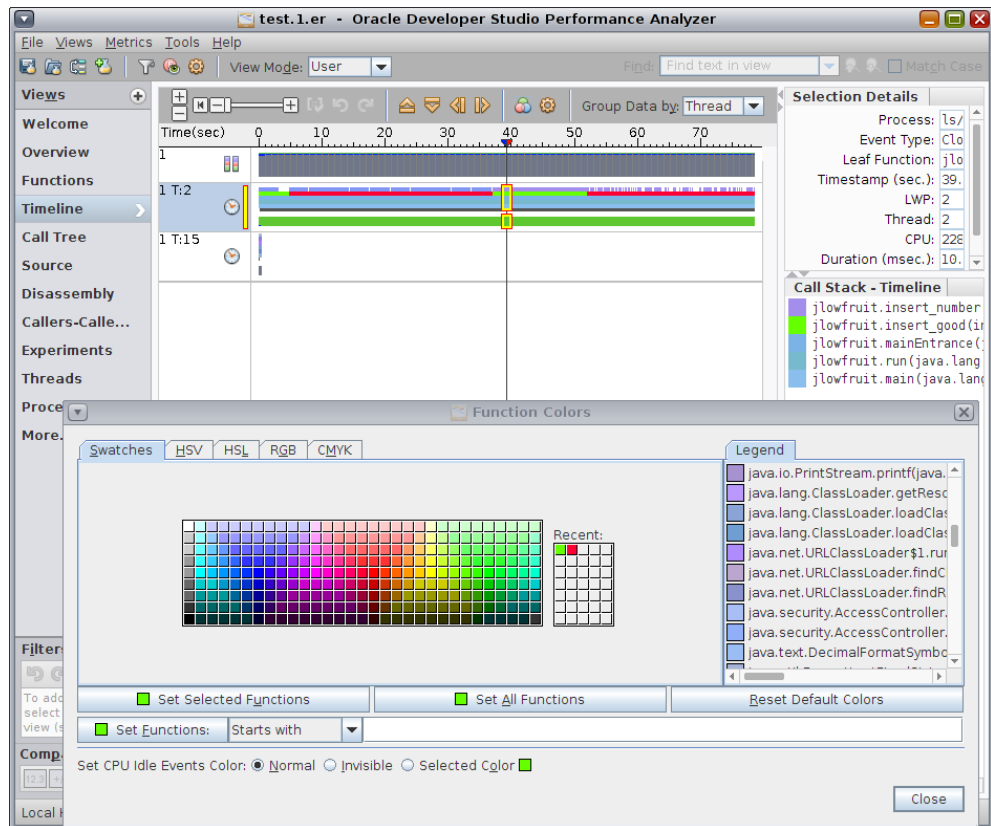
You can also see the differences in the routines graphically in the Timeline view.

- Click on the Timeline view or choose Views → Timeline from the menu bar.

Tip - If the Selection Details panel is not visible on the right side of the screen, restore it by clicking the small left-arrow in the right margin.

The profiling data is recorded as a series of events, one for every tick of the profiling clock for every thread. The Timeline view shows each individual event with the call stack recorded in that event. The call stack is shown as a list of the frames in the callstack, with the leaf PC (the instruction next to execute at the instant of the event) at the top, and the call site calling it next, and so forth. For the main thread of the program, the top of the callstack is always main.

14. In the Timeline tool bar, click the Call Stack Function Colors icon  for coloring functions or choose Tools → Function Colors from the menu bar and see the dialog box as shown below.



The function colors were changed to distinguish the good and bad versions of the functions more clearly for the screen shot. The `jlowfruit.init_bad()` and `jlowfruit.`

`insert_bad()` functions are both now red and the `jlowfruit.init_good()` and `jlowfruit.insert_good()` are both bright green.

15. To make your Timeline view look similar, do the following in the Function Colors dialog box:

- Scroll down the list of java methods in the Legend to find the `jlowfruit.init_bad()` method.
- Select the `jlowfruit.init_bad()` method, click on a red color square in Swatches, and click Set Selected Functions button.
- Select the `jlowfruit.insert_bad()` method, click on a red color square in Swatches, and click Set Selected Functions button.
- Select the `jlowfruit.init_good()` method, click on a green color square in Swatches, and click Set Selected Functions button.
- Select the `jlowfruit.insert_good()` method, click on a green color square in Swatches, and click Set Selected Functions button.

16. Look at the top bar of the Timeline.

The top bar of the Timeline is the Process-Wide Resource-Utilization Samples bar as you can see in the tool tip if you move your cursor over the first column. Each segment of the Process-Wide Resource-Utilization Samples bar represents a one-second interval showing the resource usage of the target during that second of execution.

In this example, the segments are mostly gray with some green, reflecting the fact that only a small fraction of the Total Time was spent accumulating User CPU Time. The Selection Details window shows the mapping of colors to microstate although it is not visible in the screen shot.

17. Look at the second bar of the Timeline.

The second bar is the Clock Profiling Call Stacks bar, labeled "1 T:2" which means Process 1 and Thread 2, the main user thread in the example. The Clock Profiling Call Stacks bar shows color-coded representations of the callstack. For applications profiled on Oracle Solaris, an additional bar is placed just below the callstack which shows the thread state for each event. In this example, the thread state was always User CPU, so the bar shows a solid green line.

You should see one or two additional bars labeled with different thread numbers but they will only have a few events at the beginning of the run.

If you click anywhere within that Clock Profiling Call Stacks bar you select the nearest event and the details for that event are shown in the Selection Details window. From the pattern of the call stacks, you can see that the time in the `jlowfruit.init_good()` and `jlowfruit.insert_good()` routines shown in bright green in the screen shot is considerably shorter than the corresponding time in the `jlowfruit.init_bad()` and `jlowfruit.insert_bad()` routines shown in red.

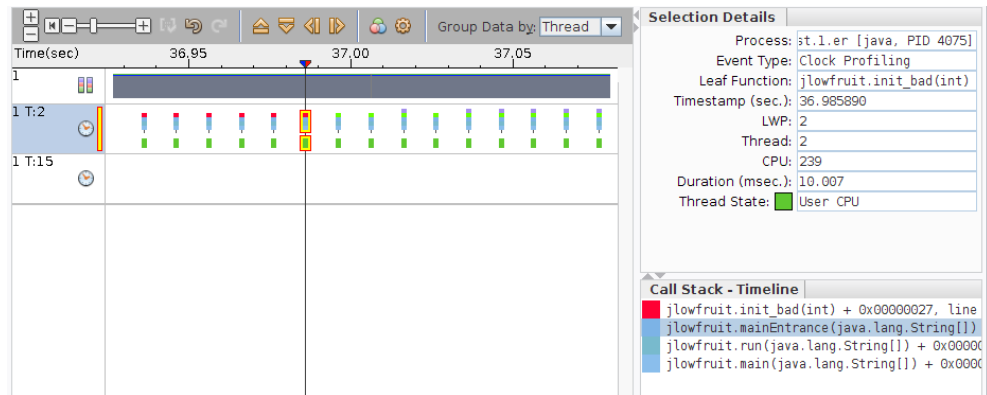
18. Select events in the regions corresponding to the good and bad routines in the timeline and look at the call stacks in the Call Stack - Timeline window below the Selection Details window.

You can select any frame in the Call Stack window, and then select the Source view on the Views navigation bar, and go to the source for that source line. You can also double-click a frame in a call stack to go to the Source view or right-click the frame in the call stack and select from a pop-up menu.

19. Zoom in on the events by using one of the following methods:

- Double-click on the area of interest.
- Drag the cursor in the ruler to adjust vertical time markers, then press Enter.
- Use the + or - icons in the toolbar.
- Press the plus (+) and minus (-) keys to further adjust the zoom.

If you zoom in enough you can see that the data shown is not continuous but consists of discrete events, one for each profile tick, which is about 10 ms in this example.



Press the F1 key to see the Help for more information about the Timeline view.

20. Click on the Call Tree view or choose Views → Call Tree to see the structure of your program.

The Call Tree view shows a dynamic call graph of the program, annotated with performance information.

Using Performance Analyzer to Examine the jlowfruit Data

test.1.er - Oracle Developer Studio Performance Analyzer

File Views Metrics Tools Help

View Mode: User Find: Find text in view Match Case

Views

Call Tree: FUNCTIONS. Complete view. Threshold: 1% Sort by: metric. Metric: Attributed Total CPU Time

Function	Time (ms)	Percentage
<Total>	78.515	100%
jlowfruit.main(java.lang.String[])	78.175	100%
jlowfruit.run(java.lang.String[])	78.175	100%
jlowfruit.mainEntrance(java.lang.String[])	78.175	100%
jlowfruit.init_bad(int)	32.173	41%
jlowfruit.insert_bad(int)	26.388	34%
jlowfruit.insert_good(int)	14.990	19%
jlowfruit.init_good(int)	4.603	6%
java.io.PrintStream.printf(java.lang.String, java.lang.Object[])	0.020	0%
<JVM-System>	0.330	0%
sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, java.lang.String)	0.010	0%
_wlp_start	0	0%

Selection Details

Name: line 23 in "jlowfru
PC Address: 401:0x000043F
Size: 0
Source File: jlowfruit.java
Object File: test.1.er/archives/f
Load Object: jlowfruit.class (fo
Mangled Name: jlowfruit.mainEntra
Aliases:

Total Thread Time: 0. (Exclu
Total CPU Time: n (

Call Stack - Timeline

- jlowfruit.init_bad(int) + 0x0000
- jlowfruit.mainEntrance(java.lang
- jlowfruit.run(java.lang.String[]
- jlowfruit.main(java.lang.String[]

Java and Mixed Java-C++ Profiling

This chapter covers the following topics.

- [“About the Java-C++ Profiling Tutorial” on page 49](#)
- [“Setting Up the jsynprog Sample Code” on page 50](#)
- [“Collecting the Data From jsynprog” on page 51](#)
- [“Examining the jsynprog Data” on page 52](#)
- [“Examining Mixed Java and C++ Code” on page 56](#)
- [“Understanding the JVM Behavior” on page 60](#)
- [“Understanding the Java Garbage Collector Behavior” on page 64](#)
- [“Understanding the Java HotSpot Compiler Behavior” on page 70](#)

About the Java-C++ Profiling Tutorial

This tutorial demonstrates the features of the Oracle Developer Studio Performance Analyzer for Java profiling. It shows you how to use a sample code to do the following in Performance Analyzer:

- Examine the performance data in various data views including the Overview page, and the Threads, Functions, and Timeline views.
- Look at the Source and Disassembly for both Java code and C++ code.
- Learn the difference between User Mode, Expert Mode, and Machine Mode.
- Drill down into the behavior of the JVM executing the program and see the generated native code for any HotSpot-compiled methods.
- See how the garbage collector can be invoked by user code and how the HotSpot compiler is triggered.

jsynprog is a Java program that has a number of subtasks typical of Java programs. The program also loads a C++ shared object and calls various routines from it to show the seamless transition from Java code to native code from a dynamically loaded C++ library, and back again.

`jsynprog.main` is the main method that calls functions from different classes. It uses `gethrtime` and `gethrvtime` through Java Native Interface (JNI) calls to time its own behavior, and writes an accounting file with its own timings, as well as writing messages to `stdout`.

`jsynprog.main` has many methods:

- `Routine.memalloc` does memory allocation, and triggers garbage collection
- `Routine.add_int` does integer addition
- `Routine.add_double` does double (floating point) additions
- `Sub_Routine.add_int` is a derived calls that overrides `Routine.add_int`
- `Routine.has_inner_class` defines an inner class and uses it
- `Routine.recurse` shows direct recursion
- `Routine.recursedeep` does a deep recursion, to show how the tools deal with a truncated stack
- `Routine.bounce` shows indirect recursion, where `bounce` calls `bounce_b` which in turn calls back into `bounce`
- `Routine.array_op` does array operations
- `Routine.vector_op` does vector operations
- `Routine.sys_op` uses methods from the `System` class
- `jsynprog.jni_JavaJavaC`: Java method calls another Java method that calls a C function
- `jsynprog.JavaCJava`: Java method calls a C function which in turn calls a Java method
- `jsynprog.JavaCC`: Java calls a C function that calls another C function

Some of those methods are called from others, so they do not all represent the top-level tasks.

The data you see in the experiment that you record will be different from that shown here. The experiment used for the screen-shots in the tutorial was recorded on a SPARC T5 system running Oracle Solaris 11.3. The data from an x86 system running Oracle Solaris or Linux will be different. Furthermore, data collection is statistical in nature and varies from experiment to experiment, even when run on the same system and OS.

The Performance Analyzer window configuration that you see might not precisely match the screen shots. Performance Analyzer enables you to drag separator bars between components of the window, collapse components, and resize the window. Performance Analyzer records its configuration and uses the same configuration the next time it runs. Many configuration changes were made in the course of capturing the screen shots shown in the tutorial.

Setting Up the jsynprog Sample Code

Before You Begin:

See the following for information about obtaining the code and setting up your environment.

- [“Getting the Sample Code for the Tutorials” on page 10](#)
- [“Setting Up Your Environment for the Tutorials” on page 11](#)

You might want to go through the introductory tutorial in [“Introduction to Java Profiling”](#) first to become familiar with Performance Analyzer.

1. Copy the contents of the jsynprog directory to your own private working area with the following command:

```
% cp -r OracleDeveloperStudio12.6-Samples/PerformanceAnalyzer/jsynprog directory
```

where *directory* is the working directory you are using.

2. Change to that working directory copy.

```
% cd directory/jsynprog
```

3. Build the target executable.

```
% make clobber
```

```
% make
```

Note - The `clobber` subcommand is only needed if you ran `make` in the directory before, but safe to use in any case.

After you run `make`, the directory contains the target application to be used in the tutorial, a Java class file named `jsynprog.class` and a shared object named `libcloop.so` which contains C++ code that will be dynamically loaded and invoked from the Java program.

Tip - If you prefer, you can edit the `Makefile` to do the following: use the GNU compilers rather than the default of the Oracle Developer Studio compilers; build in 32-bits rather than the default of 64-bits; and add different compiler flags.

Collecting the Data From jsynprog

The easiest way to collect the data is to run the following command in the jsynprog directory:

```
% make collect
```

The `collect` target of the Makefile launches a `collect` command and records an experiment. By default, the experiment is named `test.1.er`.

The `collect` target specifies options `-J "-Xmx100m -XX:ParallelGCThreads=10"` for the JVM and collects clock-profiling data by default.

Alternatively, you can use the Performance Analyzer's Profile Application dialog to record the data. Follow the procedure [“Using Performance Analyzer to Collect Data from jlowfruit” on page 33](#) in the introductory Java tutorial and specify `jsynprog` instead of `jlowfruit` in the Arguments field.

Examining the jsynprog Data

This procedure assumes you have already created an experiment as described in the previous section.

1. Start Performance Analyzer from the `jsynprog` directory and load the experiment as follows, specifying your experiment name if it is not called `test.1.er`.

```
% analyzer test.1.er
```

When the experiment opens, Performance Analyzer shows the Overview page.

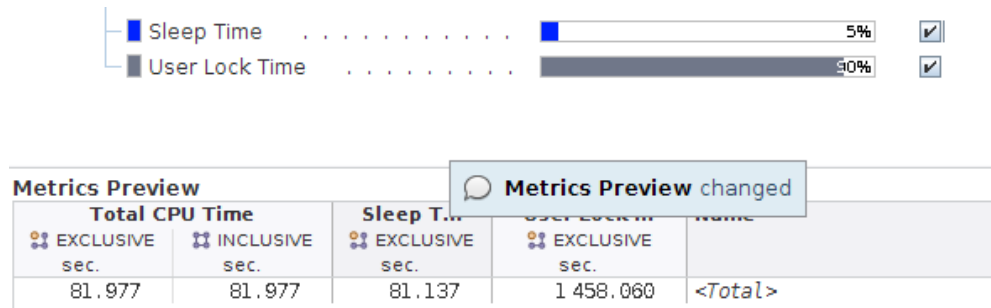
The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window displays the 'test.1.er' experiment. The 'Metrics' section is active, showing 'Available Metrics' for 'Clock Profiling'. The 'Total Thread Time' is 1621.194 Seconds. The 'Total CPU Time' is 81.977 seconds. The 'Metrics Preview' table shows the following data:

Total CPU Time		Name
EXCLUSIVE	INCLUSIVE	
sec.	sec.	
81.977	81.977	<Total>

Notice that the tool bar of Performance Analyzer now has a view mode selector that is initially set to User Mode, showing the user model of the program.

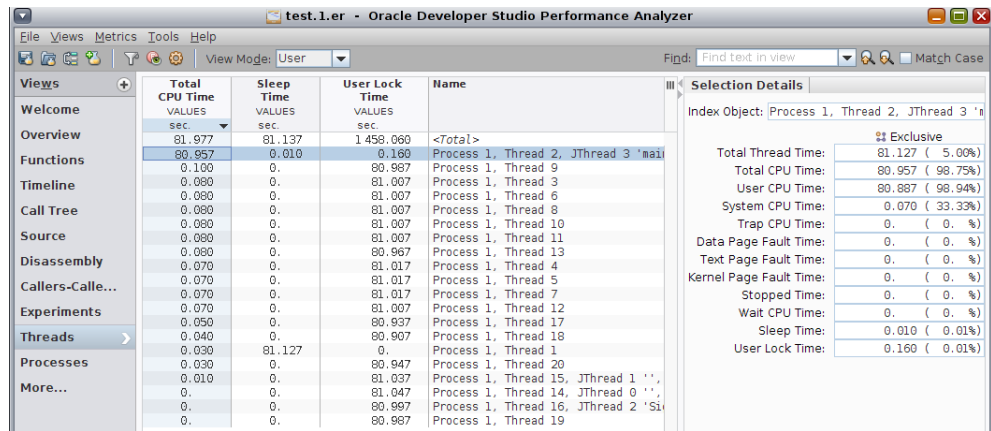
The Overview shows that the experiment ran about 81 seconds but used more than 1600 seconds of total time, implying that on average there were 20 threads in the process.

