Oracle[®] Developer Studio 12.5: Performance Analyzer Tutorials



Part No: E60756 June 2016

Oracle Developer Studio 12.5: Performance Analyzer Tutorials

Part No: E60756

Copyright © 2015, 2016, 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=trs if you are hearing impaired.

Référence: E60756

Copyright © 2015, 2016, 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=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	55
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	77
About the Hardware Counter Profiling Tutorial	77
Setting Up the mttest Sample Code	78
Collecting Data From mttest for Hardware Counter Profiling Tutorial	79
Examining the Hardware Counter Profiling Experiment for mttest	80
Exploring Clock-Profiling Data	82
Understanding Hardware Counter Instruction Profiling Metrics	84
Understanding Hardware Counter CPU Cycles Profiling Metrics	86
Understanding Cache Contention and Cache Profiling Metrics	88
Detecting False Sharing	92
Synchronization Tracing on a Multithreaded Program	97
About the Synchronization Tracing Tutorial	97
About the mttest Program	98
About Synchronization Tracing	98
Setting Up the mttest Sample Code	99
Collecting Data from mttest for Synchronization Tracing Tutorial	100
Examining the Synchronization Tracing Experiment for mttest	100
Understanding Synchronization Tracing	102
Comparing Two Experiments with Synchronization Tracing	107
Exploring More in Performance Analyzer	113
Using the Remote Performance Analyzer	113
Additional Tutorials	114
More Information	115

Using This Documentation

- **Overview** Provides step-by-step instructions for using the Oracle Developer Studio 12.5 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.

8 Oracle Developer Studio 12.5: Performance Analyzer Tutorials • June 2016

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 in 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 j synprog 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.

- Go to the Oracle Developer Studio 12.5 Sample Applications page at the Oracle Developer Studio web page http://www.oracle.com/technetwork/server-storage/ solarisstudio.
- 2. Navigate to the downloads section of the Oracle Developer Studio web page.
- 3. Read the license from the link on the page and accept by selecting Accept.
- 4. 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 OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer directory.

Note that the directory includes some additional samples that are not described in this document: cachetest, ksynprog, omptest, and synprog. Each sample subdirectory includes a Makefile and a README file with instructions that you can use for some additional demonstrations of Performance Analyzer.

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 Chapter 5, "After Installing Oracle Developer Studio 12.5" in *Oracle Developer Studio 12.5: Installation Guide*.

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

12 Oracle Developer Studio 12.5: Performance Analyzer Tutorials • June 2016

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 113.

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 OracleDeveloperStudio12.5-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.

	🔄 Profile Application [lowfruit]
General Data to Collect Output]]
Target Program: * //expo	rt/home/demol/PerformanceAnalyzerTutorials/lowfruit/low
<u>A</u> rguments:	
Working Directory: /expo	rt/home/demol/PerformanceAnalyzerTutorials/lowfruit
En <u>v</u> ironment Variables:	
Target Input/Output:	○ Use External Terminal
Speciry Experiment	
Experiment Name: test.1	.er
Experiment Directory:	
Experiment Group:	
Advanced Experiment Settings	
Data Limit (MB):	Unlimited 🔽
Time Limit (Seconds):	Unlimited 👻
Archive Mode:	On 🔽 on
Follow Descendant Processes:	On on
Signal to Pause/Resume Collection:	Off off
	Start state: Paused Resumed
Preview Command:	
Pa <u>u</u> se <u>S</u> ample	Run Terminate Close Help

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 builtin 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.

🔽 🧧 Profile A	pplication [lowfruit]		X
General Data to Collect Output			
Specify the type of data to collect. As you select da	ta types, other incompati	ble data types are o	lisabled.
✓ Clock Profiling	Profiling Rate:	Normal (100 Hz)	on
Hardware Counter Profiling			
Selected Hardware Counters:			
Add Properties Remove			
🖌 Java Profiling			
Periodic Samples of Process Resource Utilization	Interval (sec.):	Normal	on
Manual Samples of Process Resource Utilization	Signal:	Off 🗖	off
🔲 I/O Tracing			
Heap Tracing			
Synchronization Wait Tracing	Minimum Delay (usec.):	Calibrate	r calibrate
MPI Tracing for specified MPI	MPI version:	OMPT 💌	OMPT
Data Race Detection			
Deadlock Detection			
Function/Instruction Counts	Instrumentation:	0n 🔻	r on
Draview Commands La test 1 or p C (we	art (hama (dama) (Darf		le (leu fruit (leu fruit
review Command:ro test.1.er -p on -S on /exp	ort/nome/demot/Performa	anceAnaiyzeri utoria	is/iowrruit/iowrruit
Pa <u>u</u> se <u>S</u> ample	Run	Terminate	<u>C</u> lose <u>H</u> elp

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.

	Profile Application [lowfruit]	X
General Data to Collector output Wed May 18 13:50: Wed May 18 13:51: Process ID: 27726 Elapsed Time: 0 ms Wed May 18 13:50: Creating experimer	2 Collect Output 41 PDT 2016: Running: lowfruit 43 PDT 2016: Execution completed, exit status is 0 41 2016 1t database test.1.er (Process ID: 27726)	
Process output Running init_good(Running init_bad() Running insert_goo Running insert_bad	Open Experiment Data collection is complete. Do you want to open the experiment? Experiment: /export/home/demo1/PerformanceAnalyzerTutorials/lowfruit/test.1.er Use this Configuration Always re-open this experiment with this Configuration OK Cancel	Clear
Preview Comman	d: lib///bin/collect -0 /tmp/collect_1463604641304_pid.txt -o test.1.4 nple Run T <u>e</u> rminate	er -p on -S on lowfruit <u>C</u> lose <u>H</u> elp

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.

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

		≦ test. 1. er	- Oracle Develo	per Studio Pe	rformance Ana	lyzer		= = 🛛
File Views Metrics	<u>T</u> ools <u>H</u> elp							
🛃 🖾 🕮 🕙 🛛 🏹 🤇	ē @ < ≻				Fig	nd: Find	text in view 🔽	💫 😡 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 😛	Total CI	PU Time	Name	III	Selection Det	ails		
Welcome	sec.	sec.			Name:	init_sta	tic_routine	
Overview	0.	62.103	main		PC Address:	2:0x0000	1180	
Functions	0. 0.	62.103	_start		Size:	208	-	
	2.862	8.936	insert_bad		Source File:	towrruit	.c	
Timeline	3.633	7.245	init_good		Object File:	nd as te	st.1.er/archives/low	fruit_CUOvY61FFe8)
Call Tree	3.033	39.818	incert number		Load Object:	lowfruit	(found as test.l.er	/archives/lowfruit
Call free	30 708	30 709	init static rout	ine	Mangled Name:			
Source	62.103	62.103	<total></total>	THE	Aliases:			
Disassembly							St Exclusive	# Inclusive
Callers-Callees					Total Threa	d Time:	39.808 (64.09%)	39.808 (64.09%)
- · · ·					Total CP	U Time:	39.798 (64.08%)	39.798 (64.08%)
Experiments					User CP	U Time:	39.798 (64.08%)	39.798 (64.08%)
Threads					System CP	U Time:	0. (0.%)	0. (0.%)
Processes					Data Baga Fou	U Time:	0. (0.%)	0. (0.%)
					Text Page Fau	it Time:	0. (0.%)	0. (0.%)
More					Kernel Page Fau	It Time:	0. (0.%)	0. (0.%)
					Stoppe	d Time:	0. (0.%)	0. (0.%)
	Called-by / C	alls			Wait CP	U Time:	0.010 (100.00%)	0.010 (100.00%)
		init_sta	atic_routine		Slee	p Time:	0. (0.%)	0. (0.%)
	Total C	init_static_ro.	Total ATTRIBUTED	init_static	User Loc	k Time:	0. (0.%)	0. (0.%)
	sec.	is called by	sec.	cans				
	3.613	init good						
Filters	36.185	init_bad						
bex =								
To add a filter, select								
a row from a view (such as Functions)								
Compare								
12.3 +/- 1.1X ↓↑	4			•				
Local Host:	Remote Host:	Working Dir	rectory: /lowfruit	Compare: off	Filters: off	🔔 Warnir	ng	8/9

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.

 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 \rightarrow 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.



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:

		[test. 1. er	- Oracle Developer Studio Performance	Analyzer	= = 🛛
File Views Metrics 1	ools <u>H</u> elp				
🛛 🖾 🛱 🖄 🛛 🏹 🤇	è (2) < >			Find: Find text in view	💌 👧 😡 💷 Mat <u>c</u> h Case
Vie <u>w</u> s 🔸	Total	lowfruit.c			III 🙈
Welcome	CPU TIME				V
Overview	Sec. O.	84.	$for(i = 0; i < 10; i ++) $ {		
Functions	0.	85.	for(j= 0; j < 50; j++) {		
Timeline	10.657	87.	for(k=0; k<1000000; k++) {		
Call Tree	29.140	88. 89.	x = x + 1.0; }		_
Source		90. 91.	}		
Disassembly	Θ.	92. }			
Callers-Callees		94. /* ===		*/	=
Experiments		95. int 96. int	<pre>*insert_table; count;</pre>		_
Threads		97. 98. void			
Processes		99. insert	bad(int insert_count)		
	0.	100. {			
More		101.	int i, done, new;		
	Ο.	102.	count = 0;		
	Θ.	103.	insert_table = insert_init(insert_count);		
	Ο.	105.	<pre>for(i = 0; i < insert count; i ++) {</pre>		
		106.	/* get a new element */		
	Ο.	107.	new = rand();		
	Θ.	108.	done = 0;		
	0.901	109.	for (int j = 0; j < count; j ++) {		
510	1.961	110.	if(new < insert_table[j]) {		
Fitters	6.074	111.	<u>insert_number(new,j);</u>		
bex =	0.	112.	done = 1;		
	Θ.	113.	break;		
To add a filter, select		114.	}		
(ouch as Eusctions)		115.	}		
(such as runctions)	0.	116.	1† (done == 0) {		
Compare	O.	117.	insert_number(new, count);		
		118.	}		•
12.3 +/- 1.1X V1 =					► E
Local Host:	Remote Host:	Working Di	rectory:/lowfruit Compare: off Filters: of	🕴 🔔 Warning	6/9

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().

		[test. 1. e	er - Oracle Developer Studio Performance Analy	zer	
<u>F</u> ile <u>V</u> iews <u>M</u> etrics <u>T</u>	ools <u>H</u> elp				
2 🖾 🕮 🖏 🍸 🤅) 🙆 < >		Fi <u>n</u> d	Find text in view	🔽 🔗 🔗 🗌 Mat <u>c</u> h Ca
Vie <u>w</u> s 🔶 Welcome	Total CPU Time	lowfruit.c			III 4
relevine	Sec				
Overview		131. void			*
unctions		132. inse <fun< td=""><td>rt_good(int insert_count) ction: insert good></td><td></td><td></td></fun<>	rt_good(int insert_count) ction: insert good>		
imeline	Θ.	133. {			
		134.	<pre>int i, x, done, new, left, right, curval;</pre>		
all Tree		135.			
	0.	136.	count = 0;		
Jource	Θ.	137.	insert_table = insert_init(insert_count);		
Disassembly	0	138.	for(i = 0, i < incort count, i ()) {		
,	0.	140	/* get a new element */		
allers-Callees	0	140.	new = rand();		
voorimonto	0.	142.			
xperiments		143.	/* do a binary search to find its position	*/	
hreads	Ο.	144.	left = 0;		
	Θ.	145.	right = count-1;		
rocesses		146.			
lore		147.	/* figure out where it belongs */		
1010	0.	148.	while (left <= right) {		
	0.	149.	x = (left + right) / 2;		
	0.	150.	curvat = insert_table[x];		
	0	152	if (curval > new) {		
	0.	153.	right = x - 1;		
	Ο.	154.	} else if (curval < new) {		
	Ο.	155.	left = x + 1;		
		156.	} else {		
iltoro		157.	/* a duplicate value */		
iners	Θ.	158.	left = X;		
bcx =	υ.	160	right = x -1;		
o add a filter, select		161	3		
row from a view	6.104	162.	insert number(new. left):		
such as Functions)	0.104	163.	}		
		164.	-		
.om <u>p</u> are		165. #if	9		-
12.3 +/- 11X JA =	4				Þ

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.

10. Click on the Timeline view or choose Views \rightarrow Timeline from the menu bar.

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 CPU 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 CPU 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 two bars of data for events occurring during program execution. The upper bar shows color-coded representations of the callstack and the lower bar shows the state of the thread at each event. The state in this example was always User CPU Time so it appears to be 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 the slider at the top of the Timeline, or using the + key, or by double-clicking with the mouse.

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.

		[test.	.1.er - Oracle Develo	per Studio Performance	Analyzer	
Eile ⊻iews Metrics	<u>T</u> ools <u>H</u> elp					
🛃 🖾 🕮 🕙 🛛 🏹 I	6 🙆				Find: Find	text in view 📃 🔍 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	E ME-I-	 	െ പറ കെക	Group Data by Thread	Selection Details	
Welcome				22.0	Process	fruit/test.l.er [lowfruit, PID 27726]
Overview	1 me(sec)	33.5	33.0 33.7 T	33.8	Event Type	Clock Profiling
e	- II				Leaf Function	init_static_routine
Functions	1 T:1				Imestamp (sec.)	1
Timeline >	$_{\odot}$				Thread	1
Call Tree					CPU	183
Source					Duration (msec.)	10.007
Disassembly					Thread State:	User CPU
Callers-Callees					Call Stack - Timelin	e
Experiments					init_static_routi	ne + 0x0000006C, line 88 in "lowfruit.c
Threads					main + 0x00000054	line 31 in "lowfruit.c"
Processes					_start + 0x000001	08

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

17. Click on the Call Tree view or choose Views \rightarrow 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.

Station Studio	Performance Analyzer		= = 🛛
<u>Eile ⊻iews Metrics T</u> ools <u>H</u> elp			
🖾 📾 🗳 7° 😔 🥹	Fi <u>n</u> d: Fir	nd text in view 🔽	🖌 💦 🗌 Mat <u>c</u> h Case
Views Call Tree: FUNCTIONS. Complete view. Threshold: 1% Sort	Selection Details		
Welcome start	Name: insert n	umber	-
Overview 9 main	PC Address: 2:0x0000	01680	
init_bad init_static_routine	Size: 228		
Functions	Source File: lowfruit	t.c	
Timeline init_static_routine	Object File: ind as te	est.l.er/archives/low	/fruit_CU0vY6TFFe8)
Call Tree	Load Object: lowfruit	t (found as test.l.er	/archives/lowfruit_
e insert good	Mangled Name:		
Sourceinsert_number	Aliases:		
Disassembly		Sector	# Inclusive
Callers Callees	Total Thread Time:	12.179 (19.61%)	12.179 (19.61%)
callets-callees	Total CPU Time:	12.179 (19.61%)	12.179 (19.61%)
Experiments	User CPU Time:	12.179 (19.61%)	12.179 (19.61%)
Threads	System CPU Time:	0. (0.%)	0. (0.%)
	Trap CPU Time:	0. (0.%)	0. (0.%)
Processes	Data Page Fault Time:	0. (0.%)	0. (0.%)
More	Text Page Fault Time:	0. (0.%)	0. (0.%)
	Kernel Page Fault Time:	0. (0.%)	0. (0.%)
	Stopped Time:	0. (0.%)	0. (0.%)

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.

30 Oracle Developer Studio 12.5: Performance Analyzer Tutorials • June 2016

Introduction to Java Profiling

This chapter covers the following topics.

- "About the Java Profiling Tutorial" on page 31
- "Setting Up the jlowfruit Sample Code" on page 32
- "Using Performance Analyzer to Collect Data from jlowfruit" on page 33
- "Using Performance Analyzer to Examine the jlowfruit 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 jlowfruit 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 113.