2. If you ran the application on Oracle Solaris, select the check boxes for the Sleep Time and User Lock Time metrics to add them to the data views.



Notice that the Metrics Preview updates to show you how the data views will look with these metrics added.

3. Select the Threads view in the navigation panel and you will see the data for the threads:



Only Thread 2 accumulated significant Total CPU time. The other threads each had only a few profile events for Total CPU time.

4. Select any thread in the Threads view and see all the information for that thread in the Selection Details window on the right.

You should see that almost all of the threads except Thread 1 and Thread 2 spend all their time in User Lock state. This shows how the JVM synchronizes itself internally. Thread 1 launches the user Java code and then sleeps until it finishes.

- If you selected Sleep Time and User Lock Time in Step 2, go back to the Overview and deselect these checkboxes.
- Select the Functions view in the navigation panel, then click on the column headers to sort by Exclusive Total CPU Time, Inclusive Total CPU Time, or Name.

You can sort by descending or ascending order.

Leave the list sorted by Inclusive Total CPU Time in descending order and select the top-most function `jsynprog.main()`. That routine is the initial routine that the JVM calls to start execution.

The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window displays a table of functions sorted by Inclusive Total CPU Time. The top function is `jsynprog.main(java.lang.String[])` with 65.376 seconds. Below it are various routines like `Routine.bounce`, `Routine.recurse`, and `Routine.vector_op`. The 'Called-by / Calls' panel at the bottom shows that `jsynprog.main` is called by `<Total>` and that it calls numerous other routines.

EXCLUSIVE	INCLUSIVE	Name
sec	sec	
81.977	81.977	<Total>
2.262	65.376	jsynprog.main(java.lang.String[])
0	15.421	<Truncated-stack>
15.421	15.421	Routine.recursedeep(int, int, int)
15.351	15.351	Routine.bounce(int, int, int)
0	15.351	Routine.bounce_b(int, int, int)
15.351	15.351	Routine.recurse(int, int, int)
0.010	5.294	jsynprog.Jni_JavaJavaC(int, int)
0	5.294	jsynprog.Jni_JavaJavaC(int, int)
5.294	5.294	Java_jsynprog_JavaJavaC
4.673	4.673	java.lang.System.arraycopy(java.lang.Object, int, java.lang.Object, int, int)
0	4.013	Routine.vector_op(int)

Notice that the Called-by/Calls panel at the bottom of the Functions view show that the `jsynprog.main()` function is called by `<Total>`, meaning it was at the top of the stack.

The Calls side of the panel shows that `jsynprog.main()` calls a variety of different routines, one for each of the subtasks shown in [“About the Java-C++ Profiling Tutorial” on page 49](#) that are directly called from the main routine. The list also includes a few other routines.

Examining Mixed Java and C++ Code

This section features the Call Tree view and Source view, and shows you how to see the relationships between calls from Java and C++ and back again. It also shows how to add the Disassembly view to the navigation panel.

1. Select each of the functions at the top of the list in the Function view in turn, and examine the detailed information in the Selection Details window.

Note that for some functions the Source File is reported as `jsynprog.java`, while for some others it is reported as `cloop.cc`. That is because the `jsynprog` program has loaded a C++ shared object named `libcloop.so`, which was built from the `cloop.cc` C++ source file. Performance Analyzer reports calls from Java to C++ and vice-versa seamlessly.

2. Select the Call Tree in the navigation panel.

The Call Tree view shows graphically how these calls between Java and C++ are made.

The screenshot displays the Oracle Developer Studio Performance Analyzer interface. The Call Tree view is expanded, showing a hierarchy of functions. The Selection Details window is open, showing information for the selected function: `jsynprog.JavaCJava(int)`.

Call Tree: FUNCTIONS. Complete view. Threshold: 1% Sort by: metric. Metric: At

Function	Time (ns)	Percentage
<Total>	81,977	100%
jsynprog.main(java.lang.String[])	65,376	80%
Routine.bounce(int, int, int)	15,351	19%
Routine.recurse(int, int, int)	15,351	19%
jsynprog.jni.JavaCJavaC(int, int)	5,294	6%
jsynprog.javaCJavaC(int, int)	5,294	6%
Java_jsynprog_JavaCJavaC	5,284	6%
Routine.vector_op(int)	4,013	5%
jsynprog.javaCC(int)	3,763	5%
Java_jsynprog_JavaCC	3,763	5%
cfunc(int)	3,763	5%
Routine.sys_op(int)	3,042	4%
Routine.add_double(int)	3,012	4%
jsynprog.javaCJava(int)	3,012	4%
Java_jsynprog_JavaCJava	3,012	4%
JNIEnv_::CallStaticIntMethod(_jclass*, _methodID*, ...)	3,012	4%
jsynprog.javaFunc(int)	3,012	4%
Sub_Routine.add_int(int)	3,002	4%
Routine.add_int(int)	2,992	4%
Routine.has_inner_class(int)	2,992	4%
Routine.array_op(int)	0,791	1%
Routine.memalloc(int, int)	0,440	1%
jsynprog.printValue(java.lang.String, boolean)	0,040	0%
Launcher.main(java.lang.String[])	0,010	0%
java.lang.ClassLoader.loadClass(java.lang.String)	0,010	0%
java.lang.System.gc()	0	0%
<Truncated-stack>	15,421	19%
<VM-System>	1,131	1%
sun.launcher.LauncherHelper.checkAndLoadMain(boolean, ...)	0,030	0%
<no java callstack recorded>	0,010	0%
_lwp_start	0,010	0%
java.lang.ref.Finalizer\$FinalizerThread.run()	0	0%
java.lang.ref.Reference\$ReferenceHandler.run()	0	0%

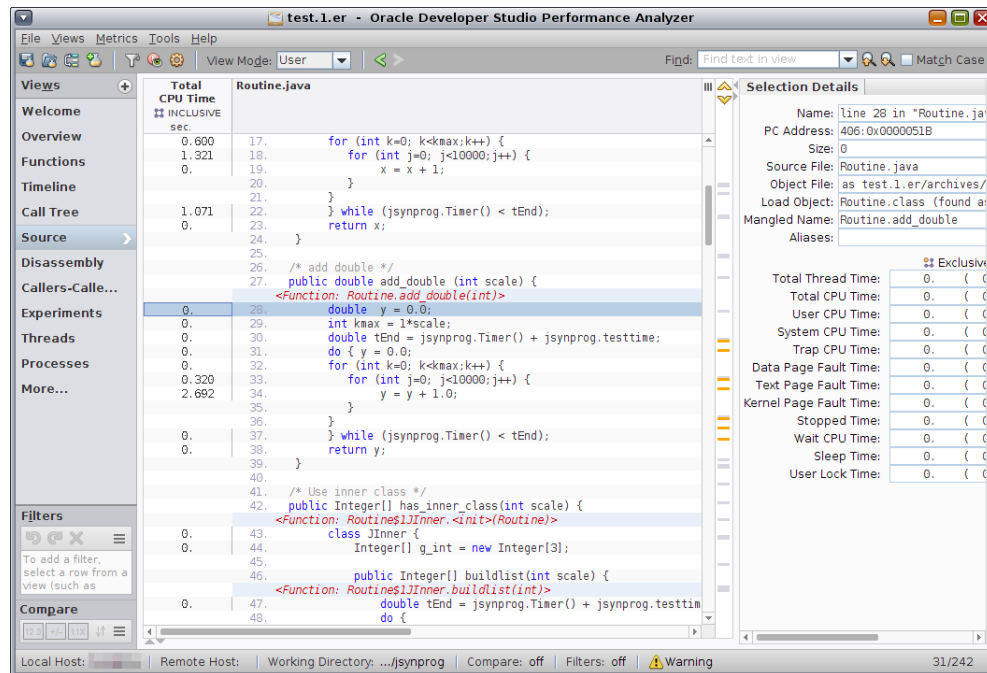
Selection Details

Name: `jsynprog.JavaCJava(int)`
PC Address: `401:0x00000000`
Size: `4294967295`
Source File: `jsynprog.java`
Object File: `synprog.class (found as test.1.er/arch`
Load Object: `jsynprog.class (found as test.1.er/arch`
Mangled Name: `jsynprog.JavaCJava`
Aliases:

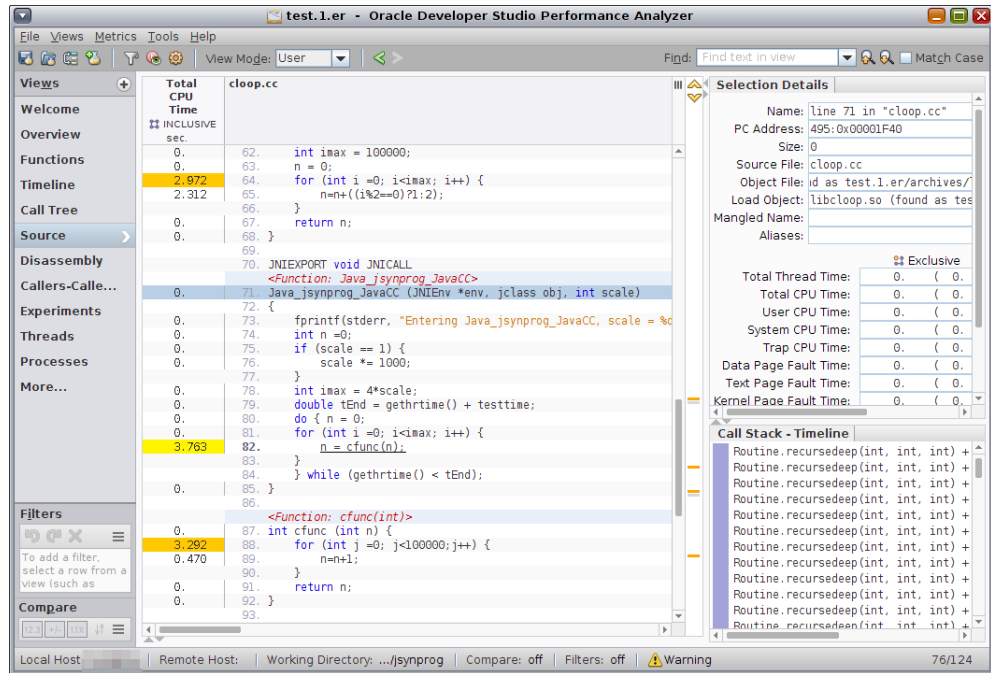
	Exclusive	Inc
Total Thread Time:	0. (0. %)	3.012
Total CPU Time:	0. (0. %)	3.012
User CPU Time:	0. (0. %)	3.012
System CPU Time:	0. (0. %)	0.
Trap CPU Time:	0. (0. %)	0.
Data Page Fault Time:	0. (0. %)	0.
Text Page Fault Time:	0. (0. %)	0.
Kernel Page Fault Time:	0. (0. %)	0.
Stopped Time:	0. (0. %)	0.
Wait CPU Time:	0. (0. %)	0.
Sleep Time:	0. (0. %)	0.
User Lock Time:	0. (0. %)	0.

3. In the Call Tree view, do the following to see the calls from Java to C++ and back to Java:
 - Expand the lines referring to the various functions with "C" in their name.

- Select the line for `jsynprog.JavaCC()`. This function comes from the Java code, but it calls into `Java_jsynprog_JavaCC()` which comes from the C++ code.
 - Select the line for `jsynprog.JavaCJava()`. This function also comes from the Java code but calls `Java_jsynprog_JavaCJava()` which is C++ code. That function calls into a C++ method of the `JNIEnv::CallStaticIntMethod()` which calls back into Java to the method `jsynprog.javafunc()`.
4. Select a method from either Java or C++ and switch to the Source view to see the source shown in the appropriate language along with performance metrics.
- An example of the Source view after selecting a Java method is shown below.

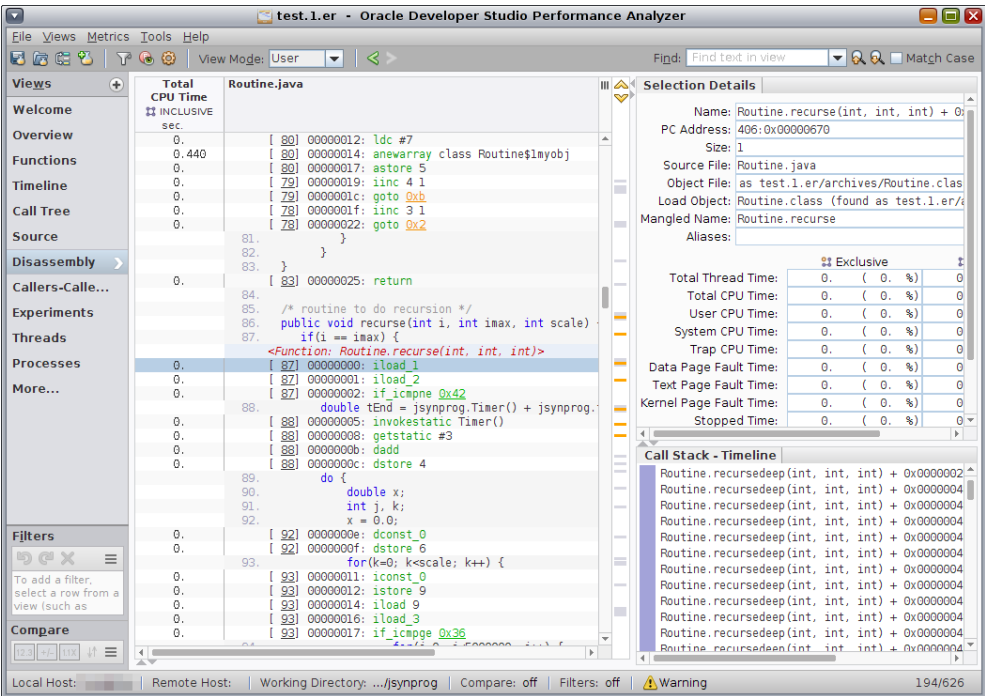


An example of the Source view after selecting a C++ method is shown below.

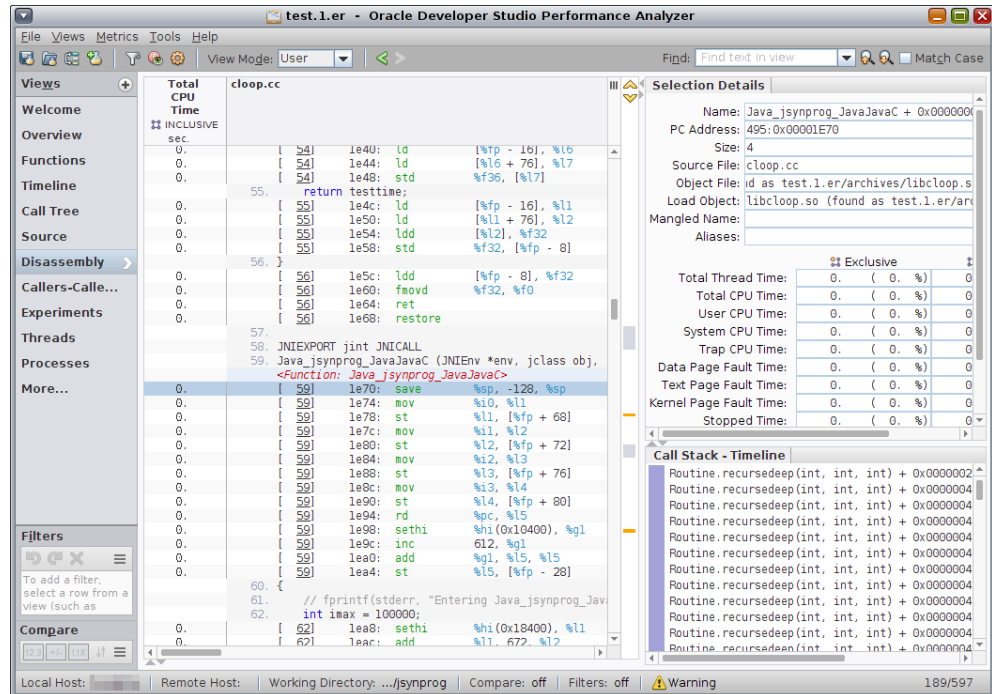


5. If you don't already see the Disassembly tab in the navigation panel, add the View by clicking the + button next to the Views label at the top of the navigation panel and selecting the check box for Disassembly.

The Disassembly view for the function that you last selected is displayed. For a Java function, the Disassembly view shows Java byte code, as shown in the following screen shot.



For a C++ function, the Disassembly view shows native machine code, as shown in the following screen shot.



The next section uses the Disassembly view further.

Understanding the JVM Behavior

This section shows how to examine what is occurring in the JVM by using filters, Expert Mode, and Machine Mode.

1. Select the Functions view and find the routine named <JVM-System>.

You can find it very quickly using the Find tool in the tool bar if you type <JVM> and press Enter.

In this experiment, <JVM-System> consumed about one second of Total CPU time. Time in the <JVM-System> function represents the workings of the JVM rather than the user code.

2. Right-click on <JVM-System> and select "Add Filter: Include only stacks containing the selected functions".

Notice that the filters panel below the navigation panel previously displayed No Active Filters and now shows 1 Active Filter with the name of the filter that you added. The Functions view refreshes so that only <JVM-System> is remaining.

3. In the Performance Analyzer tool bar, change the view mode selector from User Mode to Expert Mode.

The Functions view refreshes to show many functions that had been represented by <JVM-System> time. The function <JVM-System> itself is no longer visible.

4. Remove the filter by clicking the X in the Active Filters panel.

The Functions view refreshes to show the user functions again, but the functions represented by <JVM-System> are also still visible while the <JVM-System> function is not visible.

The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window displays a table of functions with their CPU times. The table has columns for 'EXCLUSIVE sec', 'INCLUSIVE sec', and 'Name'. The 'Name' column is highlighted in blue for the selected row.

EXCLUSIVE sec	INCLUSIVE sec	Name
81.977	81.977	<Total>
2.262	65.376	jsynprog.main(java.lang.String[])
0	15.421	<Truncated-stack>
15.421	15.421	Routine.recursedeep(int, int, int)
15.351	15.351	Routine.bounce(int, int, int)
0	15.351	Routine.bounce_b(int, int, int)
15.351	15.351	Routine.recurse(int, int, int)
0.010	5.284	jsynprog.JavaJavaC(int, int)
0	5.284	jsynprog.jni_JavaJavaC(int, int)
5.284	5.284	Java.jsynprog_JavaJavaC
4.673	4.673	java.lang.System.arraycopy(java.lang.Object, int, java.lang.Object, int, int)
0	4.013	Routine.vector_op(int)
0	3.923	Routine.vrem_first(java.util.Vector)
0.020	3.923	java.util.Vector.remove(int)
0	3.763	Java.jsynprog_JavaCC
3.763	3.763	cfunc(int)
0	3.763	jsynprog.JavaCC(int)
3.032	3.042	Routine.sys_op(int)
0	3.012	JNIEnv_::CallStaticIntMethod(_jclass*,_jmethodID*,...)
3.012	3.012	Java.jsynprog_JavaCJava
0	3.012	Routine.add_double(int)
3.012	3.012	jsynprog.JavaCJava(int)
0.150	3.002	jsynprog.javafunc(int)
2.992	2.992	Sub_Routine.add_int(int)
2.992	2.992	Routine\$IJInner.buildlist(int)
0	2.992	Routine.add_int(int)
2.852	2.852	Routine.has_inner_class(int)
0	1.121	Sub_Routine.addcall(int)
0	0.991	_lwp_start
0.010	0.791	java_start
0	0.791	Routine.array_op(int)
0.190	0.610	GCTaskThread::run()
0.440	0.440	StealTask::do_it(GCTaskManager*, unsigned)
0.270	0.420	Routine.memalloc(int, int)
0	0.130	ParallelTaskTerminator::offer_termination(TerminatorTerminator*)
0	0.120	JavaMain
0	0.120	JNI_CreateJavaVM
0	0.120	Threads::create_vm(JavaVMInitArgs*, bool*)

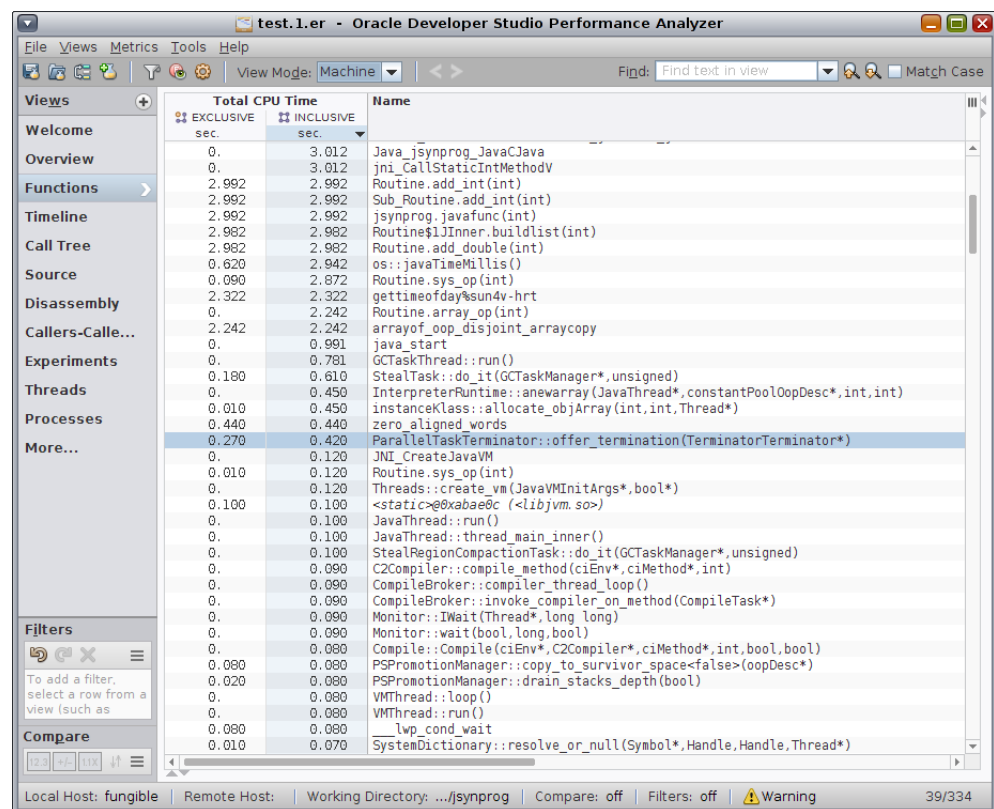
The 'Filters' panel on the left shows one active filter: 'ParallelTaskTerminator::offer_termination(TerminatorTerminator*)'. The status bar at the bottom indicates 'Local Host: ... | Remote Host: ... | Working Directory: .../jsynprog | Compare: off | Filters: off | Warning | 35/344'.

Note that you do not need to perform filtering to expand the <JVM-System>. This procedure includes filtering to more easily show the differences between User Mode and Expert Mode.

To summarize: User Mode shows all the user functions but aggregates all the time spent in the JVM into <JVM-System> while Expert Mode expands that <JVM-System> aggregation.

Next you can explore Machine Mode.

5. Select Machine Mode in the view mode list.



In Machine Mode, any user methods that are interpreted are not shown by name in the Functions view. The time spent in interpreted methods is aggregated into the Interpreter entry, which represents that part of the JVM that interpretively executes Java byte code.

However, in Machine Mode the Functions view displays any user methods that were HotSpot-compiled. If you select a compiled method such as `Routine.add_int()`, the Selection Details window shows the method's Java source file as the Source File, but the Object File and Load Object are shown as `JAVA_COMPILED_METHODS`.

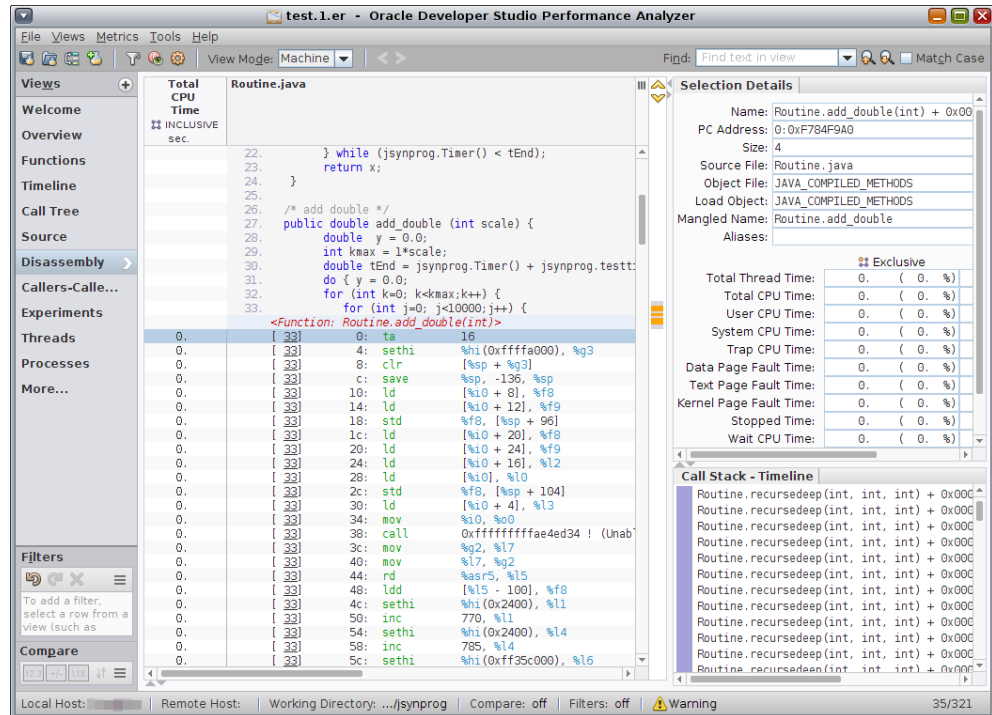
The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window is titled "test.1.er - Oracle Developer Studio Performance Analyzer". The "Views" panel on the left is set to "Functions". The main area displays a table of functions with columns for "Total CPU Time", "EXCLUSIVE sec.", and "INCLUSIVE sec.". The function `Routine.add_double(int)` is selected. The "Selection Details" window on the right shows the following information:

- Name: `Routine.add_double(int)`
- PC Address: `0:0xF784F9A0`
- Size: 384
- Source File: `Routine.java`
- Object File: `JAVA_COMPILED_METHODS`
- Load Object: `JAVA_COMPILED_METHODS`
- Mangled Name: `Routine.add_double`
- Aliases:

Below the Selection Details window, the "Call Stack - Timeline" shows a list of recursive calls to `Routine.recursedeeep(int, int, int) + 0x000`.

- While still in Machine Mode, switch to the Disassembly view while a compiled method is selected in the Functions view.

The Disassembly view shows the machine code generated by the HotSpot Compiler. You can see the Source File, Object File and Load Object names in the column header above the code.



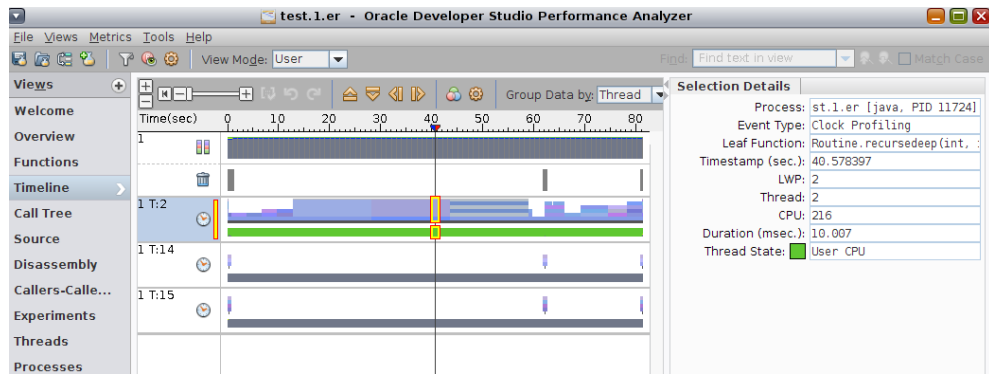
The Total CPU Time shown on most of the visible lines is zero, because most of the work in that function is performed further down in the code.

Continue to the next section.

Understanding the Java Garbage Collector Behavior

This procedure shows you how to use the Timeline view and the affect of the view mode setting on the Timeline, while examining the activities that trigger Java garbage collection.

1. Set the view mode to User Mode and select the Timeline view in the navigation panel to reveal the execution detail of this hybrid Java/native application, jsynprog.



You should see the Process-Wide Resource-Utilization Samples bar at the top and profile data for three threads. In the screen shot you can see data for Process 1, Threads 2, 14, 15. The numbering and the number of threads you see might depend on the OS, the system, and the version of Java you are using.

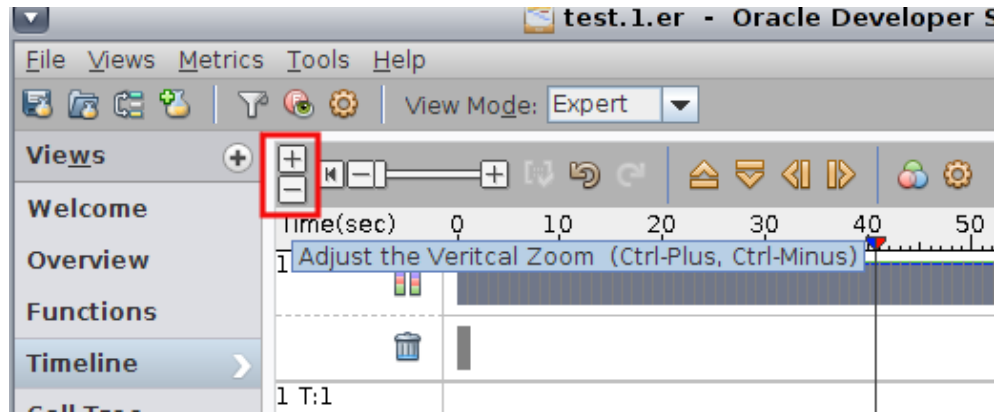
If the application was run on Oracle Solaris, rows showing thread data will also include the thread's state. For example, only the main user thread, labeled as T:2 in the example, shows a green bar, indicating the thread was in User CPU. The other two threads, T:14 and T:15 show grey bars, which indicates User Lock time, part of the JVM synchronization.


2. Set the view mode to Expert Mode.

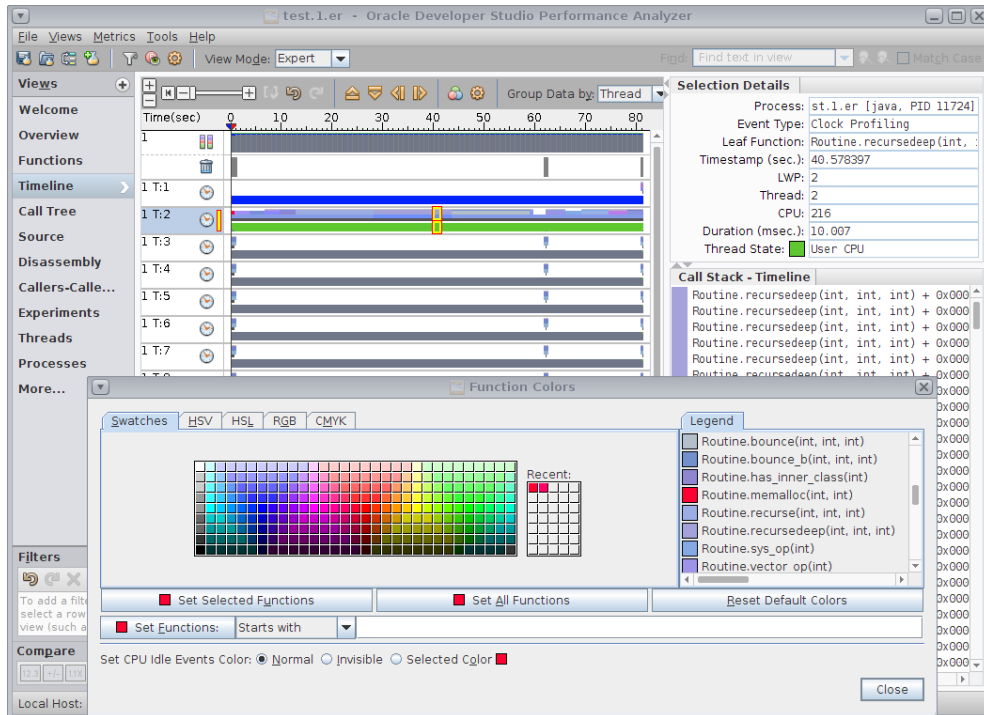
The Timeline view should now show more threads although the user thread T:2 appears almost unchanged.

3. Zoom the time axis to better view callstacks with `Routine.memalloc()`, now in red. You can do this using one of the following methods:
 - Double-click on the callstacks in red.
 - Drag the cursor in the ruler to adjust vertical time markers, then press Enter.
 - Use the + or - icons in the toolbar.
 - Press the plus (+) and minus (-) keys to further adjust the zoom.
4. Click the minus (-) button to reduce the height of the thread rows until you can see all twenty threads.

The vertical zoom control is outlined in red in the following screen shot.



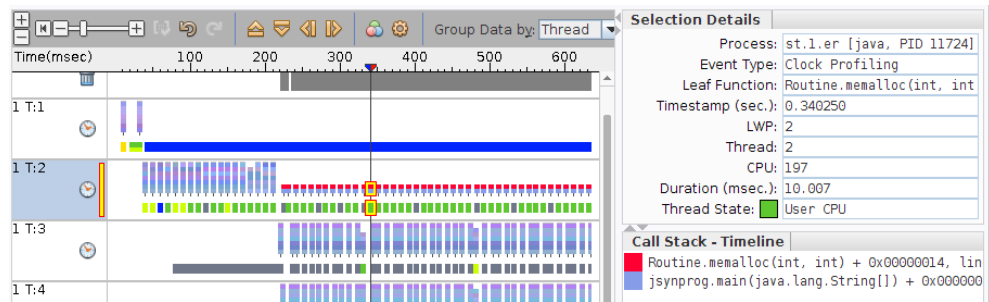
5. Click the Call Stack Function Colors icon  in the Timeline tool bar to set the color of the function Routine.memalloc() to red.
In the Function Colors dialog, select the Routine.memalloc() function in the Legend, click a red box in Swatches and click Set Selected Functions.



Note that Thread 2 now has a bar of red across the top of its stack. That area represents the portion of time where the `Routine.memalloc()` routine was running.

You might need to zoom out vertically to see more frames of the callstack, and zoom in horizontally to the region of time that is of interest.