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

```
% cp -r OracleDeveloperStudio12.5-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

💽 🔄 Profile Application [java]
General Data to Collect Output
Specify Application to Profile
Iarget Program: * java
Arguments: -Xmx100m -XX:ParallelGCThreads=10 jlowfruit
Working Directory: /export/home/demol/PerformanceAnalyzerTutorials/jlowfruit
Environment Variables:
Target Input/Output: 🔾 Use External Terminal 💿 Use Built-in Output Window
Specify Experiment
Experiment Name: test.l.er
Experiment Directory:
Experiment Gr <u>o</u> up:
Advanced Experiment Settings
Data Limit (MB): Unlimited 🔽
Time Limit (Seconds): Unlimited 🔽
Archive Mode: On 💌 on
Follow Descendant Processes: On 💌 on
Signal to Pause/Resume Collection: Off 🛛 🗨 off
Start state: Paused Resumed
Preview Command:
Pause Sample Run Terminate Close Help

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 builtin 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.

🔽 🧧 Profile .	Application [java]		_	×
General Data to Collect Output				
Specify the type of data to collect. As you select dat	a types, other incompati	ble data types are	disabled.	
Clock Profiling	Profiling Rate:	Normal (100 Hz)	▼	on
Hardware Counter Profiling				
Selected Hardware Counters:				
Add Properties Remove				
🖌 Java Profiling				
Periodic Samples of Process Resource Utilization	Interval (sec.):	Normal	▼	on
Manual Samples of Process Resource Utilization	Signal:	Off	-	off
🗌 I/O Tracing				
Heap Tracing				
Synchronization Wait Tracing	Minimum Delay (usec.):	Calibrate	- ca	librate
MPI Tracing for specified MPI	MPI version:	OMPT	•	OMPT
Data Race Detection				
Deadlock Detection				
Function/Instruction Counts	Instrumentation:	On	-	on
Braview Command				
Ereview Command:				
Pa <u>u</u> se <u>S</u> ample	Run	T <u>e</u> rminate	<u>C</u> lose	<u>H</u> elp

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.

Using Performance Analyzer to Examine the jlowfruit Data

•	Profile Application [java]	2
General Data to Collector output Wed May 18 15:34:0 Wed May 18 15:35:2 Process ID: 4075 Elapsed Time: 0 ms Wed May 18 15:34:0 Creating experiment	Collect Output O3 PDT 2016: Running: java O3 PDT 2016: Execution completed, exit status is 0 O3 2016 t database test.1.er (Process ID: 4075)	
	🖸 🔤 Open Experiment 🛛 🔀	Clear
Process output Running init_good() Picked up JAVA_TOC Running init_bad() Running insert_goo Running insert_bad	Data collection is complete. Do you want to open the experiment? Experiment: /export/home/demo1/PerformanceAnalyzerTutorials/jlowfruit/test.1.er Use this Configuration As Previous Experiment Closed Always re-open this experiment with this Configuration OK Cancel	
Preview Command	: 0843629_pid.txt -o test.1.er -p on -S on java -Xmx100m -XX:ParallelGCThr	eads=10 jlowfruit
Pa <u>u</u> se <u>S</u> am	ple Run Terminate	<u>C</u> lose <u>H</u> elp

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.

	🔄 test. 1. er 🕒 Oracle Developer Studio	Performance Analyzer 📃 🗐 🛽
<u>F</u> ile <u>V</u> iews <u>M</u> etrics <u>T</u>	pols <u>H</u> elp	
🗷 🖾 🕮 🕙 🍸 🍕	View Mode: User	Fi <u>n</u> d: 📕 Find text in view 🔽 🔍 🔍 🗖 Mat <u>c</u> h Case
Vie <u>w</u> s 😛	Experiment(c)	
Welcome	<u>Experiment(s)</u>	
Overview	▷ test.l.er	
Functions	Metrics	
Timeline	Select the metrics to display in the data views, then click	a data view in the navigation panel on the left.
Call Tree	Available Metrics	* Hot Reset Clear All
Source	Experiment Duration: 78.383 Seconds	
Disassembly	Java Garbage Collection Duration: 0.000 Seconds	
Callers-Callees	✓ Total Thread Time: 546.913 Seconds	
Experiments	🖓 Total Thread Time 🛛 📃	100%
Throado	🕈 🧧 Total CPU Time 🛛 📃	14% 🗰 🖌
meaus	User CPU Time	14%
Processes	System CPU Time	0%
More	Trap CPU Time	0%
	Data Page Fault Time	0%
	Text Page Fault Time	0%
	Kernel Page Fault Time	0%
	Stopped Time	0%
	Wait CPU Time	0%
	Sleep Time	14%
	User Lock Time	71%
F <u>i</u> lters		
bex ≡	Metrics Preview	
To add a filter, select	Total CPU Time Name	
a row from a view (such as Eunctions)	SEC SEC	
(such as ranctions)	78.515 78.515 <total></total>	
Compare		
12.3 +/- 1.1X ↓↑ Ξ		
Local Host:	Remote Host: Working Directory:/jlowfruit Compar	e: off Filters: off 🔥 Warning

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 \rightarrow Threads from the menu bar.

		🚰 test.1.er 🔸 Oracle Developer Studio Performance Analyzer			= = 🛛
Eile ⊻iews Metrics	<u>T</u> ools <u>H</u> elp				
R 🖾 🛱 💙 🖓	🚱 🙆 View	Mode: User 🔍	Fin	d: Find text in view	💌 💫 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total	Name	Ш	Selection Details	
Welcome	VALUES		Ľ	Index Object: Process 1,	Thread 2, JThread 3 'r
Overview	sec.	<total></total>			\$\$ Exclusive
Eupetions	78.325	Process 1, Thread 2, JThread 3 'main', Group 'main', Parent 'system'	11	Total Thread Time:	78.345 (14.32%)
Functions	0.050	Process 1, Thread 17		Total CPU Time:	78.325 (99.76%)
Timeline	0.040	Process 1, Thread 20		User CPU Time:	78.255 (99.86%)
Call Tree	0.030	Process 1, Thread 18		System CPU Time:	0.070 (50.00%)
cui nec	0.020	Process 1, Thread 13		Trap CPU Time:	0. (0.%)
Source	0.010	Process 1, Thread 16, JThread 2 'Signal Dispatcher', Group 'system', Parent '	1	Data Page Fault Time:	0. (0.%)
Disassembly	0.010	Process 1, Thread 19		Text Page Fault Time:	0. (0.%)
	0.	Process 1, Inread 15, Jinread 1 , Group , Parent		Kernel Page Fault Time:	0. (0.%)
Callers-Calle				Stopped Time:	0. (0.%)
Experiments				Wait CPU Time:	0. (0.%)
There do				Sleep Time:	0.010 (0.01%)
Inreads				User Lock Time:	0.010 (0.00%)
Processes					
More					

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".

5. Click on the Functions view in the Views navigation panel, or choose Views \rightarrow Functions from the menu bar.

			test.1.er - Oracle Developer	Studio Performance Analyzer					
Eile ⊻iews Metrics	<u>T</u> ools <u>H</u> elp								
🛛 📾 😂 🖉 🖓	G CA CA View	Mode: User	- <>		Find	Find text in view		Mato	h Cas
		1103201							
Vie <u>w</u> s (+	Total CF	PU Time	Name		III (Selection Details			
Welcome	Sec.	54 INCLUSIVE				Name: <total:< th=""><th></th><th></th><th></th></total:<>			
	78,515	78,515	<total></total>			PC Addroses 1, 0y00	00000		
Overview	33.714	33,714	ilowfruit.init static routine()			PC Address: 1:0x00	100000		
Eunctions	30.171	30.171	jlowfruit.insert_number(int, int	1)		Size: 0			
Tunctions /	11.048	26.388	jlowfruit.insert_bad(int)			Source File: (unknow	/n)		
Timeline	1.531	32.173	jlowfruit.init_bad(int)			Object File: <total:< td=""><td></td><td></td><td></td></total:<>			
	1.531	4.603	jlowfruit.init_good(int)			Load Object: <total:< td=""><td>*</td><td></td><td></td></total:<>	*		
Call free	0.330	0.330	<jvm-system></jvm-system>			Mangled Name:			
Source	0.000	0.090	jova util Pandon povtTot()		111	Aliases:			
	0.030	0.000	java.util concurrent atomic Atom	iclong compareAndSet(long long)	-				
Disassembly	0.020	0.050	java.util.Bandom.next(int)	recongreenpareshaber(cong, cong)			\$1 Exc	lusive	
Callero Calle	0.010	0.010	java.net.URLClassLoader.findReso	ources(java.lang.String)		Total Thread Time:	546.913	(100.00	0%)
Callers-Calle	0.010	0.010	java.util.Formatter.access\$000(j	java.util.Formatter)		Total CPU Time:	78.515	(100.00	0%)
Experiments	0.010	0.010	sun.misc.URLClassPath\$JarLoader\$	1.run()		User CPU Time:	78.365	(100.00	0%)
	Θ.	Θ.	Interpreter			System CPU Time	0.140	(100.00	(0%)
Threads	0.	0.	InterpreterRuntime::_new(JavaThr	read*,constantPoolOopDesc*,int)		Trap CDU Time	0.010	(100.0)	00.1
Brococcoc	0.	O.	InterpreterRuntime::resolve_invo	oke(JavaThread*,Bytecodes::Code)		trap CFO fille:	0.010	(100.00	0-87
FIUCESSES	U.	U.	JVM_Monitorwait			Data Page Fault Time:	0.	(0.	%)
More	0.	0.	lavaCallecall victual (lavaValue	a*,methodHandte*,Javatattargument a* Handla KlaceHandla Symbol* Sy	2	Text Page Fault Time:	Θ.	(0.	%)
	0.	0.	lavaThreadrun()	ie*, Hallute, Ktasshallute, Sylibot*, Sy		Kernel Page Fault Time:	0.	(0.	%)
	0.	0.	JavaThread::thread main inner()			Stopped Time:	Ο.	(0.	%)
	0.	0.	Java java io UnixFileSystem cano	onicalize0		Wait CPU Time:	O .	(0.	%)
	0.	0.	Java_java_lang_ClassLoader_defin	neClass1		Sleep Time:	70.255	(100.00	08-1
	0.	Ο.	Java_java_security_AccessControl	ller_doPrivilegedLjava_security		Steep Time.	70.000	(100.00	0.0)
	0.	0.020	Java_java_security_AccessControl	ller_doPrivilegedLjava_security	4 11	User Lock Time:	390.043	(100.00	098)
	0.	0.010	Java_java_security_AccessControl	ller_doPrivilegedLjava_security					
	0.	0.	LinkHesolver::resolve_invoke(Cal	llinto&, Handle, constantPoolHandle					
	0.	0.	LinkHesolver::resolve_invokestat	llc(CallInto&, constantPoolHandle,	÷				
	4	0.	TETIKNESUTVEL. TESUTVE Static cat	ktricattiining, ktasshalluteg, sviibot.					
	Called by / C	alle							
Filters	called-by / C	ans							
bax =			<lotat></lotat>						
	I otal	< I otal>	I otal C	<lotal></lotal>					
To add a filter,	ATRIBUTED	s called by	ATTRIBUTED	calls					
view (such as	00C. V		79.175	ilowfruit main(iava lang String)	1				
		1							
Compare			0.010	sun.launcher.LauncherHelper.chec	k				
12.3 +/- 1.1X J1 =			0	lwn start		4			
			P 4			1			
Local Host:	Remote Host	t: Working	Directory:/jlowfruit Compare: or	ff Filters: off 🧘 Warning				1	1/103

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.

 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.

- 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.



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 - Oracl	e Developer Studio Perfo	rmance Analyzer	= = 🛛
<u>File</u> <u>Views</u> <u>M</u> etrics	<u>T</u> ools <u>H</u> elp				
🛃 🖾 🗯 🖏 🛛 🏹	😼 🙆 🛛 View	/ Mo <u>d</u> e: User		Find: Find text in view	🔽 💫 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total	jlowfruit.java			III 🙈 (
Welcome	CPU Time				
Overview	sec.	92.			
Functions		93. void	ad(int insert count)		