6. Zoom in close enough to see individual events in thread T:2. You can use one of the following methods:
 - Double-click on the area of interest.
 - Drag the cursor in the ruler to adjust vertical time markers, then press Enter.
 - Use the + or - icons in the toolbar.
 - Press the plus (+) and minus (-) keys to further adjust the zoom.

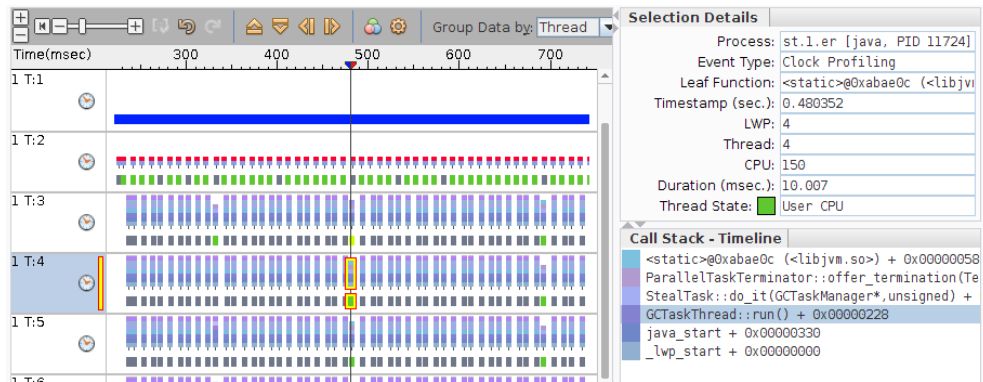


For each active thread of the application, the timeline will show one or more rows of data. The upper portion of each row shows color-coded representations of the thread's callstacks. If you click on a callstack, the function details will be shown in the Call Stack panel. In addition, if an application was profiled on Oracle Solaris, Timeline will show the thread state just below the callstack. In this example, thread T:1 shows predominantly dark blue, representing Sleep. Thread T:2 shows predominantly green, representing User CPU. Thread T:3 shows mostly dark gray, representing Lock.

Notice however that all of those threads 3 through 12 have many events clustered together arriving at the same time as the user thread T:2 is in Routine.memalloc, the routine shown in red.

7. Zoom in to the Routine.memalloc region, which have the callstacks with the red bar, and filter to include only that region by doing the following:
 - In the T:2 bar, locate and click on calls to Routine.memalloc().
 - In the timeline ruler, the area showing the timescale below the toolbar, click and drag the cursor to set time markers that enclose the calls to Routine.memalloc().
 - Right-click and select Zoom → To Selected Time Range.
 - With the range still selected, right-click and select Add Filter: Include only events intersecting selected time range.
8. Click on any of the events on threads 3-12 and you see in the Call Stack panel that each thread's events include GCTaskThread : : run() in the stack.

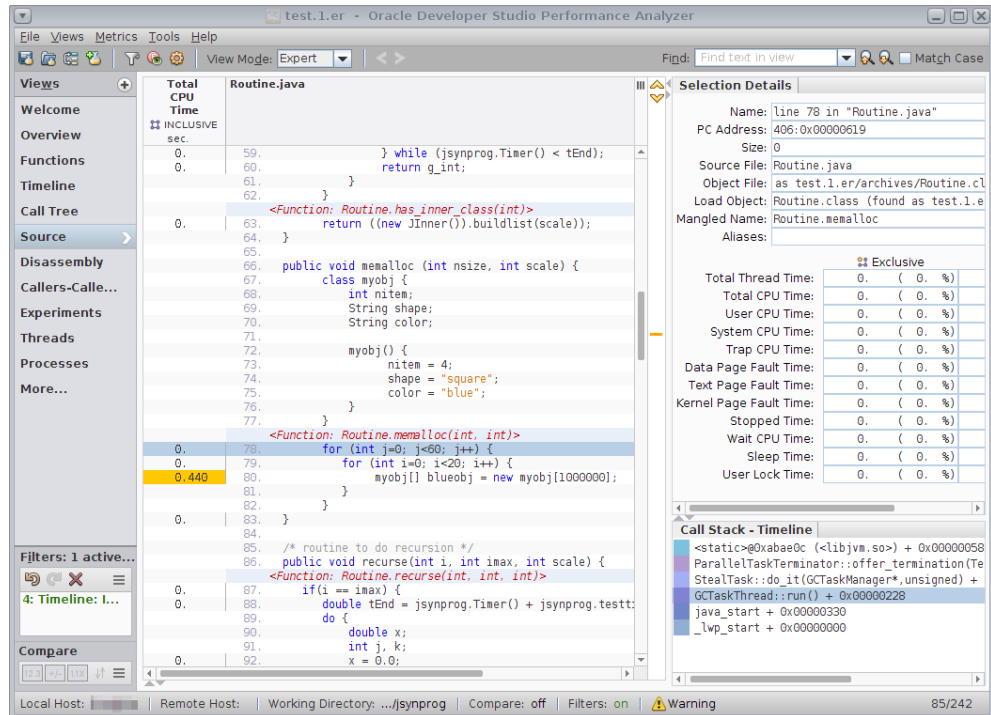
Those threads represent the threads that the JVM uses to run garbage collection. The GC threads do not take a great amount of User CPU Time and only run while the user thread is in Routine.memalloc.



- Go back to the Functions view and click on the Incl. Total CPU column header to sort by inclusive Total CPU Time.

You should see that one of the top functions is the `GCTaskThread::run()` function. This leads you to the conclusion that the user task `Routine.memalloc` is somehow triggering garbage collection.

- Select the `Routine.memalloc` function and switch to the Source view.



From this fragment of source code it is easy to see why garbage collection is being triggered. The code allocates an array of one million objects and stores the pointers to those objects in the same place with each pass through the loop. This renders the old objects unused, and thus they become garbage.

Continue to the next section.

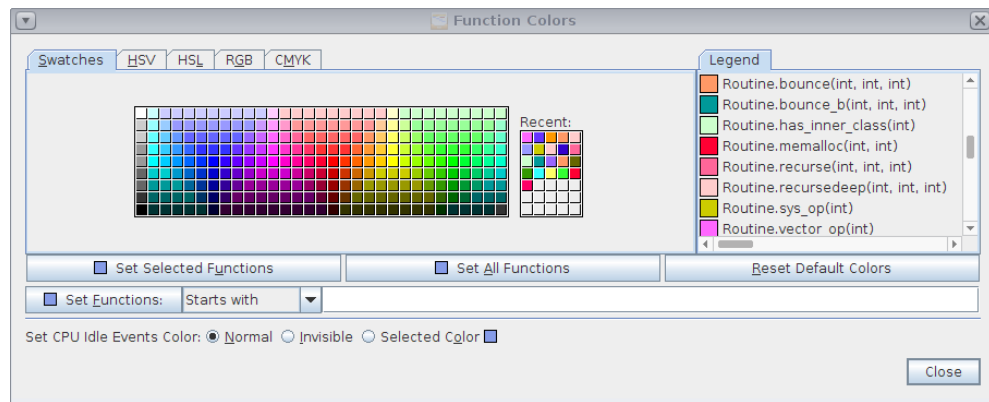
Understanding the Java HotSpot Compiler Behavior

This procedure continues from the previous section, and shows you how to use the Timeline and Threads views to filter and find the threads responsible for HotSpot compiling.

1. Select the Timeline view and remove the filter by clicking the X in the Active Filters panel
2. Reset the zoom levels by doing one of the following:
 - Right-click in the Timeline and select Zoom → Reset Time Zoom

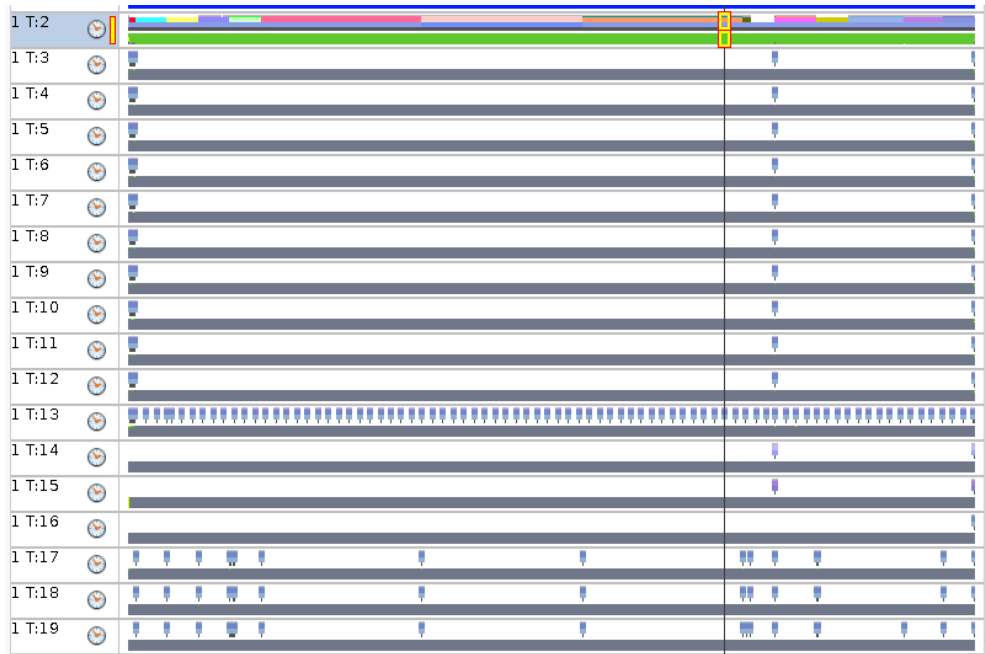
- Click the |< button to the left of the horizontal slider in the Timeline tool bar
 - Press zero (0) on your keyboard.
3. Open the Function Colors dialog again, and pick different colors for each of the Routine.* functions.

In the Timeline view, the color changes appear in call stacks of thread 2.



4. Look at all the threads of the Timeline in the period of time where you see the color changes in thread 2.

You should see that there are some threads with patterns of events occurring at just about the same time as the color changes in thread 2. In this example, they are threads 17, 18, and 19.

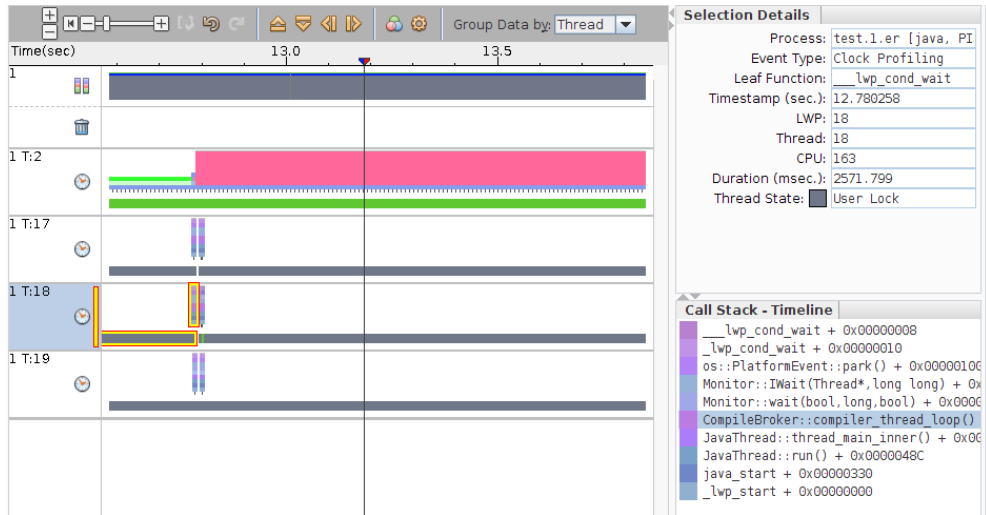


5. Press Ctrl and multi-select the main thread (T:2 in this example) and the rows in your experiment that show a pattern similar to T:17 - T:19 in this example.
6. Right-click on the timeline view and select the filter Include Only Selected Rows.

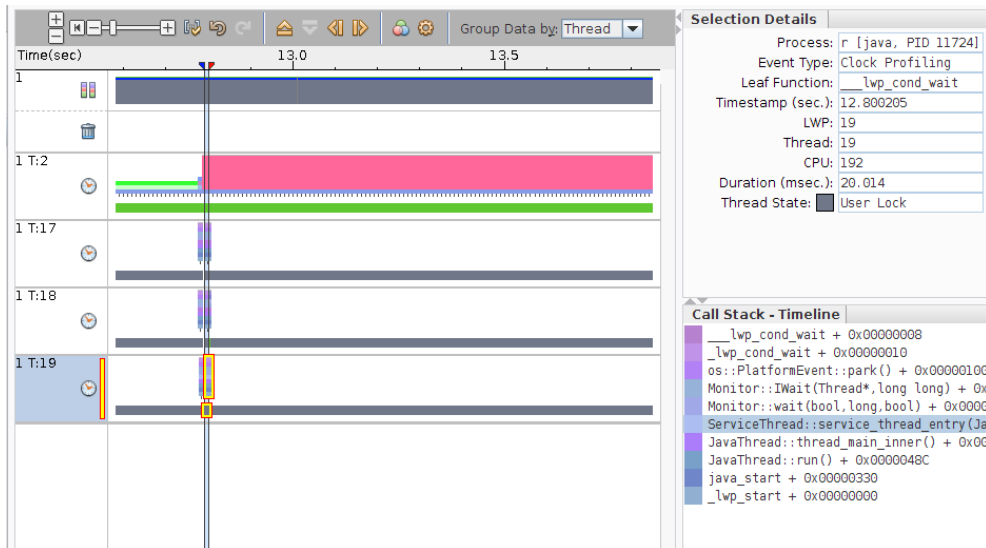
Alternatively, click the filter button  in the toolbar and select the filter.

7. Adjust the horizontal zoom to make the pattern easier to see.
8. Click on events in threads 17 and 18.

Note that the Call Stack panel shows `CompileBroker::compiler_thread_loop()`. Those threads are the threads used for the HotSpot compiler.



Thread 19 shows call stacks with `ServiceThread::service_thread_entry()` in them.



The reason the multiple events occur on those threads is that whenever the user code invokes a new method and spends a fair amount of time in it, the HotSpot compiler is triggered to generate machine code for that method. The HotSpot compiler is fast enough that the threads that run it do not consume very much User CPU Time.

The details of exactly how the HotSpot compiler is triggered is beyond the scope of this tutorial.

Hardware Counter Profiling on a Multithreaded Program

This chapter covers the following topics.

- [“About the Hardware Counter Profiling Tutorial” on page 75](#)
- [“Setting Up the `mttest` Sample Code” on page 76](#)
- [“Collecting Data From `mttest` for Hardware Counter Profiling Tutorial” on page 77](#)
- [“Examining the Hardware Counter Profiling Experiment for `mttest`” on page 78](#)
- [“Exploring Clock-Profiling Data” on page 80](#)
- [“Understanding Hardware Counter Instruction Profiling Metrics” on page 82](#)
- [“Understanding Hardware Counter CPU Cycles Profiling Metrics” on page 84](#)
- [“Understanding Cache Contention and Cache Profiling Metrics” on page 86](#)
- [“Detecting False Sharing” on page 90](#)

About the Hardware Counter Profiling Tutorial

This tutorial shows how to use Performance Analyzer on a multithreaded program named `mttest` to collect and understand clock profiling and hardware counter profiling data.

You explore the Overview page and change which metrics are shown, examine the Functions view, Callers-Callees view, and Source and Disassembly views, and apply filters.

You first explore the clock profile data, then the HW-counter profile data with Instructions Executed which is a counter available on all supported systems. Then you explore Instructions Executed and CPU Cycles (available on most, but not all, supported systems) and with D-cache Misses (available on some supported systems).

If run on a system with a precise hardware counter for D-cache Misses (`dcm`), you will also learn how to use the `IndexObject` and `MemoryObject` views, and how to detect false sharing of a cache line.

The program `mttest` is a simple program that exercises various synchronization options on dummy data. The program implements a number of different tasks and each task uses a basic algorithm:

- Queue up a number of work blocks, four by default. Each one is an instance of a structure `Workblk`.
- Spawn a number of threads to process the work, also four by default. Each thread is passed its private work block.
- In each task, use a particular synchronization primitive to control access to the work blocks.
- Process the work for the block, after the synchronization.

The data you see in the experiment that you record will be different from that shown here. The experiment used for the screen shots in the tutorial was recorded on a SPARC T5 system running Oracle Solaris 11.3. The data from an x86 system running Oracle Solaris or Linux will be different. Furthermore, data collection is statistical in nature and varies from experiment to experiment, even when run on the same system and OS.

The Performance Analyzer window configuration that you see might not precisely match the screen shots. Performance Analyzer enables you to drag separator bars between components of the window, collapse components, and resize the window. Performance Analyzer records its configuration and uses the same configuration the next time it runs. Many configuration changes were made in the course of capturing the screen shots shown in the tutorial.

Setting Up the `mttest` Sample Code

Before You Begin:

See the following for information about obtaining the code and setting up your environment.

- [“Getting the Sample Code for the Tutorials” on page 10](#)
- [“Setting Up Your Environment for the Tutorials” on page 11](#)

You might want to go through the introductory tutorial in [“Introduction to C Profiling”](#) first to become familiar with Performance Analyzer.

1. Copy the contents of the `mttest` directory to your own private working area with the following command:

```
% cp -r OracleDeveloperStudio12.6-Samples/PerformanceAnalyzer/mttest directory
```

Replace *directory* with the working directory you are using.

2. Change to that working directory copy.

```
% cd directory/mttest
```

3. Build the target executable.

```
% make clobber
```

```
% make
```

Note - The `clobber` subcommand is only needed if you ran `make` in the directory before, but safe to use in any case.

After you run `make` the directory contains the target application to be used in the tutorial, a C program called `mttest`.

Tip - If you prefer, you can edit the `Makefile` to do the following: use the GNU compilers rather than the default of the Oracle Developer Studio compilers; build in 32-bits rather than the default of 64-bits; and add different compiler flags.

Collecting Data From `mttest` for Hardware Counter Profiling Tutorial

The easiest way to collect the data is to run the following command in the `mttest` directory:

```
% make hwcperf
```

The `hwcperf` target of the `Makefile` launches a `collect` command and records an experiment.

Note - This tutorial might take a longer time compiling and collecting data than the previous introductory tutorials.

The experiment is named `test.1.er` by default and contains clock-profiling data and hardware counter profiling data for the default counters for your system. In this chapter's example, the counters are: `inst` (instructions), `cycles` (cycles), and `dcm` (data-cache-misses). If Performance Analyzer does not define a default set of hardware counters for your system, the `collect` command fails. In that case, edit the `HWC_OPT` definition in your `Makefile` to specify counters that are supported in your system.

Tip - You can use the command `collect -h` to determine which counters your system does support. For information about the hardware counters, see [“Hardware Counter Lists” in Oracle Developer Studio 12.6: Performance Analyzer](#).

Examining the Hardware Counter Profiling Experiment for `mttest`

This section shows how to explore the data in the experiment you created from the `mttest` sample code in the previous section.

Start Performance Analyzer from the `mttest` directory and load the experiment as follows:

```
% analyzer test.1.er
```

When the experiment opens, Performance Analyzer shows the Overview page.

The screenshot displays the Oracle Developer Studio Performance Analyzer interface. The main window shows the 'Metrics' section for an experiment named 'test.1.er'. The 'Clock Profiling' section is expanded, showing a tree view of metrics with colored bars and checkboxes. The 'Total Thread Time' is 337.946 Seconds. The 'Total CPU Time' is 59%, with 'User CPU Time' at 58%, 'System CPU Time' at 0%, 'Trap CPU Time' at 0%, 'Data Page Fault Time' at 0%, 'Text Page Fault Time' at 0%, 'Kernel Page Fault Time' at 0%, 'Stopped Time' at 0%, 'Wait CPU Time' at 0%, 'Sleep Time' at 25%, and 'User Lock Time' at 17%. The 'Derived and Other Metrics' section shows 'Instructions Per Cycle: 0.201' and 'Cycles Per Instruction: 4.981'. The 'HW Counter Profiling' section shows 'Memoryspace Hardware Counters' (L1 D-cache Misses: 1622907507) and 'General Hardware Counters' (Instructions Executed: 141900334433, CPU Cycles Time: 196.328 Seconds, CPU Cycles: 706782153629). At the bottom, a 'Metrics Preview' table shows 'Total CPU Time' with 'EXCLUSIVE sec.' and 'INCLUSIVE sec.' columns, both showing 197.738, and a 'Name' column with '<Total>'. The status bar at the bottom indicates 'Local Host: ... Remote Host: ... Working Directory: .../mttest Compare: off Filters: off Warning'.

If the application was run on Oracle Solaris, the Clock Profiling metrics are shown first and include colored bars. Most of the thread time is spent in User CPU Time. Some time is spent in Sleep Time or User Lock Time.

If the application was run on Linux, the Clock Profiling section will include only Total CPU Time.

The Derived and Other Metrics group is present if you have recorded both cycles and insts counters. The derived metrics represent the ratios of the metrics from those two counters. A high value of Instructions Per Cycle or a low value of Cycles Per Instruction indicates relatively

efficient code. Conversely, a low value of Instructions Per Cycle or a high value of Cycles Per Instruction indicates relatively inefficient code.

The HW Counter Profiling group shows two subgroups in this experiment, Memoryspace Hardware Counters and General Hardware Counters. The Instructions Executed counter (`insts`) is listed under General Hardware Counters. If the data you collected included the `cycles` counter, CPU Cycles is also listed under General Hardware Counters. If counters were enabled that support memoryspace profiling, they will be listed under Memoryspace Hardware Counters. In the previous example, L1 D-cache Misses is such a counter. On Linux, memoryspace profiling is not supported. For more information about memoryspace profiling, see [“Dataspace Profiling and Memoryspace Profiling” in Oracle Developer Studio 12.6: Performance Analyzer](#).

Your system will likely have different default counters and metrics. You can edit the `Makefile` to choose other counters.

You will explore these metrics and their interpretation in the following sections of the tutorial.

Exploring Clock-Profiling Data

This section explores the clock profiling data using the Overview page and the Functions view with the Called-by/Calls panel.

1. In the Overview page, deselect the check boxes for three HW counter metrics, leaving only the Total CPU Time check box selected.
2. Go to the Functions view and click the column heading once for Inclusive Total CPU Time to sort according to inclusive total CPU time.

The function `do_work()` should now be at the top of the list.

The screenshot shows the Oracle Developer Studio Performance Analyzer interface. The main window displays a list of functions with their exclusive and inclusive CPU times. The 'Called-by / Calls' panel at the bottom shows that the `do_work()` function is called by `_lwp_start()` and `locktest()`. The 'Selection Details' panel on the right provides information about the selected `do_work()` function, including its source file and various performance metrics.

EXCLUSIVE sec	INCLUSIVE sec	Name
197.738	197.738	<Total>
0.570	197.728	do_work
0.	165.700	_lwp_start
0.	70.940	cache_trash
70.940	70.940	computeB
0.	29.748	cache_trash_odd
0.	31.192	cache_trash_even
0.510	30.021	trylock_global
5.374	17.502	mutex_trylock
0.	12.128	do_exit_critical
12.128	12.128	take_deferred_direct
0.	12.038	_start
12.038	12.038	computeA
12.038	12.038	computeE
12.038	12.038	computeH
0.	12.038	cond_timeout_global
0.	12.038	lock_local
0.	12.038	lock_none
0.	12.038	locktest

Total Thread Time:	Exclusive	Incl
Total CPU Time:	0.570 (0.17%)	254.74
User CPU Time:	0.570 (0.29%)	197.498
System CPU Time:	0. (0. %)	0.120
Trap CPU Time:	0. (0. %)	0.110
Data Page Fault Time:	0. (0. %)	0.
Text Page Fault Time:	0. (0. %)	0.
Kernel Page Fault Time:	0. (0. %)	0.
Stopped Time:	0. (0. %)	0.
Wait CPU Time:	0. (0. %)	0.
Sleep Time:	0. (0. %)	0.
User Lock Time:	0. (0. %)	57.060
L1 D-cache Misses:	0 (0. %)	1622907507
Instructions Executed:	0 (0. %)	141900334433
CPU Cycles Time:	0.009 (0.00%)	196.328
CPU Cycles:	32010046 (0.00%)	706782153629
Instructions Per Cycle:	0. (0. %)	0.201
Cycles Per Instruction:	0. (0. %)	4.981

- Select the `do_work()` function and look at the Called-by/Calls panel at the bottom of the Functions view.

Note that `do_work()` is called from two places, and it calls ten functions.

The ten functions that `do_work()` calls represent ten different tasks, each with a different synchronization method that the program executed. In some experiments created from `mttest` you might see an eleventh function which uses relatively little time to fetch the work blocks for the other tasks. This function is not shown in the screen shot.

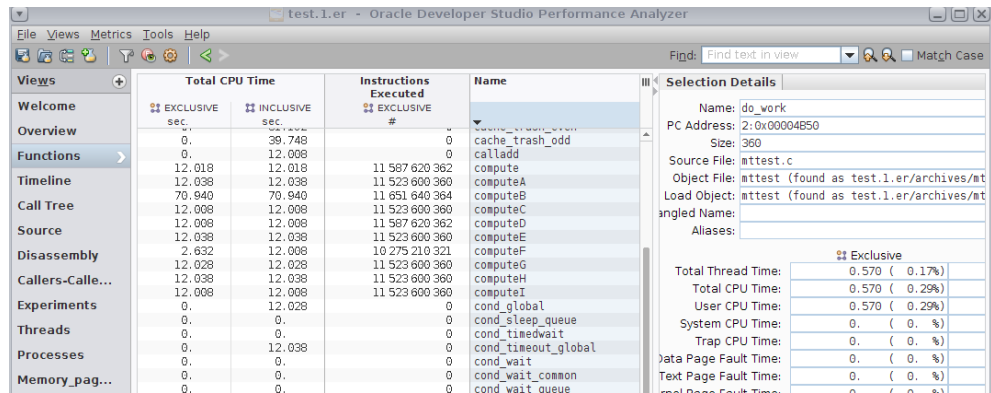
Most often, `do_work()` is called when a thread to process the data is created, and is shown as called from `_lwp_start()`. In one case, `do_work()` calls one single-threaded task called `nothreads()` after being called from `locktest()`.

In the Calls side of the panel, note that except for the first two of the callees, all callees show about the same amount of time (~12 seconds) of Attributed Total CPU.

Understanding Hardware Counter Instruction Profiling Metrics

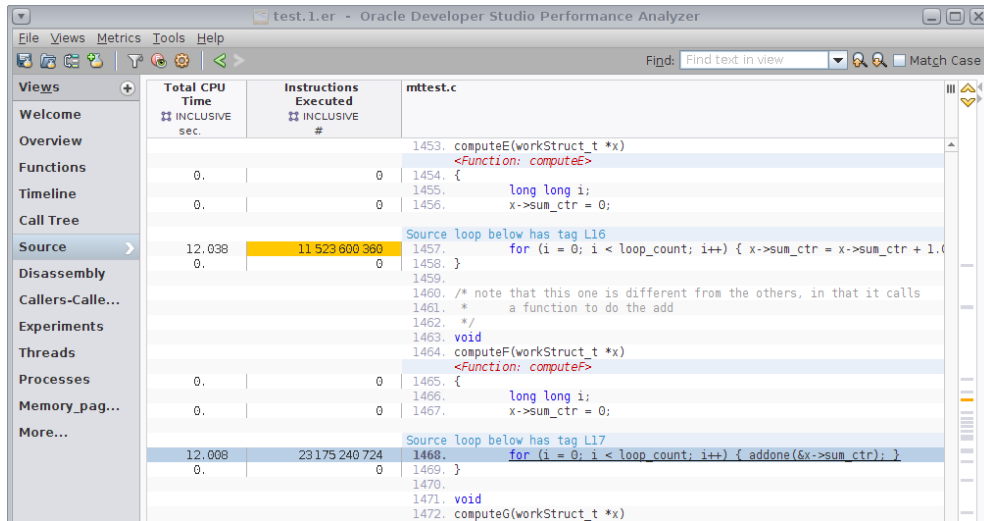
This section shows how to use general hardware counters to see how many instructions are executed for functions.

1. Select the Overview page and enable the HW Counter Profiling metric named Instructions Executed, which is under General Hardware Counters.
2. Return to the Functions view, and click on the Name column header to sort alphabetically.
3. Scroll down to find the functions `compute()`, `computeA()`, `computeB()`, etc.



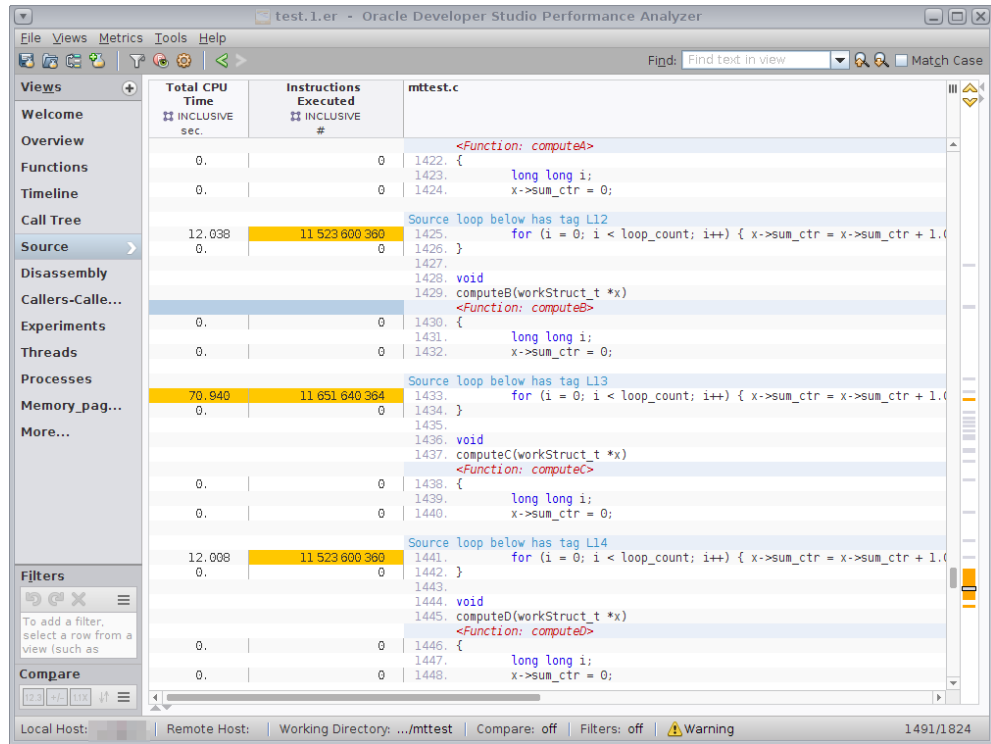
Note that all of the functions except `computeB()` and `computeF()` have approximately the same amount of Exclusive Total CPU time and of Exclusive Instructions Executed.

4. Select `computeF()` and switch to the Source view. You can do this in one step by double-clicking `computeF()`.



The computation kernel in computeF() is different because it calls a function addone() to add one, while the other compute*() functions do the addition directly. This explains why its performance is different from the others.

5. Scroll up and down in the Source view to look at all the compute*() functions. Note that all of the compute*() functions, including computeB(), show approximately the same number of instructions executed. Yet computeB() shows a very different CPU Time cost.



The next section helps show why the Total CPU time is so much higher for computeB().

Understanding Hardware Counter CPU Cycles Profiling Metrics

This part of the tutorial requires an experiment with data from the cycles counter. If your system does not support this counter, your experiment cannot be used in this section. Skip to the next section [“Understanding Cache Contention and Cache Profiling Metrics” on page 86.](#)

1. Select the Overview page and enable the derived metric Cycles Per Instruction and the General Hardware Counter metric, CPU Cycles Time.

You should keep Total CPU Time and Instructions Executed selected.

your experiment that the Incl. Cycles Per Instruction (CPI) is much higher for `computeB()` than it is for the other `compute*()` functions. This indicates that more CPU cycles are needed to execute the same number of instructions, and `computeB()` is therefore less efficient than the others.

The data you have seen so far shows the difference between that `computeB()` function and the others, but does not show why they might be different. The next part of this tutorial explores why `computeB()` is different.

Understanding Cache Contention and Cache Profiling Metrics

This section and the rest of the tutorial requires an experiment with data from the precise `dcm` hardware counter. If your system does not support the precise `dcm` counter, the remainder of the tutorial is not applicable to the experiment you recorded on the system.

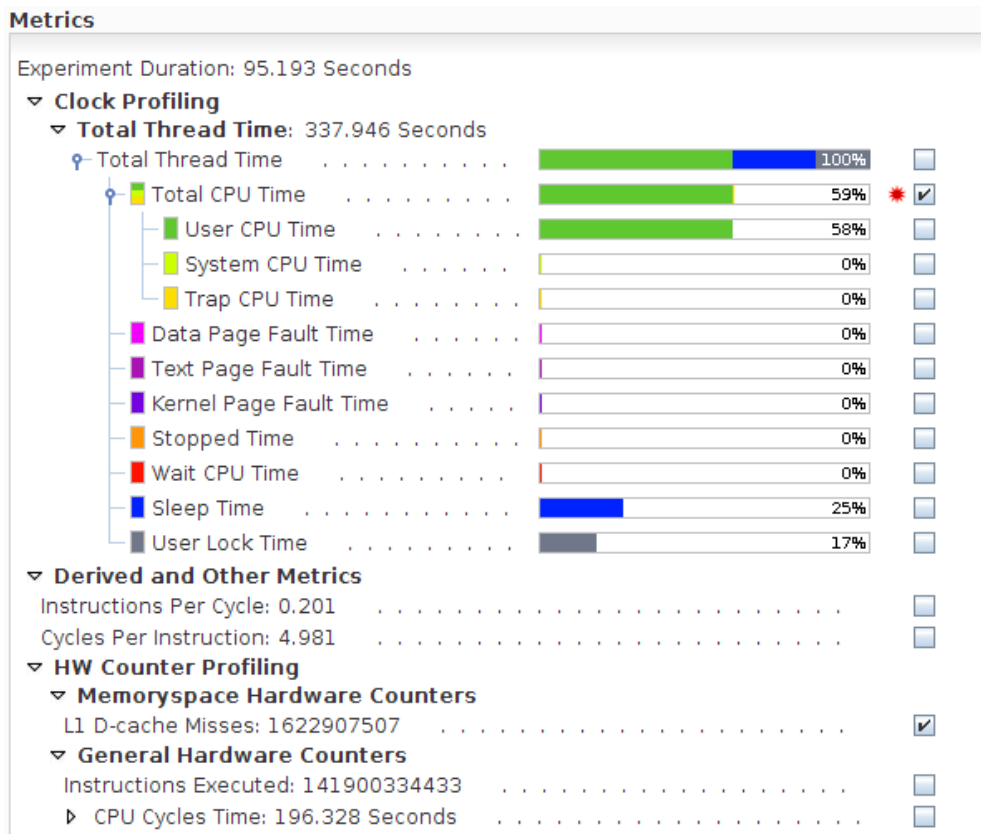
The `dcm` counter is counting cache misses, which are loads and stores that reference a memory address that is not in the cache.