Timeline		95. {	ad(int insert_count)		
Call Tree		90. <functio< th=""><th>n: jlowfruit.insert_bad(in</th><th>t)></th><th></th></functio<>	n: jlowfruit.insert_bad(in	t)>	
Source >	0. 0.	97. 98.	count = 0; insert_table = insert_init	(insert_count);	
Disassembly	Θ.	99. 100.	<pre>for(i = 0: i < insert coun</pre>	- t; i ++) {	=
Callers-Calle	0.050	101.	<pre>/* get a new element * novP = rand();</pre>	/	
Experiments	0.	103.	done = 0;		
Threads	9.877	104.	if(newR < insert_t	unt;] ++) { :able[j]) {	_
Processes	15.291 0.	106. 107.	<u>insert_number(</u> done = 1;	newR,j);	
More	Ο.	108.	break; }		
	0	110.	}		
	0.	112.	insert_number(newR	R, count);	
		113. 114.	}		•
		115. 116.			=
	0	117. //	free(insert_table);		-
	0.	110. 7			
		120. vold 121. insert_(ood(int insert_count)		_
F <u>i</u> lters		122. { 123.	int i, x, done, newR,	left, right, curval;	
bcx ≡		124. <function< th=""><th>n: ilowfruit insert good(i</th><th>nt)></th><th></th></function<>	n: ilowfruit insert good(i	nt)>	
To add a filter, select a row from a	0.	125.	count = 0;	(incont count).	_
view (such as	0.	120.	insert_table = insert_init	(insert_count);	_
compare	0.	128.	<pre>tor(1 = 0; 1 < insert_coun</pre>	t; 1 ++) {	
12:3 +1-112 +1 =					
Local Host:	Remote Hos	t: Vorking E	irectory: /jlowfruit Compa	are: 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().

	🔁 tes	t.l.er - Orac	le Developer Studi	o Performance Analyzer	= = 2
<u>File Views</u> Metrics	<u>T</u> ools <u>H</u> elp				
🗷 🕼 🗯 🖏 🛛	🗟 🙆 Viev	w Mo <u>d</u> e: User	_ < >	Find: Find text in view	🔽 🗟 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total	jlowfruit.java			III 🙈
Welcome	CPU Time				×
Overview	sec.	121 insert	good(int insert cou	n+)	
Functions		122. {			
Timolino		123.	int 1, x, done	newR, left, right, curval;	
milenne		<functs< th=""><th>ion: jlowfruit.inser</th><th>t_good(int)></th><th></th></functs<>	ion: jlowfruit.inser	t_good(int)>	
Call Tree	0.	125.	count = 0;		
Source	Ο.	126.	insert_table = ins	ert_init(insert_count);	
Disassembly	Ο.	128.	for(i = 0; i < ins)	ert_count; i ++) {	
	0.030	129.	/* get a new e newB = rand():	lement */	
Callers-Calle	0.000	131.	newry - rund (77		
Experiments		132.	/* do a binary	search to find its position */	
-	0.	133.	left = 0;		
Threads	υ.	134.	right = count	L;	_
Processes		136.	/* figure out	/here it belongs */	
	0.010	137.	<pre>while (left <=</pre>	right) {	
More	0.010	138.	x = (left ·	right) / 2;	
	0.020	139.	curval = i	nsert_table[x];	
		140.			
	Θ.	141.	if (curval	> newR) {	
	0.030	142.	ri	ht = x - 1;	
	0.	143.	} else 1†	(curval < newR) {	
	Θ.	144.	le	tt = x + 1;	
		145.	} else {	1.11.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	
		146.	/*	a duplicate value */	
	0.	147.	Le	rt = x;	
	Ο.	148.	r1	jnτ = x -1;	
		149.	3		
	14,000	150.	incont number/	novP loft).	
F <u>i</u> lters	14.890	152	1 Insert_number(newk, tert);	
bdx =	Θ.	153. }	1		
To add a filter.		154.			
select a row from a		155. int []	// int *		
view (such as		156. insert	_init(int maxcount)		
Compare		157. {	/* popot o popdom -	umber cood */	
compare		100.	ion, iloufruit incor	tipit(int)>	T
12.3 ±/- 1.1X ↓↑	1				E E
Local Host:	Remote Hos	st: Working	Directory:/jlowfruit	Compare: off 🛛 Filters: off 🛛 🧘	Warning 130/229

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.

13. Click on the Timeline view or choose Views \rightarrow Timeline from the menu bar.

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 \bigcirc for coloring functions or choose Tools \rightarrow Function Colors from the menu bar and see the dialog box as shown below.

	<u></u>	🔄 test. 1. er - Or	acle Developer St	tudio Perfo	rmance Ana	alyzer	= = 🛛		
Eile Views Metrics	s <u>T</u> ools <u>H</u> elp								
🖬 🖾 🛱 🖏 🛛 T	2 😼 🙆 Vie	ew Mo <u>d</u> e: User	-		Fi <u>n</u> d: F	ind text in view	🔽 🔍 🔍 🗌 Mat <u>c</u> h Case		
Vie <u>w</u> s 🔶	t men-				roup Doto b	• Throad	Selection Details		
Welcome				40 50	en en	70	Process: ls/		
Overview	l nine(sec)	I		40 30			Event Type: Clo		
Eunctions							Timestamp (sec.): 39.		
Timeline	1 T:2						LWP: 2		
Call Trees	Ŭ			ļ.			Thread: 2		
Call free	1 T:15						CPU: 228 Duration (msec.): 10		
Source		i					Call Stack Timeling		
Disassembly							ilowfruit.insert number		
Callers-Calle							jlowfruit.insert_good(ir		
Experiments							jlowfruit.mainEntrance(j ilowfruit.run(java lang		
Threads							jlowfruit.main(java.lan(
Proce			🖾 Euno	ction Colors			X		
More.									
Swatches	s <u>[H</u> SV HS <u>L</u>	<u> [RG</u> B [C <u>M</u> YK				Legend			
						java.i	o.PrintStream.printf(java ang.ClassLoader.getResc		
						java.l	ang.ClassLoader.loadClas		
				Recer	nt:	📃 java.l	ang.ClassLoader.loadClas		
						java.r	net.URLClassLoader\$1.rur		
						java.r	net.URLClassLoader.findR		
						java.s	security.AccessController.		
Filter						java.s	security.AccessController.		
1 In Col						Java.t	ext.DecimaiFormatSymbo		
To add	To add Set Selected Functions Set All Functions Reset Default Colors								
select									
Comp	unctions: St								
Set CPU Idl	e Events Color:	● <u>N</u> ormal ○ <u>I</u> nvis	ible 🔾 Selected C <u>o</u> l	or 📃					
							Close		
Local									

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 CPU 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 CPU 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 two bars of data for events occurring during program execution. The upper bar shows color-coded representations of the callstack and the lower bar shows the state of the thread at each event. The state in this example was always User CPU Time so it appears to be 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.

 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 the slider at the top of the Timeline, or using the + key, or by double-clicking with the mouse.

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.

			Tel P	പെ		≥ ₩	<i>«</i> П П	N	A 6	8	Gro	ם מוווס)ata	hv: T	hrear	1 🔻	Selection Details
									<u> </u>	~	0.0	Jub r	and	- 1. L	in out		Process: st.l.er [java, PID 4075]
lime(se	(C)	36.95		y . `	37.00					37.0	. 21			Event Type: Clock Profiling			
1																	Leaf Function: jlowfruit.init_bad(int)
1 7.2																	Timestamp (sec.): 36.985890
11:2	\odot																LWP: 2
		÷ .	÷.	÷.	÷.	1		i.	i.	÷	i.	÷.	÷.	÷.	÷.	÷.,	Thread: 2
1 T:15																	CPU: 239
	(>)																Duration (msec.): 10.007
																	Thread State: 🗾 User CPU
																	Call Stack - Timeline jlowfruit.init_bad(int) + 0x00000027, line jlowfruit.mainEntrance(java.lang.String[]) jlowfruit.run(java.lang.String[]) + 0x0000 jlowfruit.main(java.lang.String[]) + 0x000

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 \rightarrow 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.

2	🖾 test. 1. er 🕒 Oracle Developer Studio Performance Analyzer		📃 🖃 🔀
<u>F</u> ile ⊻iews <u>M</u> etrics	Tools Help		
2 🖻 🛱 🖏 🖣	😼 🔞 View Mo <u>d</u> e: User 👻 Fi <u>n</u> d: Find te	∉ in view 🔹	🕶 👧 😡 🗆 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Call Tree: FUNCTIONS. Complete view. Threshold: 1% Sort by: metric. Metric: Attributed Total CPU Tim + 78.515 (100%) <total></total>	Selection Det	tails
Welcome	78.175 (100%) jlowfruit.main(java.lang.String[])	Name:	line 23 in "jlowfru
Overview	78.175 (100%) jlowfruit.run(java.lang.String[])	PC Address:	401:0x0000043F
Functions	32.173 (41%) ilowfruit.init bad(int)	Size:	0
Functions	26.388 (34%) jlowfruit.insert_bad(int)	Source File:	jlowfruit.java
Timeline	14.990 (19%) jlowfruit.insert_good(int)	Object File:	:est.l.er/archives/j
	 4.603 (6%) [IOWTRUIT.INIT_good(INT) 0.020 (0%) java jo PrintStream printf(java lang String java lang Object[]) 	Load Object:	jlowfruit.class (fo
	- 0.330 (0%)	Mangled Name:	jlowfruit.mainEntra
Source	 0.010 (0%) sun launcher.LauncherHelper.checkAndLoadMain(boolean, int, java.lang.String) 	Aliases:	
Disassembly	⊶ 0. (0%) _lwp_start		St Exclu
Callers-Calle		Total Threa	ad Time: 0. (
Experiments		Total CP	CUTime O (*
experiments		Call Stack - T	imeline
Threads		jlowfruit.i	nit bad(int) + 0x0000
Processes		jlowfruit.m	ainEntrance(java.lang
Mara		jlowfruit.r	un(java.lang.String[]
More		jlowfruit.m	ain(java.lang.String[

48 Oracle Developer Studio 12.5: Performance Analyzer Tutorials • June 2016

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 j synprog" on page 51
- "Examining the jsynprog Data" on page 52
- "Examining Mixed Java and C++ Code" on page 55
- "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.

j synprog 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
- synprog.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
- synprog.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 OracleDeveloperStudio12.5-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.ler.

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.l.er.

% analyzer test.1.er

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

	🔤 test.1.er - Or	acle Developer Studi	o Performance A	nalyzer	= = 2							
<u>File Views Metrics</u>	[ools <u>H</u> elp											
🗟 🖾 😂 🛛 🏹 🤇	🗟 🥘 View Mo <u>d</u> e: User	-	Fi <u>n</u> d:	Find text in view	🔽 🔍 🔍 🗌 Mat <u>c</u> h Case							
Vie <u>w</u> s 📀	Experiment(s)											
Welcome	b test 1 er											
Overview >	r collion											
Functions	Metrics											
Timeline	Select the metrics to display	in the data views, then	click a data view in	the navigation pane	l on the left.							
Call Tree	Available Metrics			₩ Hot <u>R</u> eset	Cle <u>a</u> r All							
Source	Experiment Duration: 81.163	3 Seconds										
Disassembly	Java Garbage Collection Dur	ation: 0.127 Seconds										
Callers-Calle		1621.194 Seconds										
Experiments	P− Total Thread Time	🔳		100%								
Threads	👇 🧧 Total CPU Time			5% 🗰 🖌								
Processes	- User CPU Ti	ne 📕		5%								
More	Trap CPU Tir	me		0%								
-internet -	📕 Data Page Fau	t Time		0%								
	– 📕 Text Page Fault	:Time		0%								
	– 📕 Kernel Page Fa	ult Time 🛛 📘		0%								
	Stopped Time			0%								
	Sleen Time	· · · · · · · · · · · [_		5%								
	User Lock Time			90%								
Filters												
bGX =	Metrics Preview											
To add a filter,	Total CPU Time	Name										
select a row from a	ST EXCLUSIVE ST INCLUSIVE											
new (Such as	81,977 81,977	<total></total>										
Compare												
12.3 +/- 1.1X ↓↑ =												
ocal Host:	Remote Host: Working [)irectory:/jsynprog	Compare: off Fil	ters: off 🥂 Warni	ng							

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. Select the check boxes for the Sleep Time and User Lock Time metrics to add them to the data views.

— 📕 Sleep Time 🌐	5%	V
User Lock Time 🛛	90%	\mathbf{v}

Metrics Previe	ew	\square	Metrics Preview	v changed
Total CI	PU Time	Sleep T	OJCI LOCICIII	
ST EXCLUSIVE	CLUSIVE 11	SI EXCLUSIVE	ST EXCLUSIVE	
sec.	sec.	sec.	sec.	
81.977	81.977	81.137	1 458.060	<total></total>

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:

		🔁 test.	1.er - Oracle D	eveloper Studio Performance Ana	yzer	
Eile ⊻iews Metrics	<u>T</u> ools <u>H</u> elp					
🛃 📾 😂 🛛 🏹	🔞 🙆 View	Mo <u>d</u> e: User	-		Find: Find text in view	💌 💫 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total	Sleep	User Lock	Name	Selection Details	
Welcome	CPUTIme	TIME	lime		Index Object: Bresses 1	Thread 2 Thread 2 1
Televine	VALUES	VALUES	VALUES		index object: Process 1	Thread 2, Stillead S
Overview	01 077	01 107	1 459 060	<total></total>		2 Exclusive
	90.057	01.137	0.160	Process 1 Thread 2 IThread 3 'mai	Total Thread Time	81,127 (5,00%)
Functions	0.100	0.010	80.987	Process 1 Thread 9	Tatal CDU Time	00.057 (00.750)
T	0.080	0.	81,007	Process 1, Thread 3	Total CPU Time:	80.957 (98.75%
rimeline	0.080	0	81 007	Process 1 Thread 6	User CPU Time:	80.887 (98.94%
Call Tree	0.080	0.	81.007	Process 1. Thread 8	System CPU Time:	0.070 (33.33%
cummoo	0.080	0.	81.007	Process 1, Thread 10	Trap CPU Time:	0. (0.%)
Source	0.080	0.	81.007	Process 1, Thread 11	Data Page Fault Time	0 (0.8)
	0.080	0.	80.967	Process 1, Thread 13	The second secon	0. (0. 0
Disassembly	0.070	0.	81.017	Process 1, Thread 4	Text Page Fault Time:	U. (U. %
Callero Calle	0.070	0.	81.017	Process 1, Thread 5	Kernel Page Fault Time:	0. (0.%
callers-calle	0.070	0.	81.017	Process 1, Thread 7	Stopped Time:	0. (0.%
Experiments	0.070	Ο.	81.007	Process 1, Thread 12	Wait CPU Time	0 (0 %
experimence	0.050	0.	80.937	Process 1, Thread 17	Class Time	0.010 (0.010
Threads 💦 📎	0.040	0.	80.907	Process 1, Thread 18	Sleep Time:	0.010 (0.01%
	0.030	81.127	Ο.	Process 1, Thread 1	User Lock Time:	0.160 (0.01%)
Processes	0.030	Ο.	80.947	Process 1, Thread 20		
	0.010	Ο.	81.037	Process 1, Thread 15, JThread 1 '',		
More	Ο.	Ο.	81.047	Process 1, Thread 14, JThread 0 '',		
	Ο.	Ο.	80.997	Process 1, Thread 16, JThread 2 'Si	(
	Θ.	Ο.	80.987	Process 1, Thread 19		

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.

5. Go back to the Overview and deselect Sleep Time and User Lock Time.

6. 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 topmost function jsynprog.main(). That routine is the initial routine that the JVM calls to start execution.

			🔁 test. 1. er 🕒	Oracle Developer Studio Performanc	e Ana	alyzer		= =
File Views Metrics	Tools <u>H</u> elp							
V 🖄 🛱 🖏 🖓	🗟 🙆 🛛 Viev	w Mo <u>d</u> e: User	▼ <>			Fi <u>n</u> d: F	ind text in view	💌 👧 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total C	PU Time	Name		Ш	Selection Detail	s	
Welcome	Sec.	Sec.				Name: isynor	on main(iava.lang.	String[])
	81,977	81,977	<total></total>			PC Addroses 401.0v	000000002	Set anget?
Overview	2.262	65.376	isynprog.main(j	ava.lang.String[])		Cias: 410	00000002	
Eunctions	0.	15.421	<truncated-stack< th=""><th>6</th><th></th><th>Size: 410</th><th></th><th></th></truncated-stack<>	6		Size: 410		
runctions y	15.421	15.421	Routine.recurse	deep(int, int, int)		Source File: jsynpr	og.java	
Timeline	15.351	15.351	Routine.bounce(:	int, int, int)		Object File: nd as	test.l.er/archives	/jsynprog.class_dbrs1000
	0.	15.351	Routine.bounce_	b(int, int, int)		.oad Object: jsynpr	og.class (found as	test.l.er/archives/jsyn
Call Tree	15.351	15.351	Routine.recurse	(int, int, int)		idled Name: isynor	ng main	
Fourse	0.010	5.294	jsynprog.JavaJa	vaC(int, int)		Alianae	ogradan	
Source	0.	5.294]synprog.jni_Ja	vaJavaC(int, int)		Allases:		
Disassembly	5.284	D. 284	Java_Jsynprog_J	avajavac	1.000		St Exclusive	11 Inclusive
() ()	4.0/3	4.0/3	Pouting, system	i.arraycopy(java.tang.object, int, java.	Lang	Total Thread Time	2,262 (0.)	(4%) 65,526 (4,04%)
Callers-Calle		4.015	[Tourine.vector]	59(111)		Total CPU Time	2 262 (2	76%) 65 376 (79 75%)
Exporimonte	1.			-		Lloor CPU Time	2.202 (2.	77%) 65.376 (70.04%)
experiments	Called-by / C	Calls				Oser CFU Time	2.202 (2.	03.330 (79.946)
Threads			jsynprog.main(ja	ava.lang.String[])		System CPU Time	: 0. (0.	%) 0.020 (9.52%)
-	Total C	jsynprog.m	Total C	jsynprog.main(java.lang.String[])		Trap CPU Time	: 0. (0.	%) 0. (0. %)
Processes	ATTRIBUTED	is called by	ATTRIBUTED	calls		ita Page Fault Time	: 0. (0.	%) O. (O. %)
More	sec. 🔻		sec. 🔻	•		ext Page Fault Time	. 0. (0.	%) 0. (0. %)
	65.376	<total></total>	15.351	Routine.bounce(int, int, int)	-	hel Page Fault Time	. 0. (0.	S) 0, (0, S)
			15.351	Routine.recurse(int, int, int)		Stopped Time	0 (0	(0)
			5.294	jsynprog.jni_JavaJavaC(int, int)	- 1	Weit CDU Time	0. (0.	a) 0 (0 a)
			4.013	Houtine.vector_op(int)		wait CPU Time		6) 0. (0. 6)
			3.703	Routine sys on(int)	- 1	Sleep Time	: 0. (0.	 6. (0.%)
			3.012	Routine add double(int)		User Lock Time	: 0. (0.	%) 0.150 (0.01%)
Filters			3.012	isynprog. JavaCJava(int)	- 11			
ritters			3.002	Sub Routine.add int(int)				
bex ≡			2.992	Routine.add int(int)				
To add a filter			2.992	Routine.has_inner_class(int)				
select a row from a			0.791	Routine.array_op(int)	- 11			
view (such as			0.440	Routine.memalloc(int, int)				
-			0.040	jsynprog.printValue(java.lang.String, b	bool			
Compare			0.010	Launcher.main(java.lang.String[])				
12.3 +/- 1.1X J1 =			0.010	java.lang.ClassLoader.loadClass(java.la	ang. 👻			
		•			P			
Local Host:	Remote Hos	st: Working	Directory:/jsynp	rog Compare: off Filters: off 🛕 War	rning			2/145

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.

			r						
	.est.1.er - Oracle Deve	eloper Studio Per	formance	Analyzer					
Elle Views Metrics Loois Help									
🐱 🖾 🖽 💙 🛛 🏹 😼 🔯 🛛 View Mo <u>d</u> e: Us	ser 🔻			Find: Find t	ext in viev	N I	- K K	X 🗆 M	lat <u>c</u> h Case
Views Call Tree: FUNCTIONS. Co	omplete view. Threshold: 1	% Sort by: metric.	Metric: At	Selection Det	ails				
Welcome 65.3	376 (80%) jsynprog.main	(java.lang.String[])		Name:	jsynprog	. JavaCJav	a(int)		
Overview - 15.351 (1	.9%) Routine.bounce(int,	int, int)		PC Address:	401:0x000	000000			
• 15.351 (1 • 5.294 (6%)	.9%) Routine.recurse(int, isymprog.ini lavalavaC(int)	int, int) . int)		Size:	42949672	95			
Punctions •−■ 5.294 (6	i%) jsynprog.JavaJavaC(int	int)		Source File:	jsynprog	. java			
Timeline 5.284	(6%) Java_jsynprog_JavaJ	avaC		Object File:	synprog.	class (fo	ound as	test	.l.er/arch
Call Tree 3.763 (5%)	isynprog.lavaCC(int)			Load Object:	jsynprog	.class (1	ound a	s test	.l.er/arc
9 ■ 3.763 (5	%) Java_jsynprog_JavaCC			Mangled Name:	jsynprog	. JavacJav	a		
Source 3.763	(5%) cfunc(int) Boutine sys on(int)			Allases:					
Disassembly 3.012 (4%)	Routine.add_double(int)					\$\$ E>	clusive		11 Inc
Callers-Calle 9 3.012 (4%)	jsynprog.JavaCJava(int)	-		Total Threa	d Time:	0.	(0	. %)	3.012
Fxperiments	(4%) INIEnv ::CallStaticIn	a tMethod(iclass*, ir	nethodID*	Total CP	U Time:	0.	(0	. ~6) e.)	3.012
Laperinients	12 (4%) jsynprog.javafur	nc(int)		System CP	U Time	0.	(0	. ~>/ %)	0
Threads • 3.002 (4%)	Sub_Routine.add_int(int) Boutine.add_int(int)			Trap CP	U Time:	0.	(0	. %)	0.
Processes - 2.992 (4%)	Routine.has_inner_class(i	nt)		Data Page Fau	lt Time:	Ο.	(0	. %)	0.
More	Routine.array_op(int)			Text Page Fau	lt Time:	0.	(0	. %)	0.
 ► 0.040 (1%) ► 0.040 (0%) 	isvnprog.printValue(iava.la	ng.String. boolean)		Kernel Page Fau	lt Time:	0.	(0	. %)	0.
► 0.010 (0%)	Launcher.main(java.lang.St	tring[])		Stoppe	d Time:	Θ.	(0	. %)	Θ.
► 0.010 (0%)	java.lang.ClassLoader.load	Class(java.lang.Stri	ng)	Wait CP	U Time:	0.	(0	. %)	0.
► 0. (0%) ja) <truncated-stack></truncated-stack>			Slee	p Time:	0.	(0	. %)	0.
_ 1.131 (1%) <j< td=""><td>/VM-System></td><td>1 1 A 10 10 40 40 40 40 40 40 40 40 40 40 40 40 40</td><td></td><td>User Loo</td><td>k Time:</td><td>0.</td><td>(0</td><td>. %)</td><td>0.</td></j<>	/VM-System>	1 1 A 10 10 40 40 40 40 40 40 40 40 40 40 40 40 40		User Loo	k Time:	0.	(0	. %)	0.
- 0.030 (0%) sur - 0.010 (0%) <n< td=""><td>n.iauncher.LauncherHeiper. 10 Java callstack recorded></td><td>спескапdLoadMain</td><td>(boolean, lí</td><td></td><td></td><td></td><td></td><td></td><td></td></n<>	n.iauncher.LauncherHeiper. 10 Java callstack recorded>	спескапdLoadMain	(boolean, lí						
Filters ~ 0.010 (0%) _lw	/p_start								
I I I I I I I I I I I I I I I I I I I	lang.ref.Finalizer\$FinalizerTh	nread.run()							
To add a filter.	lang.rei.keierenceşkeierei	icenarialer.run()							
select a row from a									
C									
compare									
[12.3] +/- 11X ↓1			Þ	•					•
Local Host: Remote Host: Worl	king Directory:/jsynprog	Compare: off	Filters: off	Warning					

- 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.

 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.

	🔁 t	est.1.er - O	racle Developer Studio PerformanceAnalyzer 📒 📒	
Eile Views Metrics	<u>T</u> ools <u>H</u> elp			
🗟 🖾 😂 🛛 🏹	😼 🙆 View	Mo <u>d</u> e: Expert	Find: Find text in view 🔽 😡 🔍 Match C	Case
Vie <u>w</u> s 📀	Total CI	PU Time	Name	Ш
Welcome	sec.	sec. 👻		