An address might not be in cache for any of the following reasons:

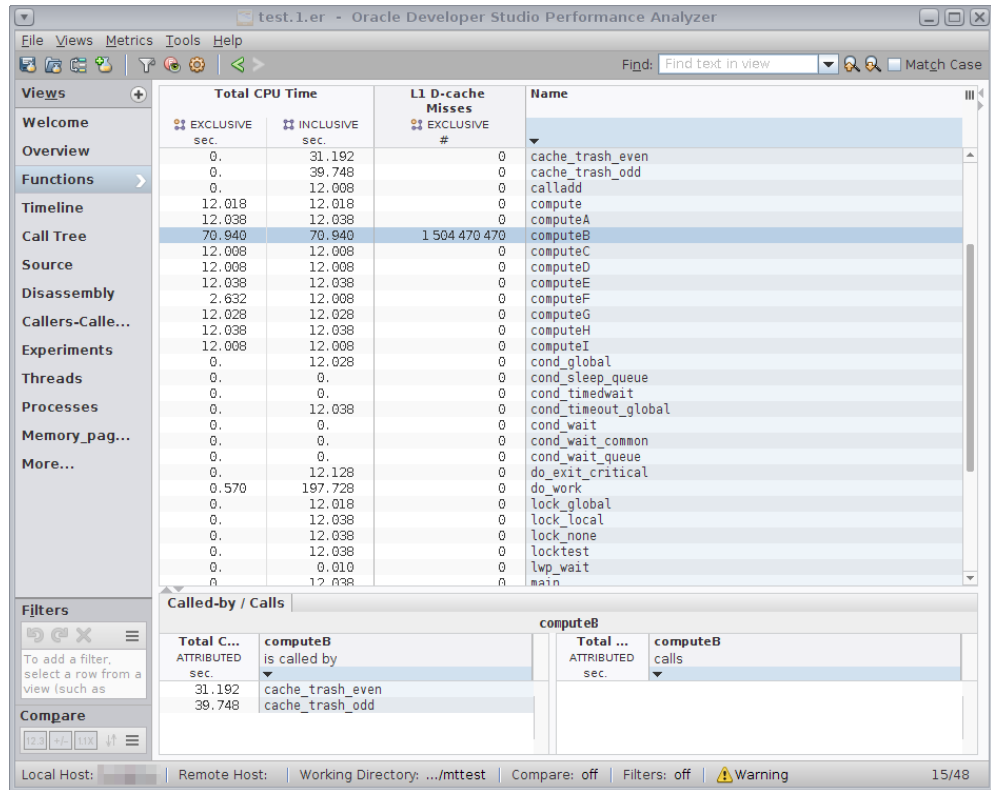
- Because the current instruction is the first reference to that memory location from that CPU. More accurately, it is the first reference to any of the memory locations that share the cache line.
- Because the thread has referenced so many other memory addresses that the current address has been flushed from the cache. This is a capacity miss.
- Because the thread has referenced other memory addresses that map to the same cache line which causes the current address to be flushed. This is a conflict miss.
- Because another thread has written to an address within the cache line which causes the current thread's cache line to be flushed. This is a sharing miss, and could be one of two types of sharing misses:
 - *True sharing*, where the other thread has written to the same address that the current thread is referencing. Cache misses due to true sharing are unavoidable.
 - *False sharing*, where the other thread has written to a different address from the one that the current thread is referencing. Cache misses due to false sharing occur because the cache hardware operates at a cache-line granularity, not a data-word granularity. False sharing can be avoided by changing the relevant data structures so that the different addresses referenced in each thread are on different cache lines.

This procedure examines a case of false sharing that has an impact on the function `computeB()`.

1. Return to the Overview, and enable the metric for L1 D-cache Misses, and disable the metric for Cycles Per Instruction.



2. Switch back to the Functions view and look at the compute*() routines.



Recall that all compute*() functions show approximately the same instruction count, but computeB() shows higher Total CPU Time and is the only function with significant counts for Exclusive L1 D-cache Misses.

- Go back to the Source view and note that in computeB() the cache misses are in the single line loop.
- If you don't already see the Disassembly tab in the navigation panel, add the View by clicking the + button next to the Views label at the top of the navigation panel and selecting the check box for Disassembly.

Scroll the Disassembly view until you see the line with the load instruction with a high number of L1 D-Cache Misses.

Tip - The right margin of views such as Disassembly include shortcuts you can click to jump to the lines with high metrics, or hot lines. Try clicking the Next Hot Line down-arrow at the top of the margin or the Non-Zero Metrics marker to jump quickly to the lines with notable metric values.

Total CPU Time	L1 D-cache Misses	Source	Disassembly	Comment
INCLUSIVE sec.	INCLUSIVE #			
0.	0	1433: for (i = 0; i < loop_count; i++) { x->sum_ctr = x->sum_ctr + 1.0; }	[1433] 100005780: ldx [%02 + 840], %03 <Scalars>.{long_long loop_count}	
0.	0		[1433] 100005784: brlezn [%03, 0x1000057c4] %f0	
0.	0		[1433] 10000578c: sethi [%hi(0x100000), %01] %g3	
0.	0		[1433] 100005790: clr [%03] %g3	
0.	0		[1433] 100005794: or [%01, %05, %g5] %f2	{structure:workStruct_t -}.{double sum_ctr}
0.	0		[1433] 100005798: ldd [%00], %f2	
0.	0		[1430] 10000579c: add [%02, 840, %g2] %f2	
0.	0		[1433] 1000057a0: sllx [%05, 12, %g4] %f6	
0.	0		[1433] 1000057a4: ldd [%04 + 1856], %f6	
0.	0		[1433] 1000057a8: <branch target>	<=====
49.485	3 201 001		[1433] 1000057a8: fadd [%f2, %f6, %f4] %f2	{structure:workStruct_t -}.{double sum_ctr}
11.098	0		[1433] 1000057ac: std [%f4, %00] %g3	
4.643	0		[1433] 1000057b0: inc [%03] %g3	
1.481	0		[1433] 1000057b4: ldx [%g2], %g1 <Scalars>.{long_long loop_count}	
0.	0		[1433] 1000057b8: cmp [%g3, %g1] %cc, 0x1000057a8	
1.441	0		[1433] 1000057bc: bl,a,pt [%00], %f2	{structure:workStruct_t -}.{double sum_ctr}
2.822	1 501 269 469		[1433] 1000057c0: ldd [%00], %f2	
0.	0	1434: }	[1434] 1000057c4: <branch target>	<=====
0.	0		[1434] 1000057c4: retl	
0.	0		[1434] 1000057c8: nop	
0.	0	1435: void		
0.	0	1437: computeC(workStruct_t *x)		
0.	0	1438: {		
0.	0	<Function: computeC>		
0.	0		[1438] 100005800: <branch target>	<=====
0.	0		[1438] 100005800: sethi [%hi(0x100000), %05] %f4	
0.	0	1439: long long i;		
0.	0	1440: x->sum_ctr = 0;	[1440] 100005804: clr [%00] %f4	{structure:workStruct_t -}.{double sum_ctr}
0.	0		[1438] 100005808: or [%05, 263, %04] %f2	
0.	0		[1438] 10000580c: sllx [%04, 12, %02] %f2	

On SPARC systems, if you compiled with `-xhwcprof`, loads and stores are annotated with structure information showing that the instruction is referencing a double word, `sum_ctr` in the `workStruct_t` data structure. You also see lines with the same address as the next line, with `<branch target>` as its instruction. Such lines indicate that the next address is the target of a branch, which means the code might have reached an instruction that is indicated as hot without ever executing the instructions above the `<branch target>`.

On x86 systems, the loads and stores are not annotated and `<branch target>` lines are not displayed because the `-xhwcprof` is not supported on x86.

5. Go back and forth between the Functions and Disassembly views, selecting various `compute*()` functions.

Note that for all `compute*()` functions, the instructions with high counts for Instructions Executed reference the same structure field.

You have now seen that `computeB()` takes much longer than the other functions even though it executes the same number of instructions, and is the only function that gets cache misses. The cache misses are responsible for the increased number of cycles to execute the instructions because a load with a cache miss takes many more cycles to complete than a load with a cache hit.

For all the `compute*()` functions except `computeB()`, the double word field `sum_ctr` in the structure `workStruct_t` which is pointed to by the argument from each thread, is contained within the `workblk` for that thread. Although the `workblk` structures are allocated contiguously, they are large enough so that the double words in each structure are too far apart to share a cache line.

For `computeB()`, the `workStruct_t` arguments from the threads are consecutive instances of that structure, which is only one double-word long. As a result the double-words used by the different threads will share a cache line, which causes any store from one thread to invalidate the cache line in the other threads. That is why the cache miss count is so high, and the delay refilling the cache line is why the Total CPU Time and CPU Cycles Metric is so high.

In this example, the data words being stored by the threads do not overlap although they share a cache line. This performance problem is referred to as "false sharing". If the threads were referring to the same data words, that would be true sharing. The data you have looked at so far do not distinguish between false and true sharing.

The difference between false and true sharing is explored in the last section of this tutorial.

Detecting False Sharing

This part of the tutorial is applicable only to systems where the L1 D-Cache Miss `dcm` counter is precise. Such systems include SPARC-T4, SPARC-T5, SPARC-M5 and SPARC-M6, among others. If your experiment was recorded on a system without a precise `dcm` counter, this section does not apply.

This procedure shows how to use Index Object views and Memory Object views along with filtering.

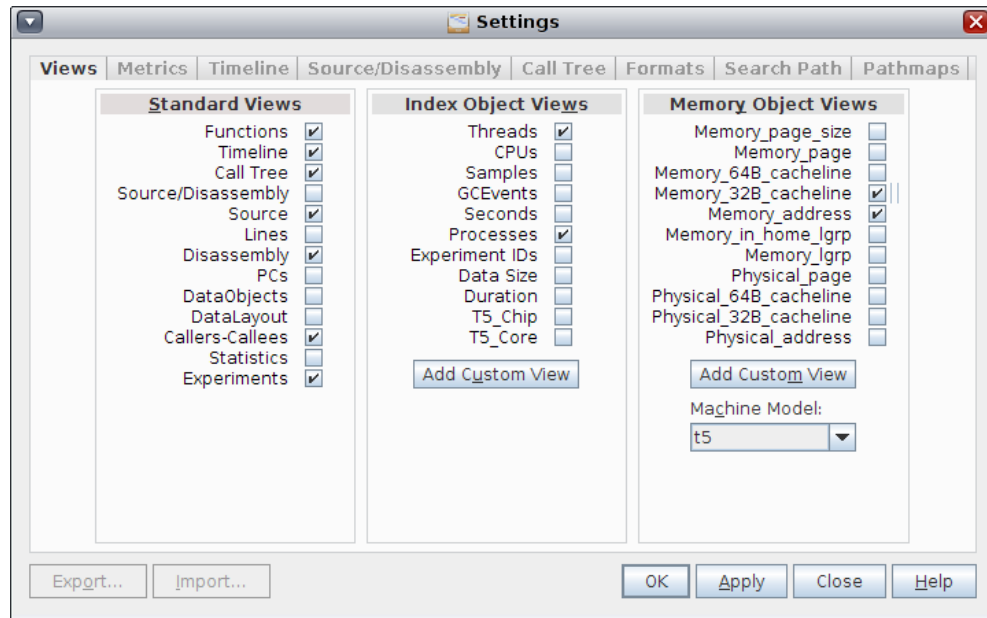
When you create an experiment on a system with precise memory-related counters, a *machine model* is recorded in the experiment. The machine model represents the mappings of addresses to the various components in the memory subsystem of that machine. When you load the experiment in Performance Analyzer or `er_print`, the machine model is automatically loaded.

The experiment used for the screen shots in this tutorial was recorded on a SPARC T5 system and the `t5` machine model for that machine is automatically loaded with the experiment. The machine model adds data views of index objects and memory objects.

1. Go to the Functions view and select `computeB()`, then right-click and select Add Filter: Include only stacks containing the selected functions.

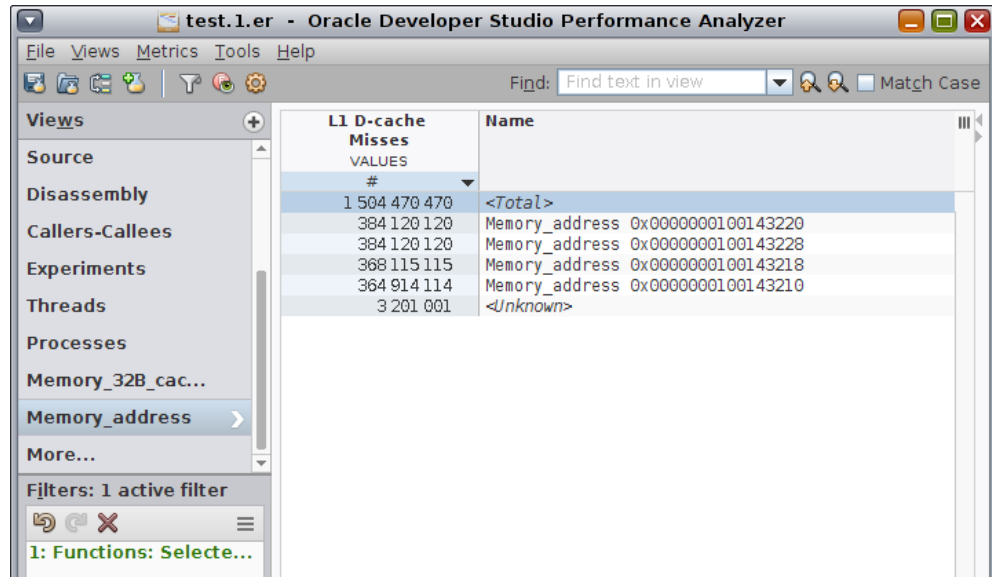
By filtering, you can focus on the performance of the `computeB()` function and the profile events occurring in that function.

2. Click the Settings button in the tool bar or choose Tools → Settings to open the Settings dialog, and select the Views tab in that dialog.



The panel on the right is labeled Memory Objects Views and shows a list of data views that represent the SPARC T5 machine's memory subsystem structure.

3. Select the check boxes for `Memory_address` and `Memory_32B_cacheline` and click OK.
4. Select the `Memory_address` view in the Views navigation panel.



In this experiment you can see that there are four different addresses getting the cache misses.

5. Select one of the addresses and then right-click and choose Add Filter: Include only events with the selected item.
6. Select the Threads view.

Total CPU Time	L1 D-cache Misses	Name
VALUES	VALUES	
sec.	#	
70.940	384120120	<Total>
19.874	384120120	Process 1, Thread 10
19.874	0	Process 1, Thread 11
15.601	0	Process 1, Thread 12
15.591	0	Process 1, Thread 13

As you can see in the preceding screen shot, only one thread has cache misses for that address.

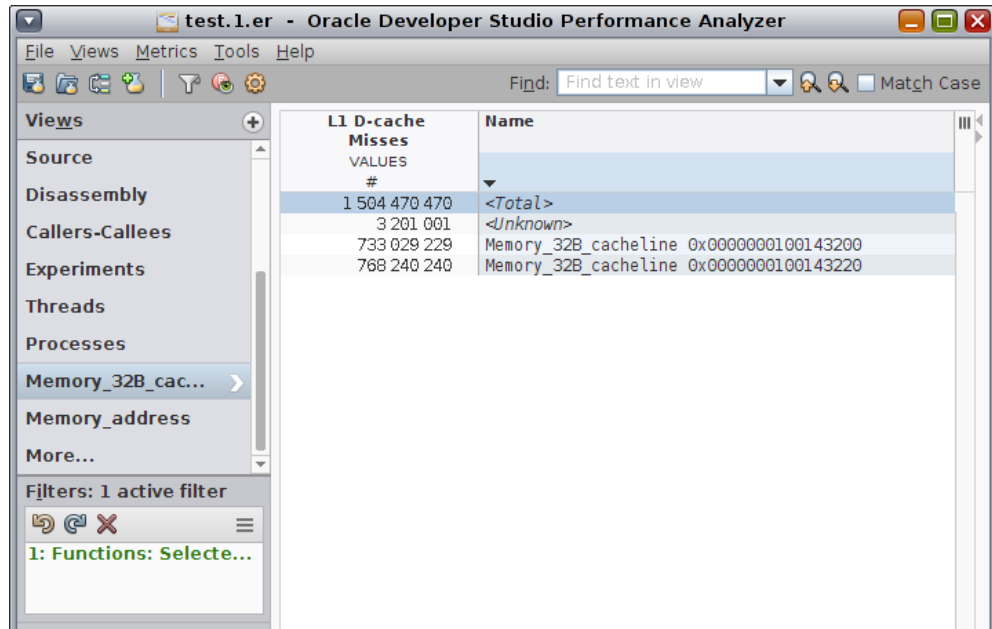
- Remove the address filter by right-clicking in the view and selecting Undo Filter Action from the context menu.

You can alternatively use the Undo Filter Action button in the Active Filters panel to remove the filter.

- Return to the Memory_address view, and select and filter on other addresses and check the associated thread in the Threads view.

By filtering and unfiltering and by switching between the Memory_address and Threads views in this manner, you can confirm that there is a one-to-one relationship between the four threads and the four addresses. That is, the four threads do not share addresses.

- Select the Memory_32B_cacheline view in the Views navigation panel.