Overview	81.977 2.262	81.977 65.376	<total> jsynprog.main(java.lang.String[])</total>	
Functions 💦 🕥	0. 15.421	15.421 15.421	<truncated-stack> Routine.recursedeep(int, int, int)</truncated-stack>	
Timeline	15.351	15.351	Routine.bounce(int, int, int) Routine bounce b(int, int, int)	
Call Tree	15.351	15.351	Routine.recurse(int, int, int)	
Source	0.	5.294	jsynprog.jni_JavaJavaC(int, int)	
Disassembly	5.284 4.673	4.673	Java_jsynprog_JavaJavac java.lang.System.arraycopy(java.lang.Object, int, java.lang.Object, int, int)	
Callers-Calle	0.	4.013 3.923	Routine.vector_op(int) Routine.vrem_first(java.util.Vector)	
Experiments	0.020	3.923 3.763	java.util.Vector.remove(int) Java jsynprog JavaCC	
Threads	3.763	3.763	cfunc(int)	
Processes	3.032	3.042	Routine.sys_op(int)	
More	0.	3.012	JNIENv_::CallStaticIntMethod(_]class*,_]methodiD*,) Java_jsynprog_JavaCJava	
	3.012	3.012 3.012	Routine.add_double(int) jsynprog.JavaCJava(int)	
	3.012	3.012	jsynprog.javafunc(int)	
	0.150	3.002	Sub_Routine.add_int(int) PoutinetllToner_buildlist(int)	
	2.992	2,992	Routine add int(int)	
	0.	2,992	Boutine.has inner class(int)	
	2.852	2.852	Sub Routine.addcall(int)	
	0.	1.121	_lwp_start	
	0.	0.991	java_start	
Filters	0.010	0.791	Routine.array_op(int)	
6 a x =	0.	0.781	GCTaskThread::run()	
	0.180	0.610	StealTask::do_it(GCTaskManager*,unsigned)	
lo add a filter,	0.440	0.440	Routine.memalloc(int, int)	
view (such as	0.270	0.420	<pre></pre>	
view (oden do	0.	0.130	JAVANALII INT Create JavaVM	
Compare	0.	0.120	Threads::create.vm(lavaVMInitArgs*.bool*)	-
12.3 +/- 1.1X ↓↑	1	51110		
Local Host:	Remote Hos	t: Working I	Directory:/jsynprog Compare: off Filters: off 🛕 Warning 35/3	44

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.

Image: Image	Image: Wight of the sec. O. 0. 0. 2.992 2.992 2.992 2.992	Mode: Machin PU Time \$ INCLUSIVE sec. 3.012 2.992	e ▼ Find: Find text in view ▼ Q Q □ Name Java_j§ynprog_JavaCJava ini_CallStaticIntMethody	Mat <u>c</u> h Case
Yie <u>ws</u> ↔ Velcome Overview unctions > imeline	Total CF SECLUSIVE SEC. 0. 0. 2.992 2.992 2.992	PU Time INCLUSIVE sec. ▼ 3.012 3.012 2.992	Name Java_jsynprog_JavaCJava	
Velcome Overview unctions	\$\$ EXCLUSIVE sec. 0. 2.992 2.992	INCLUSIVE sec. ▼ 3.012 3.012 2.992	Java_j§ynprog_JavaCJava	
Velcome Overview unctions	sec. 0. 0. 2.992 2.992	sec. ▼ 3.012 3.012 2.992	Java_jsynprog_JavaCJava	*
overview unctions	0. 0. 2.992 2.992	3.012 3.012 2.992	Java_jsynprog_JavaCJava ini CallStaticIntMethodV	-
unctions >	0. 2.992 2.992	3.012 2.992	ini CallStaticIntMethodV	
unctions	2.992 2.992	2,992		
imeline	2.992		Routine.add_int(int)	
imeline all Tree	0.000	2.992	Sub_Routine.add_int(int)	
all Tree	2.992	2.992	jsynprog.javafunc(int)	
all Tree	2.982	2.982	Routine\$1JInner.buildlist(int)	
an nee	2.982	2.982	Routine.add_double(int)	
	0.620	2.942	os::javaTimeMillis()	
ource	0.090	2.872	Routine.sys_op(int)	
leaseambly	2.322	2.322	gettimeofday%sun4v-hrt	
isassembly	0.	2.242	Routine.array_op(int)	
allers-Calle	2.242	2.242	arrayof oop disjoint arraycopy	
anoro canom	0.	0.991	java start	
xperiments	0.	0.781	GCTaskThread::run()	
	0.180	0.610	StealTask::do it(GCTaskManager*,unsigned)	
hreads	0.	0.450	InterpreterRuntime::anewarray(JavaThread*,constantPoolOopDesc*,int,int)
	0.010	0.450	instanceKlass::allocate objArray(int,int,Thread*)	
rocesses	0.440	0.440	zero aligned words	
	0.270	0.420	ParallelTaskTerminator::offer termination(TerminatorTerminator*)	
tore	0.	0.120	JNI CreateJavaVM	
	0.010	0.120	Routine.sys op(int)	
	0.	0.120	Threads::create vm(JavaVMInitArgs*,bool*)	
	0.100	0.100	<static>@0xabae0c (<libjvm.so>)</libjvm.so></static>	
	0.	0.100	JavaThread::run()	
	Ο.	0.100	JavaThread::thread main inner()	
	0.	0.100	StealRegionCompactionTask::do it(GCTaskManager*.unsigned)	
	0.	0.090	C2Compiler::compile method(ciEnv*.ciMethod*.int)	
	0.	0.090	CompileBroker::compiler thread loop()	
	0.	0.090	CompileBroker::invoke compiler on method(CompileTask*)	
	0.	0.090	Monitor::IWait(Thread*.long long)	
ilters	0.	0.090	Monitor::wait(bool.long.bool)	
	0.	0.080	Compile::Compile(ciEnv*,C2Compiler*.ciMethod*.int.bool.bool)	
	0.080	0.080	PSPromotionManager::copy to survivor space <false>(oopDesc*)</false>	
o add a filter,	0.020	0.080	PSPromotionManager::drain stacks depth(bool)	
elect a row from a	0.	0.080	VMThread::loop()	
iew (such as	0.	0.080	VMThread::run()	
	0.080	0.080	lwn cond wait	
ompare	0.010	0.070	SystemDictionary::resolve.or.null(Symbol*.Handle.Handle Thread*)	
2.2 4/ 118 14 =	4	01070	ay a compare carrier y i i roba cva_or_ina ce (ay mode y numa ce) numa ce, nin edu / /	

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.

	🖸 🖸 🖸 🖸 🖸 🖸 🖸 🖸 🖸 🖸							
Eile Views Metrics	<u>T</u> ools <u>H</u> elp							
🛃 🖾 🛱 🗳 🛛 🏹	🖲 🙆 View	/ Mo <u>d</u> e: Machin	ie 💌 <	Fi	i <mark>nd:</mark> Find text in v	iew	- 🔍 😡	🔲 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total CI	PU Time	Name	Ш	Selection Deta	ils		
Welcome	sec.	sec. V			Name: I	Routine.	add double	e(int)
Overview	0.	3.012	Java_jsynprog_JavaCJava	-	PC Address:	0:0xF784	F9A0	
Eunctions	2.992	2,992	Routine.add int(int)		Size:	384		
- unceronis	2.992	2.992	Sub_Routine.add_int(int)		Source File: I	Routine.	java DTLED METL	10000
Timeline	2.992	2,992]synprog.javafunc(int) Routine\$lJInner.buildlist(int)		Load Object:	JAVA_COM	PILED_METH	1005
Call Tree	2.982	2.982	Routine.add double(int)		Mangled Name:	Routine.	add double	
Source	0.620	2.942	os::javaTimeMillis() Routine.svs op(int)		Aliases:		-	
Disassembly	2.322	2.322	gettimeofday%sun4v-hrt				SI Exc	lusive
Callers Calle	0.	2.242	Routine.array_op(int)		Total Thread	d Time:	2.982	(0.18%)
callers-calle	0.	0.991	java_start		Total CPU	J Time:	2.982	(3.64%)
Experiments	0.180	0.781	GCTaskInread::run() StealTask::do it(GCTaskManager*	6	User CPU	J Time:	2.982	(3.65%)
Threads	0.	0.450	InterpreterRuntime::anewarray(J	la	System CPU Trap CPU	J Time:	0.	(0, %)
Processes	0.010	0.450	instanceKlass::allocate_objArra	9)	Data Page Faul	t Time:	0.	(0, %)
More	0.270	0.420	ParallelTaskTerminator::offer_t	Le 🛛	Text Page Faul	t Time:	0.	(0.%)
	0.010	0.120	JNL_CreateJavaVM Routine.svs_op(int)		Kernel Page Faul	lt Time:	0.	(0.%)
	0.	0.120	Threads::create_vm(JavaVMInitAr	·	Stopped	d Time:	0.	(0. %)
	0.100	0.100	<pre><static>@0xabae0c (<libjvm.so>) lavaThread::run()</libjvm.so></static></pre>	1	Wait CBI	I Timo:	0	
	0.	0.100	JavaThread::thread_main_inner()		Call Stack - Tin	neline		
	0.	0.100	StealRegionCompactionTask::do_i C2Compiler::compile_method(ciEn		Routine.recu	rsedeep	int, int,	int) + 0x000
	0.	0.090	CompileBroker::compiler_thread_		Routine.recu	rsedeep	int, int,	(1nt) + 0x000 (nt) + 0x000
	0.	0.090	CompileBroker::invoke_compiler_	5	Routine.recu	rsedeep	int, int,	int) + 0x000
Filters	0.	0.090	Monitor::wait(bool,long,bool)		Routine.recu	rsedeep	(int, int,	int) + 0x000
b@x ≡	0.	0.080	Compile::Compile(ciEnv*,C2Compi		Routine.recu	rsedeep	int, int,	int) + 0x000
To add a filter,	0.020	0.080	PSPromotionManager::drain_stack		Routine.recu Routine.recu	rsedeep	int, int,	(1nt) + 0x000 (nt) + 0x000
select a row from a view (such as	0.	0.080	VMThread::loop()		Routine.recu	rsedeep	int, int,	int) + 0x000
Compare	0.080	0.080	lwp_cond_wait		Routine.recu	rsedeep	int, int,	int) + 0x000
	0.010	0.070	SystemDictionary::resolve_or_nu	1 -	Routine recu	irsedeep	int int	int) + 0x000
			F		4		-	•
Local Host: fungible	Remote Hos	t: Working	Directory:/jsynprog Compare:	off	Filters: off	🔥 Warnir	ig	36/242

6. 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.

		🔁 test.1.er 🕒 Oracle Deve	loper Studio Performance A	Analyzer				×
<u>File</u> <u>Views</u> <u>M</u> etrics	<u>T</u> ools <u>H</u> elp							
🗟 🖾 🕮 🕺 🛛 🏹	😼 🙆 Vie	w Mo <u>d</u> e: Machine 💌		Fi <u>n</u> d: Fi	nd text in view	- 🔍 🔍	. 🗌 Ma	t <u>c</u> h Case
Vie <u>w</u> s 🔶	Total	Routine.java	I	🛛 🙈 🛛 Selec	tion Details			
Welcome	Time			▼	Name: Routine.a	add_double	(int)	+ 0x00
Overview	Sec.			PC	Address: 0:0xF784F	-9A0		
Functions		<pre>22. } while (jsynprog.Ti</pre>	imer() < tEnd);	A	Size: 4			
Timolino		23. return x; 24. }		0	biect File: JAVA COM	java PTLED METH	ODS	- 1
Timenne		25.		Loa	d Object: JAVA COMP	PILED METH	ODS	
Call Tree		 25. /* add double */ 27. public double add double ((int scale) {	Mangl	ed Name: Routine.a	add_double		
Source		28. double y = 0.0;			Aliases:			
Disassembly >		 Int kmax = 1*scale; double tEnd = isvnp 	rog.Timer() + isvnprog.testti			St Exc	lusive	
Callers-Calle		31. do { y = 0.0;		Т	otal Thread Time:	0.	(0.	%)
Experimente		32. for (int k=0; k <kma) 33. for (int j=0; j<)</kma) 	(; K++) { L0000; j++) {		Total CPU Time:	0.	(0.	%)
Experiments		<function: routine.add_doub<="" th=""><th>le(int)></th><th></th><th>System CPU Time:</th><th>0.</th><th>(0.</th><th>-8/ -8)</th></function:>	le(int)>		System CPU Time:	0.	(0.	-8/ -8)
Threads	0.	[<u>33</u>] 0: ta [33] 4: sethi	16 %hi(0xffffa000), %q3		Trap CPU Time:	0.	(0.	%)
Processes	0.	[<u>33]</u> 8: clr	[%sp + %g3]	Data	Page Fault Time:	0.	(0.	%)
More	0.	[<u>33</u>] C: save [33] 10: ld	%sp, -136, %sp [%i0 + 8], %f8	Text	Page Fault Time:	0.	(0.	%)
	0.	[<u>33]</u> 14: ld	[%i0 + 12], %f9	Kernel	Page Fault Time:	0.	(0.	%)
	0.	[<u>33</u>] 18: std	%f8, [%sp + 96]		Stopped Time:	0.	(0.	%)
	0.	[33] 20, 1d	[8i0 + 20], 8i0		Wait CPU Time:	Θ.	(0.	%) ₹
	0.	[33] 24: ld	[%i0 + 16], %l2					F .
	Ο.	[<u>33</u>] 28: ld	[%i0], %l0	Call 9	Stack - Timeline			
	Ο.	[<u>33</u>] 2c: std	%f8, [%sp + 104]	Rou	utine.recursedeep(int, int,	int) +	0x000 ^
	0.	[<u>33</u>] 30: ld	[%10 + 4], %13	Rou	utine.recursedeep(int, int,	int) +	0x000
	U.	[<u>33</u>] 34: mov	%10, %00 Ovfffffffffaa 4ad24 L (Upph)	Rou	utine.recursedeep(int, int,	int) +	0x00C
	0.	[<u>33</u>] 30; Call	% 17 % 17	Rou	utine.recursedeep(int, int,	int) +	0x00C
Filters	0.	[33] 40; mov	%17 %n2	Rou	utine.recursedeep(int, int,	int) +	0x000
	0.	[33] 44; rd	%asr5, %15	Rou	utine.recursedeep(int, int,	int) +	0x000
	0.	[<u>33</u>] 48: ldd	[%15 - 100], %f8	Rou	utine.recursedeep(int, int,	int) +	0x000
To add a filter,	0.	[<u>33</u>] 4c: sethi	%hi(0x2400), %ll	Rou	utine.recursedeep(int, int,	int) +	0x000
select a row from a	Ο.	[<u>33</u>] 50: inc	770, %ll	Rou	utine.recursedeep(int, int,	int) +	0x000
View (such as	0.	l <u>33</u>] 54: sethi	%h1(0x2400), %l4	Rou	utine.recursedeep(int, int,	int) +	0x000
Com <u>p</u> are	U.	[<u>33</u>] 58: 100	/85, %14	Rou	utine.recursedeep(int, int,	int) +	0x000
12.3 ±/- 1.1X ↓↑ =	4	(<u>22)</u> 30: Sethi	**************************************	Rou 4	itine_recursedeen(int int	int) +	• • • • • • • • • • • • • • • • • • •
Local Host:	Remote Ho	st: Working Directory:/jsynprog	Compare: off Filters: off	🔥 Warning	9			35/321

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 CPU 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.

Only the first thread, Thread 2 labeled as T:2 in the example, shows its microstate as User CPU. The other two threads spend all their time waiting for a User Lock, 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. Use the vertical zoom control at the top of the timeline to adjust the zoom so that you can see all the threads.

The vertical zoom control is outlined in red in the following screen shot. Click the minus button to reduce the height of the thread rows until you can see all twenty threads.

	🔤 test. 1. er 🕒 Oracle Developer S
<u>File Views Metrics</u>	s <u>T</u> ools <u>H</u> elp
🗟 🖾 😂 🛛 🕇	a 🚱 🙆 View Mo <u>d</u> e: Expert 🔽
Vie <u>w</u> s 🔶	Ħ====================================
Welcome	Lune(sec) 0 10 20 30 40 50
Overview	Adjust the Veritcal Zoom (Ctrl-Plus, Ctrl-Minus)
Functions	
Timeline >	
Coll Trees	1 T:1

4. 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.

5. Use the horizontal slider in the Timeline tool bar to zoom in close enough to see individual events in thread T:2.

You can also zoom by double-clicking or pressing the + key on your keyboard.

Each row of the timeline actually includes three data bars. The top bar is a representation of the callstack for that event. The middle bar shows black tick marks wherever events occur too closely together to show them all. In other words, when you see a tick mark, you know that there are multiple events in that space.

The lower bar is an indicator of the event state. For T:2 the lower bar is green, which indicates User CPU Time was being used. For threads 3 through 12 the lower bar is gray, which indicates User Lock Time.

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.

- 6. Zoom in to the Routine.memalloc region and filter to include only that region by doing the following:
 - Click on the T:2 bar close to the beginning of the Routine.memalloc region with the red function call on top.
 - Click and drag the mouse to close to the end of that region where the red at the top of the call stack ends.
 - 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.

After zooming you can see that there are some event states in threads 3-12 that are green to indicate User CPU time, and even a few that are red to indicate Wait CPU Time.

7. 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.

8. 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.

9. 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, then reset the horizontal zoom to the default by pressing 0 on your keyboard.

You can also click the |< button in front of the horizontal slider in the Timeline tool bar, or right-click in the Timeline and select Reset.

2. 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.

	Eunction Colors	×
Swatches HSV HSL RGB CMYK	Recent:	Legend Routine.bounce(int, int, int) Routine.has_inner_class(int) Routine.recurse(int, int, int) Routine.recurse(int, int, int) Routine.sys_op(int) Routine.vector op(int)
Set Selected Functions	Set <u>All</u> Functions	<u>R</u> eset Default Colors
Set <u>Functions</u> : Starts with	le ○ Selected C <u>o</u> lor <mark>□</mark>	
		Close

3. 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.

4. Go to the Threads view and select thread 2 and the threads in your experiment that show activity during the time period where thread 2 shows calls to Routine.* functions.

You might find it easier to first sort by name by clicking the Name column header. Then select the multiple threads by pressing Ctrl as you click the threads.

In this example, threads 2, 17, 18, 19 are selected.
Total CPU Time	Name	
VALUES		
sec.		
81.977	<total></total>	
0.030	Process 1, Thread 1	
80.957	Process 1, Thread 2, JThread 3 'main', Group 'main', Parent 'system'	
0.080	Process 1, Thread 3	
0.070	Process 1, Thread 4	
0.070	Process 1, Thread 5	
0.080	Process 1, Thread 6	
0.070	Process 1, Thread 7	
0.080	Process 1, Thread 8	
0.100	Process 1, Thread 9	
0.080	Process 1, Thread 10	
0.080	Process 1, Thread 11	
0.070	Process 1, Thread 12	
0.080	Process 1, Thread 13	
Θ.	Process 1, Thread 14, JThread 0 '', Group '', Parent ''	
0.010	Process 1, Thread 15, JThread 1 '', Group '', Parent ''	
Θ.	Process 1, Thread 16, JThread 2 'Signal Dispatcher', Group 'system', Parent ''	
0.050	Process 1, Thread 17	
0.040	Process 1, Thread 18	
Θ.	Process 1, Thread 19	
0.030	Process 1, Thread 20	

5. Click the filter button The toolbar and select Add Filter: Include only events with selected items.

This sets a filter to include only events on those threads. You could also right-click in the Threads view and select the filter.

- 6. Return to the Timeline View and reset the horizontal zoom to make the pattern easier to see.
- 7. 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.

76 Oracle Developer Studio 12.5: Performance Analyzer Tutorials • June 2016

Hardware Counter Profiling on a Multithreaded Program

This chapter covers the following topics.

- "About the Hardware Counter Profiling Tutorial" on page 77
- "Setting Up the mttest Sample Code" on page 78
- "Collecting Data From mttest for Hardware Counter Profiling Tutorial" on page 79
- "Examining the Hardware Counter Profiling Experiment for mttest" on page 80
- "Exploring Clock-Profiling Data" on page 82
- "Understanding Hardware Counter Instruction Profiling Metrics" on page 84
- "Understanding Hardware Counter CPU Cycles Profiling Metrics" on page 86
- "Understanding Cache Contention and Cache Profiling Metrics" on page 88
- "Detecting False Sharing" on page 92

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 OracleDeveloperStudio12.5-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.l.er by default and contains clock-profiling data and hardware counter profiling data for three counters: inst (instructions), cycles (cycles), and dcm (data-cache-misses).

If your system does not support a cycles counter or a dcm counter, the collect command will fail. In that case, edit the Makefile to move the # sign to the appropriate line to enable the HWC_OPT variable that specifies only those counters that are supported on your system. The experiment will not have the data from those counters that were omitted.

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 12.5: 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.

•	🔄 test.1.er - Oracle Developer Studio Performance Analyzer
<u>File Views M</u> etri	cs Iools Help
2 🖻 🛱 🖏	🝸 🚱 😳 Find text in view 🔽 🔍 🛛 Mat <u>c</u> h Cas
Vie <u>w</u> s (Experiment(s)
Welcome	h test l or
Overview	V test.i.ei
unctions	Metrics
imeline	Select the metrics to display in the data views, then click a data view in the navigation panel on the left.
all Tree	Available Metrics
ource	Experiment Duration: 95.193 Seconds
Disassembly	▼ Clock Profiling ▼ Total Thread Time: 337.946 Seconds
Callers-Calle	
Experiments	P Total CPU Time
hreads	
Processes	
Memory pag	Data Page Fault Time
More	- Text Page Fault Time
	Kernel Page Fault Time 0%
	- Stopped Time
	Wait CPU Time
	▼ Derived and Other Metrics
	Instructions Per Cycle: 0.201
	Cycles Per Instruction: 4.981
	✓ Hw Counter Profiling ✓ Memoryspace Hardware Counters
	L1 D-cache Misses: 1622907507
	✓ General Hardware Counters
	▼ CPU Cycles Time: 196.328 Seconds
	CPU Cycles: 706782153629
Filters	
bex =	Metrics Proview
To add a filter,	Total CPU Time Name
select a row from a	Ser Ser
view (such as	197.738 197.738 <total></total>
ompare	
12.3 +/- 1.1X ↓↑ =	
Local Host:	Remote Host: Working Directory:/mttest Compare: off Filters: off A Warning

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.

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 the data was collected on a machine with a *precise* dcm counter, L1 D-cache Misses is listed under General Hardware Counters. If the data voluce Memoryspace Hardware Counters. If the dcm counter was available but is not a precise counter, L1 D-cache Misses is listed under General Hardware Counters. A precise counter is one whose overflow interrupt is delivered at the execution of the instruction causing the overflow. Non-precise counters are delivered with a variable amount of "skid" past the instruction causing the overflow. Even if a non-precise counter is memory-related, it cannot be used for memoryspace profiling. For more information about memoryspace profiling, see "Dataspace Profiling and Memoryspace Profiling" in *Oracle Developer Studio 12.5: Performance Analyzer*.

If your system does not support dcm, and you edited the Makefile to remove the -h dcm, you will see the Instructions Executed and CPU Cycles counter. If you edited the Makefile to remove both the -h dcm and -h cycles, you will only see the Instructions Executed counter.

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.

		🔤 test.	1.er - Oracle De	eveloper Studio Per	for	mance Analyze	r			
<u>File Views</u> <u>M</u> etri	ics <u>T</u> ools <u>H</u> elp									
🛃 🕼 😂 😓 📗	ዮ 🔒 🥥 🐨					Fi <u>r</u>	nd: Fina	d text in view	- 🔍 🔍	Mat <u>c</u> h Case