Confirm in the Active Filters panel that there is only the filter active on the function `computeB()`. The filter is shown as `Functions: Selected Functions`. None of the filters on addresses should be active now.

You should see that there are two 32-byte cache lines getting the cache misses of the four threads and their four respective addresses. This confirms that although you saw earlier that the four threads do not share addresses, you see here that they do share cache lines.

False sharing is a very difficult problem to diagnose, and the SPARC T5 chip, along with Oracle Developer Studio Performance Analyzer enables you to do so.

Synchronization Tracing on a Multithreaded Program

This tutorial includes the following topics.

- [“About the Synchronization Tracing Tutorial” on page 95](#)
- [“Setting Up the `mttest` Sample Code” on page 97](#)
- [“Collecting Data from `mttest` for Synchronization Tracing Tutorial” on page 98](#)
- [“Examining the Synchronization Tracing Experiment for `mttest`” on page 98](#)

About the Synchronization Tracing Tutorial

This tutorial shows how to use Performance Analyzer on a multithreaded program to examine clock profiling and synchronization tracing data.

You use the Overview page to quickly see which performance metrics are highlighted and change which metrics are shown in data views. You use the Functions view, Callers-Callees view, and the Source view to explore the data. The tutorial also shows you how to compare two experiments.

The tutorial helps you understand synchronization tracing data, and explains how to relate it to clock-profiling data.

The data you see in the experiment that you record will be different from that shown here. The experiment used for the screen shots in the tutorial was recorded on a SPARC T5 system running Oracle Solaris 11.3. The data from an x86 system running Oracle Solaris or Linux will be different. Furthermore, data collection is statistical in nature and varies from experiment to experiment, even when run on the same system and OS.

The Performance Analyzer window configuration that you see might not precisely match the screen shots. Performance Analyzer enables you to drag separator bars between components of the window, collapse components, and resize the window. Performance Analyzer records its configuration and uses the same configuration the next time it runs. Many configuration changes were made in the course of capturing the screen shots shown in the tutorial.

About the `mttest` Program

The program `mttest` is a simple program that exercises various synchronization options on dummy data. The program implements a number of different tasks and each task uses the same basic algorithm:

- Queue up a number of work blocks (4, by default).
- Spawn a number of threads to process them (also, 4, by default).
- In each task, use a particular synchronization primitive to control access to the work blocks.
- Process the work for the block, after the synchronization.

Each task uses a different synchronization method. The `mttest` code executes each task in sequence.

About Synchronization Tracing

Synchronization tracing is implemented by interposing on the various library functions for synchronization, such as `mutex_lock()`, `pthread_mutex_lock()`, `sem_wait()`, and so on. Both the `pthread` and Oracle Solaris synchronization calls are traced.

When the target program calls one of these functions, the call is intercepted by the data collector. The current time, the address of the lock, and some other data is captured, and then the interposition routine calls the real library routine. When the real library routine returns, the data collector reads the time again and computes the difference between the end-time and the start-time. If that difference exceeds a user-specified threshold, the event is recorded. If the time does not exceed the threshold, the event is not recorded. In either case, the return value from the real library routine is returned to the caller.

You can set the threshold used to determine whether to record the event by using the `collect` command's `-s` option. If you use Performance Analyzer to collect the experiment, you can specify the threshold as the Minimum Delay for Synchronization Wait Tracing in the Profile Application dialog. You can set the threshold to a number of microseconds or to the keyword `calibrate` or `on`. When you use `calibrate` or `on` the data collector determines the time it takes to acquire an uncontended mutex lock and sets the threshold to five times that value. A specified threshold of `0` or `all` causes all events to be recorded.

In this tutorial, you record synchronization wait tracing in two experiments, with one experiment having a calibrated threshold and one experiment with a zero threshold. Both experiments also include clock profiling.

Setting Up the `mttest` Sample Code

Before You Begin:

See the following for information about obtaining the code and setting up your environment.

- [“Getting the Sample Code for the Tutorials” on page 10](#)
- [“Setting Up Your Environment for the Tutorials” on page 11](#)

You might want to go through the introductory tutorial in [“Introduction to C Profiling”](#) first to become familiar with Performance Analyzer.

This tutorial uses the same `mttest` code as the tutorial [“Hardware Counter Profiling on a Multithreaded Program”](#). You should make a separate copy for this tutorial.

1. Copy the contents of the `mttest` directory to your own private working area with the following command:

```
% cp -r OracleDeveloperStudio12.6-Samples/PerformanceAnalyzer/mttest directory
```

Replace *directory* with the working directory you are using.

2. Change to that working directory copy.

```
% cd directory/mttest
```

3. Build the target executable.

```
% make clobber
```

```
% make
```

Note - The `clobber` subcommand is only needed if you ran `make` in the directory before, but safe to use in any case.

After you run `make` the directory contains the target application to be used in the tutorial, a C program called `mttest`.

Tip - If you prefer, you can edit the `Makefile` to do the following: use the GNU compilers rather than the default of the Oracle Developer Studio compilers; build in 32-bits rather than the default of 64-bits; and add different compiler flags.

Collecting Data from `mttest` for Synchronization Tracing Tutorial

The easiest way to collect the data is to run the following command in the `mttest` directory:

```
% make syncperf
```

The `syncperf` target of the Makefile launches the `collect` command twice and creates two experiments.

Note - This tutorial might take a longer time compiling and collecting data than the previous introductory tutorials.

The two experiments are named `test.1.er` and `test.2.er` and each contains synchronization tracing data and clock profile data. For the first experiment, `collect` uses a calibrated threshold for recording events by specifying the `-s on` option. For the second experiment, `collect` sets the threshold to zero to record all events by specifying the `-s all` option. In both experiments, clock-profiling is enabled through the `-p on` option.

Examining the Synchronization Tracing Experiment for `mttest`

This section shows how to explore the data in the experiments you created from the `mttest` sample code in the previous section.

Start Performance Analyzer from the `mttest` directory and load the first experiment as follows:

```
% analyzer test.1.er
```

When the experiment opens, Performance Analyzer shows the Overview page.

Experiment(s)
test.1.er

Metrics
Select the metrics to display in the data views, then click a data view in the navigation panel on the left.

Available Metrics * Hot Reset Clear All
Experiment Duration: 92.855 Seconds

▼ Clock Profiling
Total Thread Time: 334.474 Seconds

Metric	Percentage	Hot	Selected
Total Thread Time	100%		<input type="checkbox"/>
Total CPU Time	59%	*	<input checked="" type="checkbox"/>
Data Page Fault Time	0%		<input type="checkbox"/>
Text Page Fault Time	0%		<input type="checkbox"/>
Kernel Page Fault Time	0%		<input type="checkbox"/>
Stopped Time	0%		<input type="checkbox"/>
Wait CPU Time	0%		<input type="checkbox"/>
Sleep Time	24%		<input type="checkbox"/>
User Lock Time	17%		<input type="checkbox"/>

▼ Synchronization Tracing
Sync Wait Time: 137.687 Seconds * Hot
Sync Wait Count: 60

Metrics Preview

Total CPU Time		Sync Wa...	Sync ...	Name
EXCLUSIVE	INCLUSIVE	INCLUSIVE	INCLUSIVE	
sec.	sec.	sec.	#	
196.788	196.788	137.687	60	<Total>

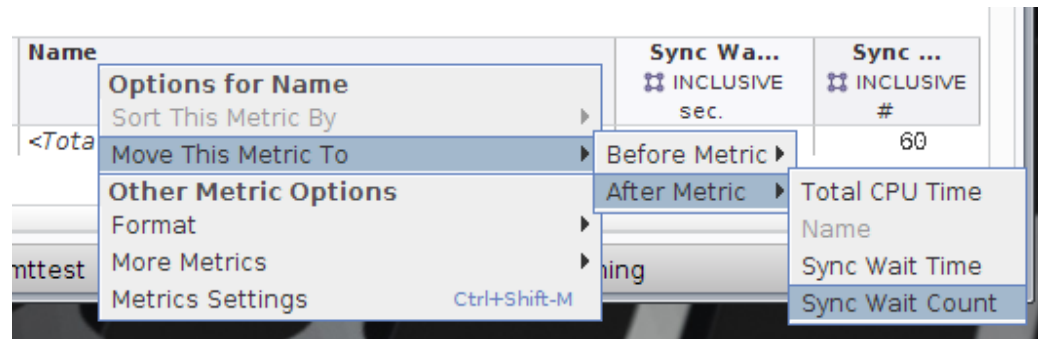
Local Host: Remote Host: Working Directory: .../mttest Compare: off Filters: off Warning

Clock Profiling metrics are shown first. On Oracle Solaris, Clock Profiling will show colored bars. Most of the thread time is spent in User CPU Time. Some time is spent in Sleep Time or User Lock Time. On Linux, only Total CPU time is shown.

Synchronization Tracing metrics are shown in a second group that includes two metrics, Sync Wait Time and Sync Wait Count.

Note - If you do not see the Sync Wait Time and Sync Wait Count metrics, you might have to scroll to the right to see the columns. You can move any column in a more convenient location by right-clicking the metric column header, selecting Move This Metric, and choosing a convenient location for you to see the metrics in relation to the other metrics.

The following example moves the Name column after the Sync Wait Count metric.



You can explore these metrics and their interpretation in the following sections of the tutorial.

Understanding Synchronization Tracing

This section explores the synchronization tracing data and explains how to relate it to clock-profiling data.

1. Go to the Functions view and sort according to inclusive Total CPU Time by clicking the column header Inclusive Total CPU.
2. Select the `do_work()` function at the top of the list.

The screenshot displays a performance analysis tool interface. The main table shows CPU time and sync wait times for various functions. The 'do_work' function is highlighted, showing a total CPU time of 0.660 seconds and a sync wait time of 56.814 seconds. The 'Called-by / Calls' panel at the bottom shows that 'do_work' is called by '_lwp_start' and 'locktest', and it calls ten other functions: 'cache_trash', 'trylock_global', 'cond_timeout_global', 'calladd', 'cond_global', 'computeE', 'computeI', 'computeH', 'computeF', and 'computeA'.

Total CPU Time		Sync Wait Time		Name
EXCLUSIVE sec.	INCLUSIVE sec.	INCLUSIVE sec.	INCLUSIVE #	
196.788	196.788	137.687	60	<Total>
0.660	196.788	56.814	23	do_work
0.	184.819	56.814	22	_lwp_start
0.	70.519	0.	0	cache_trash
70.519	70.519	0.	0	computeB
0.	35.435	0.	0	cache_trash_odd
0.	35.085	0.	0	cache_trash_even
0.741	29.891	0.	0	trylock_global
4.223	17.192	0.	0	mutex_trylock
0.	12.969	0.	0	do_exit_critical
12.969	12.969	0.	0	take_deferred_direct
11.978	11.978	0.	0	computeH
0.	11.978	17.951	11	cond_timeout_global
0.	11.968	80.872	38	_start
0.	11.968	0.	0	calladd
11.968	11.968	0.	0	computeA
2.642	11.968	0.	0	computeF
11.968	11.968	0.	0	computeG
0.	11.968	17.938	3	cond_global
0.	11.968	0.	0	lock_none
0.	11.968	80.872	37	locktest
0.	11.968	80.872	38	main
11.958	11.958	0.	0	compute
11.958	11.958	0.	0	computeD
11.958	11.958	0.	0	computeE
11.958	11.958	0.	0	computeI

Selection Details

Name: do_work
PC Address: 2:0x00004B50
Size: 360
Source File: mttest.c
Object File: ind as test.1.er/archives/mttest_gV8vCyH
Load Object: mttest (found as test.1.er/archives/mtt
angled Name:
Aliases:

	Exclusive	Inclusive
Total Thread Time:	0.660 (0.20%)	253.597 (75
Total CPU Time:	0.660 (0.34%)	196.788 (100
User CPU Time:	0.660 (0.34%)	196.788 (100
System CPU Time:	0. (0. %)	0. (0
Trap CPU Time:	0. (0. %)	0. (0
Data Page Fault Time:	0. (0. %)	0. (0
Text Page Fault Time:	0. (0. %)	0. (0
Kernel Page Fault Time:	0. (0. %)	0. (0
Stopped Time:	0. (0. %)	0. (0
Wait CPU Time:	0. (0. %)	0. (0
Sleep Time:	0. (0. %)	0. (0
User Lock Time:	0. (0. %)	56.810 (100
Sync Wait Time:	0. (0. %)	56.814 (41
Sync Wait Count:	0 (0. %)	23 (38

Called-by / Calls

Total CP... ATTRIBUTED sec.	do_work is called by	Total C... ATTRIBUTED sec.	do_work calls
184.819	_lwp_start	70.519	cache_trash
11.968	locktest	29.891	trylock_global
		11.978	cond_timeout_global
		11.968	calladd
		11.968	cond_global

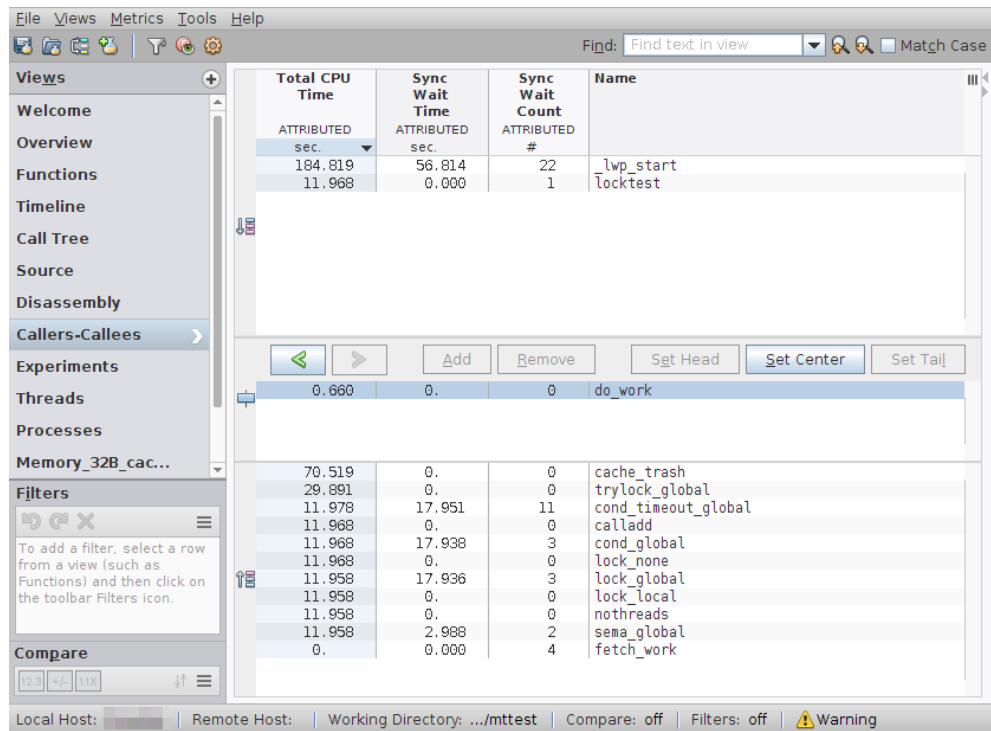
- Look at the Called-by/Calls panel at the bottom of the Functions view and note that `do_work()` is called from two places, and it calls ten functions.

Most often, `do_work()` is called when a thread to process the data is created, and is shown as called from `_lwp_start()`. In one case, `do_work()` calls one single-threaded task called `nothreads()` after being called from `locktest()`.

The ten functions that `do_work()` calls represent ten different tasks, and each task uses a different synchronization method that the program executed. In some experiments created from `mttest` you might see an eleventh function which uses relatively little time to fetch the work blocks for the other tasks. This function `fetch_work()` is displayed in the Calls panel in the preceding screen shot.

Note that except for the first two of the callees in the Calls panel, all callees show approximately the same amount of time (~10.6 seconds) of Attributed Total CPU.

- Switch to the Callers-Callees view.




Callers-Callees view shows the same callers and callees as the Called-by/Calls panel, but it also shows the other metrics that were selected in the Overview page, including Attributed Sync Wait Time.

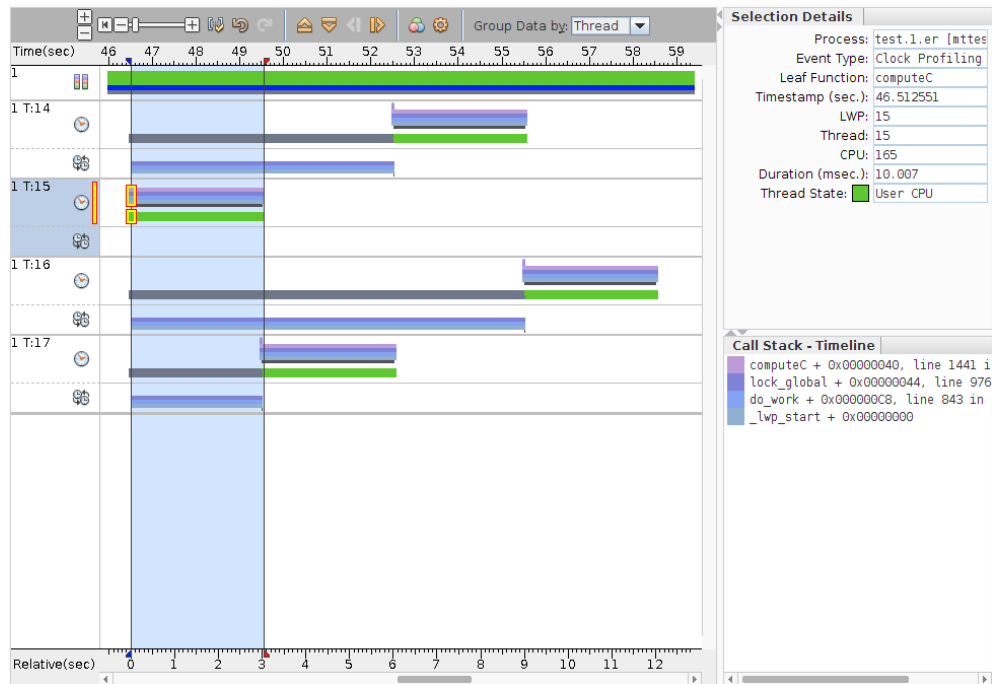
Look for the two functions `lock_global()` and `lock_local()`, and note that they show about the same amount of Attributed Total CPU time, but very different amounts of Attributed Sync Wait Time.

5. Select the `lock_global()` function and switch to Source view.


Total CPU Time	Sync Wait Time	Sync Wait Count	mtttest.c
INCLUSIVE sec.	INCLUSIVE sec.	INCLUSIVE #	
0.	0.	0	956. /* lock_global: use a global lock to process array's data */
			957. void
			958. lock_global(Workblk *array, struct scripttab *k)
			<Function: lock_global>
0.	0.	0	959. {
			960. /* acquire the global lock */
			961.
			962. #ifdef SOLARIS
			963. mutex_lock(&global_lock);
			964. #endif
0.	17.954	3	965. #ifdef POSIX
			966. pthread_mutex_lock(&global_lock);
			967. #endif
			968.
0.	0.	0	969. array->ready = gethrtime();
0.	0.	0	970. array->vready = gethrvtime();
			971.
0.	0.	0	972. array->compute_ready = array->ready;
0.	0.	0	973. array->compute_vready = array->vready;
			974.

Note that all the Sync Wait time is on the line with the call to `pthread_mutex_lock (&global_lock)` which has 0 Total CPU Time. As you might guess from the function names, the four threads executing this task all do their work when they acquire a global lock, which they acquire one by one.

6. Go back to the Functions view and select `lock_global()`, then click the Filter icon  and select Add Filter: Include only stacks containing the selected functions.
7. Select the Timeline view and you should see four threads.
8. Zoom into the areas of interest by highlighting the region in the timeline where the events happen, right-clicking, and selecting Zoom → Zoom to Selected Time Range.
9. Examine the four threads and the times spent waiting versus computing.



Note - Your experiment might have different threads executing and waiting at different times.

The first thread to get the lock (T:15 in the screen shot) works for ~2.97 seconds, then gives up the lock. If your application was run on Oracle Solaris, you can see that the thread state bar was spent in User CPU Time (green), with none in User Lock Time (grey). Notice also that the second bar for Synchronization Tracing Call Stacks marked with the  show no call stacks for this thread.

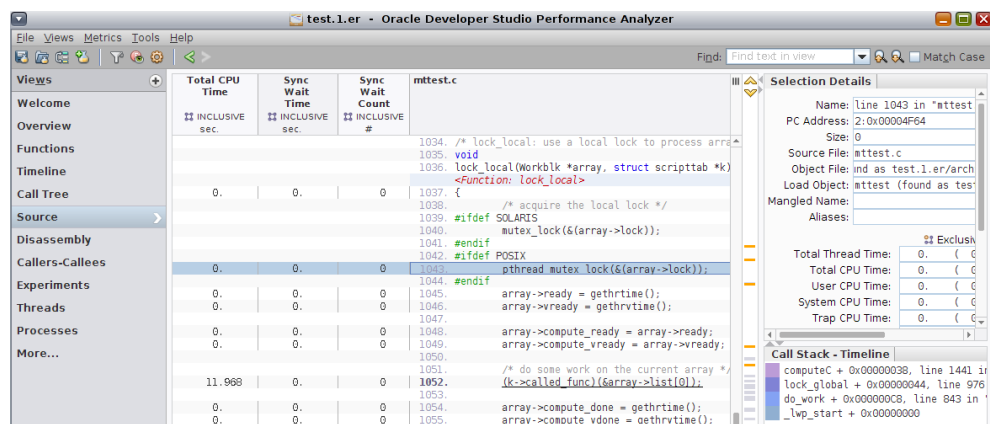
The second thread (T:17 in the screen shot) has waited 2.99 seconds in User Lock Time, and then it computes for ~2.90 seconds and gives up the lock. The Synchronization Tracing Call Stacks coincide with the User Lock Time.

The third thread (T:14) has waited 5.98 seconds in User Lock Time and it then computes for ~2.95 seconds, and gives up the lock.

The last thread (T:16) has waited 8.98 seconds in User Lock Time, and it computes for 2.84 seconds. The total computation was $2.97+2.90+2.95+2.84$ or ~ 11.7 seconds.

The total synchronization wait was $2.99 + 5.98 + 8.98$ or ~ 17.95 seconds, which you can confirm in the Functions view (which reports 17.954 seconds).

10. Remove the filter by clicking the X in the Active Filters panel.
11. Go back to the Functions view, select the function `lock_local()`, and switch to the Source view.




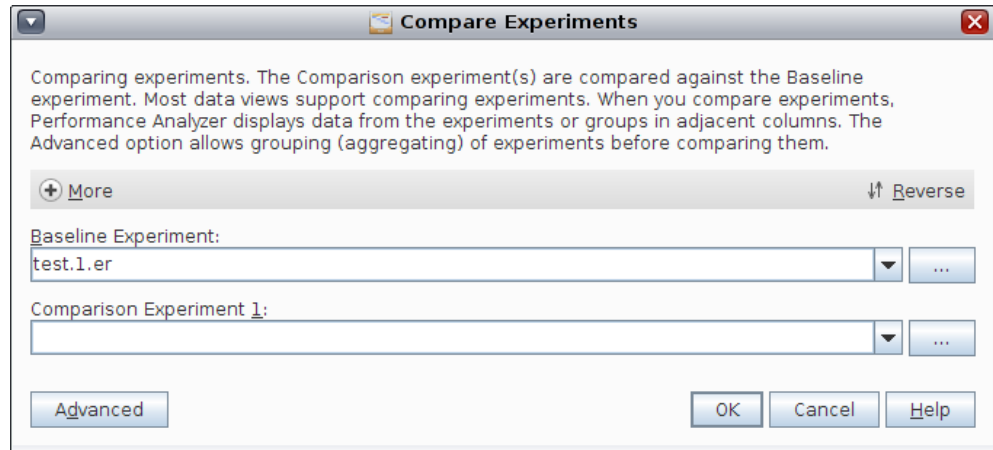
Note that the Sync Wait Time is 0 on the line with the call to `pthread_mutex_lock(&array->lock)`, line 1043 in the screen shot. This is because the lock is local to the workbook, so there is no contention and all four threads compute simultaneously.

The experiment you looked at was recorded with a calibrated threshold. In the next section, you compare to a second experiment which was recorded with zero threshold when you ran the `make` command.

Comparing Two Experiments with Synchronization Tracing

In this section you compare the two experiments. The `test.1.er` experiment was recorded with a calibrated threshold for recording events, and the `test.2.er` experiment was recorded with zero threshold to include all synchronization events that occurred in the `mttest` program execution.

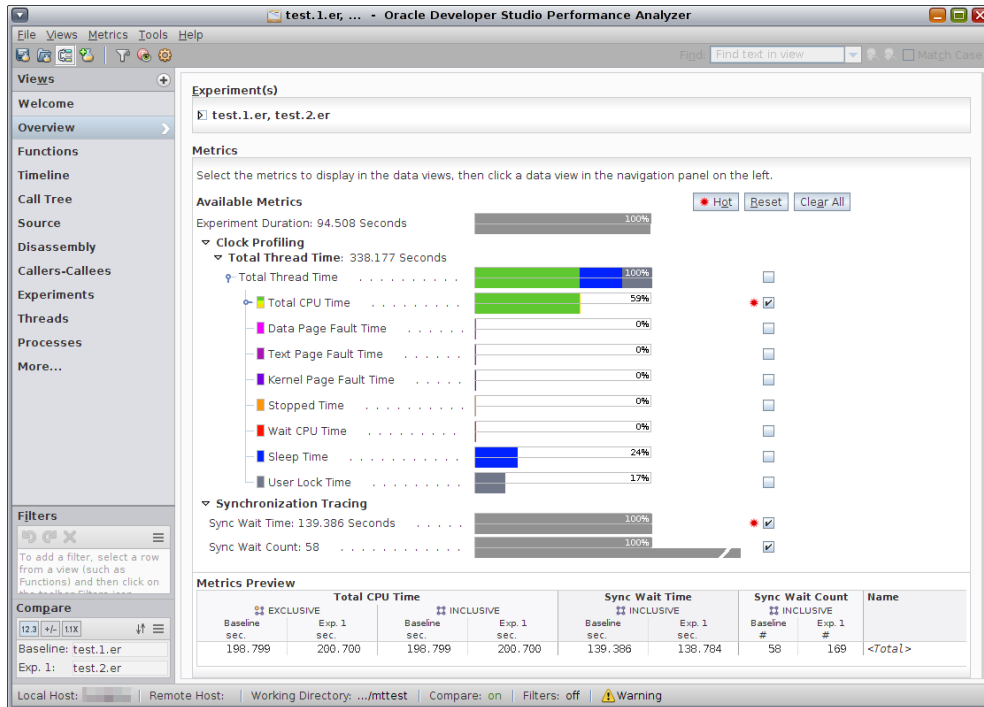
1. Click the Compare Experiments button  on the tool bar or choose File → Compare Experiments.
The Compare Experiments dialog box opens.



The test . 1 . er experiment that you already have open is listed in the Baseline group. You must add experiments to compare to the baseline experiment in the Comparison Group panel.

For more information about comparing experiments and adding multiple experiments to compare against the baseline, click the Help button in the dialog box.

2. Click the ... button next to Comparison Experiment 1, and open the test . 2 . er experiment in the Select Experiment dialog.
3. Click OK in the Compare Experiments dialog to load the second experiment.
The Overview page reopens with the data of both experiments included.



The Clock Profiling metrics display two colored bars for each metric, one bar for each experiment. The data from the test.1.er Baseline experiment is on top.

If you move the mouse cursor over the data bars, pop-up text shows the data from the Baseline and Comparison groups and difference between them in numbers and percentage.

Note that the Total CPU Time recorded is a little larger in the second experiment, but there are almost three times as many Sync Wait Counts.

- Switch to the Functions view, and click the subcolumn header labeled Baseline under the Inclusive Sync Wait Count column to sort the functions by the number of events in the first experiment.

Total CPU Time				Sync Wait Time		Sync Wait Count		Name
EXCLUSIVE		INCLUSIVE		INCLUSIVE		INCLUSIVE		
Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline #	Exp. 1 #	
198.799	200.700	198.799	200.700	139.386	138.784	58	169	<Total>
0.560	0.570	198.799	200.690	56.878	56.896	21	131	do_work
0.	0.	186.811	188.702	56.878	56.896	21	126	_lwp_start
0.	0.	0.	0.	0.000	0.003	1	77	fetch_work
0.	0.	11.988	11.988	82.507	81.888	37	43	_start
0.	0.	11.988	11.988	82.507	81.888	37	43	main
0.	0.	11.988	11.988	82.507	81.888	36	41	locktest
0.	0.	0.	0.	82.507	81.888	36	36	thread_work
0.	0.	11.988	11.988	17.969	17.972	11	27	cond_timeout_global
0.	0.	11.978	11.978	17.964	17.967	3	15	cond_global
0.	0.	11.968	11.978	17.954	17.961	3	4	lock_global
0.	0.	11.968	11.968	0.	0.	0	4	lock_local
0.	0.	11.968	11.968	2.991	2.992	3	4	sema_global
0.	0.	0.	0.	0.000	0.000	1	2	resolve_symbols
0.	0.	0.010	0.	0.	0.	0	0	__collector_write_packet
0.	0.	0.	0.010	0.	0.	0	0	__cond_timedwait
0.	0.010	0.	0.010	0.	0.	0	0	__lwp_park
0.	0.	0.	0.	0.	0.	0	0	__lwp_wait

The largest discrepancy between test.1.er and test.2.er is in do_work(), which includes the discrepancies from all the functions it calls, directly or indirectly, including lock_global() and lock_local().

Tip - You can compare the discrepancies even more easily if you change the comparison format. Click the Settings button in the tool bar, select the Formats tab, and choose Deltas for the Comparison Style. After you apply the change, the metrics for test.2.er display as the + or - difference from the metrics in test.1.er. In the preceding screen shot, the selected pthread_mutex_lock() function would show +88 in the test.2.er Incl Sync Wait Count column.

5. Select Callers-Callees view.

Total CPU Time		Sync Wait Time		Sync Wait Count		Name
ATTRIBUTED		ATTRIBUTED		ATTRIBUTED		
Baseline	Exp. 1	Baseline	Exp. 1	Baseline	Exp. 1	
sec.	sec.	sec.	sec.	#	#	
186.811	188.702	56.878	56.896	21	126	lwp_start
11.988	11.988	0.	0.	0	5	locktest

0.568	0.570	0.	0.	0	0	do_work
11.998	11.998	17.969	17.972	11	27	cond_timeout_global
11.978	11.978	17.964	17.967	3	15	cond_global
11.968	11.978	17.954	17.961	3	4	lock_global
11.998	11.968	2.991	2.992	3	4	sema_global
0.	0.	0.000	0.003	1	77	fetch_work
72.481	74.322	0.	0.	0	0	cache_trash
11.978	11.968	0.	0.	0	0	calladd
11.968	11.968	0.	0.	0	4	lock_local
11.968	12.018	0.	0.	0	0	lock_none
11.978	11.978	0.	0.	0	0	nothreads
29.921	29.921	0.	0.	0	0	trylock_global

Look at two of the callers, `lock_global()` and `lock_local()`.

The `lock_global()` function shows 3 events for Attributed Sync Wait Count in `test.1.er`, but 4 events in `test.2.er`. The reason is that the first thread to acquire the lock in the `test.1.er` was not stalled, so the event was not recorded. In the `test.2.er` experiment the threshold was set to record all events, so even the first thread's lock acquisition was recorded.

Similarly, in the first experiment there were no recorded events for `lock_local()` because there was no contention for the lock. There were 4 events in the second experiment, even though in aggregate they had negligible delays.

Exploring More in Performance Analyzer

This chapter explores more tutorials and tasks you can do with Performance Analyzer, as well as where you can find more resources.

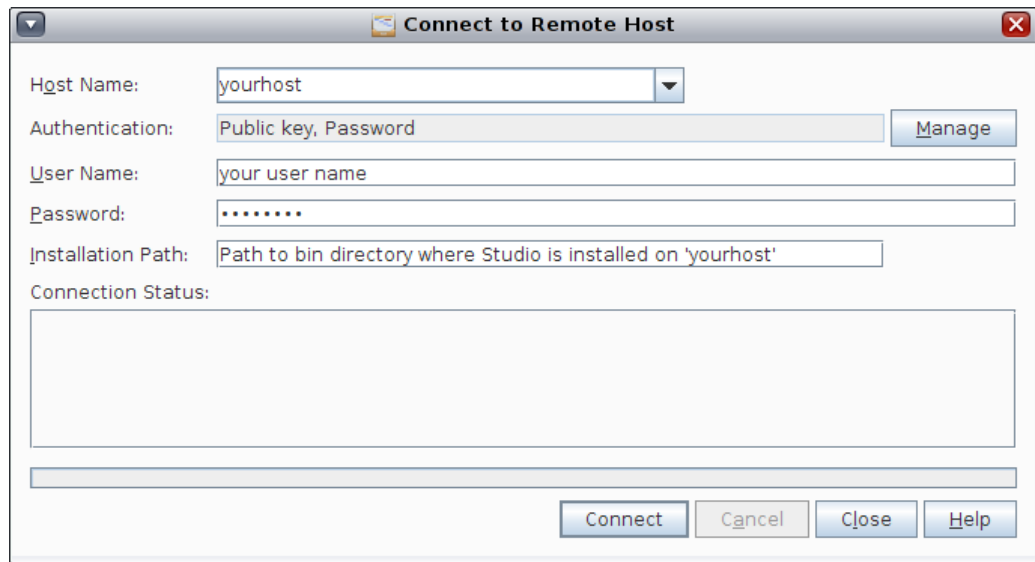
- [“Using the Remote Performance Analyzer” on page 111](#)
- [“More Information” on page 112](#)

Using the Remote Performance Analyzer

You can use the Remote Performance Analyzer either from a supported system, or from systems where Oracle Developer Studio cannot be installed, such as Mac OS or Windows. See [“Using Performance Analyzer Remotely” in *Oracle Developer Studio 12.6: Performance Analyzer*](#) for information about installing and using this special version of Performance Analyzer.

When you invoke Performance Analyzer remotely, you see the same Welcome page, but the options for creating and viewing experiments are disabled and grayed-out.

Click Connect to Remote Host and Performance Analyzer opens a connection dialog:



Type the name of the system to which you want to connect, your user name and password for that system, and the installation path to the Oracle Developer Studio installation on that system. Click Connect and Performance Analyzer logs in to the remote system using your name and password, and verifies the connection.

From that point on, the Welcome page will look just as it does with the local Performance Analyzer, except the status area at the bottom shows the name of the remote host to which you connected. Proceed from there in step 2 above.

More Information

The following resources give more information on Performance Analyzer and the related data-collection tools:

- Integrated help system in Performance Analyzer
- [Oracle Developer Studio 12.6: Performance Analyzer](#)
- Articles and white papers available on the Oracle Developer Studio developer portal (<http://www.oracle.com/technetwork/server-storage/developerstudio/>).