Vie <u>w</u> s	+ Total C	PU Time	Name		Ш	Selection Det	ails			
Welcome	Sec.				Ľ	Name	do word	k		
	197.738	197.738	<total></total>			PC Address:	2.0x00	004850		
Overview	0.570	197.728	do work			ric Address.	2.000	004000		
Functions	> 0.	185.700	_lwp_start			Source File:	mttest	. c		
	70.040	70.940	cache_trash		10	Object Filer	mttoct	(found as tost	1 on/orchiv	Inc (mttoct a)
limeline	0.540	39 748	cache trash odd			object File.	millest	(found as test	.i.er/archi	/es/millesi_gv
Call Tree	0.	31,192	cache trash even		10	Load Object:	mttest	(Tound as test	.1.er/archi	/es/mttest_gv
cummee	0.510	30.021	trylock global			Mangled Name:				
Source	5.374	17.502	mutex trylock			Aliases:				
	Θ.	12.128	do exit critical							
Disassembly	12.128	12.128	take_deferred_di	rect				\$1 EXCI	usive	
Callers-Calle	0.	12.038	_start			Total Threa	ad Time:	0.57	0 (0.17%)	254.78
canors canon	12.038	12.038	computeA			Total CF	U Time:	0.57	0 (0.29%)	197.72
Experiments	12.038	12.038	computeE			User CF	U Time:	0.57	0 (0.29%)	197.498
	12.038	12.038	computeH			System CP	U Time:	0.	(0,%)	0.120
Threads	0.	12.038	cond_timeout_glo	bal		Trop CF	U Timo	0.	(0, %)	0.110
Drocossos	0.	12.038	lock_local			nap cr	u =	0.	(0. %)	0.110
110003303	0.	12.038	lock_none			Data Page Fau	lit Time:	0.	(0, %)	θ.
Memory pag	0.	12.030	LUCKIESI		Ψ.	Text Page Fau	ult Time:	0.	(0.%)	0.
	Called-by / C	alls				Kernel Page Fau	It Time:	0.	(0.%)	0.
More			do work			Stoppe	ed Time:	0.	(0.%)	Θ.
	Total CP	do work	Total C	do work		Wait CE	U Time:	0.	(0,%)	0.
	ATTRIBUTED	is called by	ATTRIBUTED	calls		Slor	n Timor	0	(0 %)	0
	sec. 💌		sec. 💌			Jiee	spinne.	0.	(0, %)	57.000
	185.700	_lwp_start	70.940	cache_trash		User Lo	ck time:	U.	(0. %)	57.000
	12.028	locktest	30.021	trylock_global		L1 D-cache	Misses:		0 (0. %)	1622907507
			12.038	<pre>cond_timeout_global</pre>		Instructions Ex	ecuted:		0 (0. %)	141900334433
			12.038	lock_local		CPU Cycle	es Time:	0.00	9 (0.00%)	196.328
			12.038	lock_none		CPU	Cycles:	3201004	6 (0.00%)	706782153629
Filters			12.028	lock global		Instructions Pe	or Cycle	0	(0%)	0.201
i incera			12.010	nothroads		Ouslas Destrat	in cycle.	0.	(0, 0)	4.001
pcx =			12.010	calladd		Cycles Per Inst	ruction:	0.	(0, %)	4.981
To add a filter.			12.008	sema global						
select a row from a	a									
view (such as										
compare										
12:3] +/- 11X 4T			>	•		•				► F
Local Host:	Remote Hos	t: Working Di	rectory:/mttest	Compare: off Filte	ers:	off Warning	3			2/48

3. 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.

		🔤 test. 1.	er - Oracle Develo	per Studio Performanc	e Ana	lyzer				
<u>File</u> <u>Views</u> <u>M</u> etrics	<u>T</u> ools <u>H</u> elp									
🛃 🖾 🕮 🖉 🛛 🏹						Find: Find text	in view	▼ 🔍 🔍	🗌 Mat <u>c</u> h	Case
Vie <u>w</u> s +	Total C	PU Time	Instructions	Name	III	Selection Det	ails			
Welcome	21 EXCLUSIVE	11 INCLUSIVE	21 EXCLUSIVE			Name: do	work			
	sec.	sec.	#	•		PC Address: 2:	- 0x00004850)		
Overview	0.	39,748	0	cache trash odd	A	Size: 36	n n			
Functions >	0.	12.008	0	calladd		Cauraa Sila ada				
	12.018	12.018	11 587 620 362	compute		Source File: mt	test.c			
Timeline	12.038	12.038	11 523 600 360	computeA		Object File: mt	test (foun	d as test.l.e	⊧r/archiv	es/mt
	70.940	70.940	11 651 640 364	computeB		Load Object: mt	test (foun	d as test.l.e	er/archiv	es/mt
Call Tree	12.008	12.008	11 523 600 360	computeC		angled Name:				
Fourse	12.008	12.008	11 587 620 362	computeD		Alianas				
Source	12.038	12.038	11 523 600 360	computeE		Allases:				
Disassembly	2.632	12.008	10 275 210 321	computeF				21 Exclusive		
	12.028	12.028	11 523 600 360	computeG		Total Thread T	ime	0.570 (0.17%)	
Callers-Calle	12.038	12.038	11 523 600 360	computeH		Tabal COUT	inte.	0.570 (0.200)	
	12.008	12.008	11 523 600 360	computeI		Total CPU I	ime:	0.570 (0.29%)	
Experiments	0.	12.028	0	cond_global		User CPU T	'ime:	0.570 (0.29%)	
Threads	0.	0.	0	cond_sleep_queue		System CPU T	'ime:	Θ. (0. %)	
meaus	Θ.	0.	0	cond_timedwait	_	Tran CPU T	ime	0 (0 %)	
Processes	Θ.	12.038	0	cond_timeout_global		napero i		0. (0. 0)	
Frocesses	Θ.	Ο.	0	cond_wait		pata Page Fault I	ime:	θ. (0. %)	
Memory pag	0.	0.	0	cond_wait_common		Text Page Fault T	ime:	0. (0.%)	
1 1 3	0.	0.	0	cond wait queue		rool Rogo Foult T	imo	0 (0 %)	

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 doubleclicking 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.

		test.1.er - Oracl	cle Developer Studio Performance Analyzer
Eile Views Metrics	<u>T</u> ools <u>H</u> elp		
🗟 🖾 🛱 💙 🛛 🏹	• 🕞 🙆 🛛 < >		Fi <u>n</u> d: 🛛 Find text in view 🔷 😡 🖓 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total CPU	Instructions	mttest.c III 🗛
Welcome	11 INCLUSIVE	11 INCLUSIVE	
Overview	sec.	#	<function: computea=""></function:>
Functions	Θ.	0	1422. { 1423. long long i:
Timeline	0.	0	1424. x->sum_ctr = 0;
Call Tree			Source loop below has tag L12
Source >	12.038	11 523 600 360 0	1425. for (i = 0; i < loop_count; i++) { x->sum_ctr = x->sum_ctr + 1.(1426. }
Disassembly			1427
Callore Callo			1429. computeB(workStruct_t *x)
Callers-Calle	-		<function: computeb=""></function:>
Experiments	0.	0	1430. {
Threads	0.	0	1431. $x \to sum_ctr = 0;$
Processes			Source loop below has tag 113
Memory pag	70.940	11 651 640 364	1433. for (i = 0; i < loop_count; i++) { x->sum_ctr = x->sum_ctr + 1.(
· · · · ·	0.	0	1435.
More			1436. void
			1437. computeC(workStruct_t *x) <function: computec=""></function:>
	0.	O	1438. {
			1439. long long i;
	0.	O	1440. x->sum_ctr = 0;
			Source loop below has tag L14
	12.008	11 523 600 360	1441. for (i = 0; i < loop_count; i++) { x->sum_ctr = x->sum_ctr + 1.(
F <u>i</u> lters	Ο.	0	1442. }
bcx =			1445. 1444. void
To add a filter,			1445. computeD(workStruct_t *x)
select a row from a	0		<function: computed=""></function:>
view (such as	υ.	U	1440. i
Com <u>p</u> are	0.	0	1448. x->sum_ctr = 0;
12.3 +/- 1.1X ↓↑ 〓			
Local Host:	Remote Host:	Working Directory: .	/mttest Compare: off Filters: off 🔥 Warning 1491/1824

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 88.

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.

Derived and Other Metrics											
Instructions Per Cycle: 0.201											
Cycles Per Instruction: 4.981											V
▼ HW Counter Profiling											
Memoryspace Hardware Counters)										
L1 D-cache Misses: 1622907507											
Instructions Executed: 141900334433											V
CPU Cycles Time: 196.328 Seconds											V

2. Return to the Source view at computeB().



Note that the Incl. CPU Cycles time and the Incl. Total CPU Time are roughly equivalent in each of the compute*() functions. This indicates that the clock-profiling and CPU Cycles hardware counter profiling are getting similar data.

In the screen shots, the Incl. CPU Cycles and the Incl. Total CPU Time are about 12 seconds for each of the compute*() functions except computeB(). You should also see in 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.

Understanding Cache Contention and Cache Profiling Metrics



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

	3	test.l.er - Ora	cle Developer Stu	dio Performance	Analyzer	
<u>File Views Metrics</u>	<u>T</u> ools <u>H</u> elp					
🖬 🕼 🛱 🌄 🕅	©			Fi <u>n</u>	d: Find text in view	💌 🔗 🔍 🗌 Mat <u>c</u> h Case
Vie <u>w</u> s 🔶	Total C	PU Time	L1 D-cache Misses	Name		III
Welcome			CLUSIVE	-		
Overview	0.	31,192		cache trash ever	n	A
Functions 1	0.	39.748	0	cache trash odd		
Functions	Θ.	12.008	0	calladd		
Timeline	12.018	12.018	0	compute		
	12.038	12.038	0	computeA		
Call Tree	70.940	70.940	1 504 470 470	computeB		
	12.008	12.008	0	computeC		
Source	12.008	12.008	0	computeD		
Disassambly	12.038	12.038	0	computeE		
Disassembly	2.632	12.008	0	computeF		
Callers-Calle	12.028	12.028	0	computeG		
cultors-cultorn	12.038	12.038	0	computeH		
Experiments	12.008	12.008	0	computeI		
	0.	12.028	0	cond_global		
Threads	0.	0.	0	cond_sleep_queue	e	
D	0.	0.	0	cond_timedwait		
Processes	0.	12.038	0	cond_timeout_glo	obal	
Memory pag	0.	0.	0	cond_wait		
riemory_pagin	U.	0.	0	cond_wait_common	n	
More	U.	0.	0	cond_wait_queue		
	0.570	12.128	U	do_exit_critica	L	
	0.370	197.728	0	lock alebel		
	0.	12.010	0	lock_gtobat		
	0.	12.030	0	lock_tocat		
	0.	12.030	0	locktest		
	0.	0.010	0	lwn wait		
	0.	12.038	0	main		v
	Called by (C	-11-				
F <u>i</u> lters	Called-by / C	ans		computeR		
りCX =	Total C	computeB		Total	computeB	
To add a filter	ATTRIBUTED	is called by		ATTRIBUTED	calls	
select a row from a	sec.	▼		sec.	v	
view (such as	31.192	cache trash even				
-	39.748	cache trash odd				
Compare						
12.3 +/- 1.1X V1 =						
Local Host:	Remote Host	: Working Dire	ctory:/mttest	Compare: off Filt	ers: off 🛛 🔥 Warning	15/48

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.

- 3. Go back to the Source view and note that in computeB() the cache misses are in the single line loop.
- 4. 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.

Time L1 D-cache Misses XI INCLUSIVE sec. #	mttest.c		
	Source loop below has ta	g L13	
	1433. for (1 = (0; i < loop_count; i++) { x->su	m_ctr = x->sum_ctr + 1.0; }
0. 0	[<u>1433</u>] 100005780:	ldx [%02 + 840], %03	<scalars>.{long_long loop_count}</scalars>
0. 0	[<u>1433</u>] 100005784:	brlez,pn %03, <u>0x1000057c4</u>	
0. 0	[<u>1432</u>] 100005788:	tzerod %t0	
0. 0	[<u>1433</u>] 10000578c:	sethi %hi(0x100000), %o	1
0. 0	[<u>1433</u>] 100005790:	clr %g3	
0. 0	[1433] 100005794:	or %ol, 5, %g5	
0. 0	[<u>1433</u>] 100005798:	ldd [%00], %f2	{structure:workStruct_t -}.{double sum_ctr}
0. 0	[<u>1430</u>] 10000579c:	add %o2, 840, %g2	
0. 0	[<u>1433</u>] 1000057a0:	sllx %g5, 12, %g4	
0. 0	[<u>1433</u>] 1000057a4:	ldd [%g4 + 1856], %f6	
0. 0	[<u>1433</u>] 1000057a8 *	 dranch target>	<===<<<
49.465 3 201 001	[<u>1433</u>] 1000057a8 :	faddd %f2, %f6, %f4	
11.088 0	[<u>1433</u>] 1000057ac:	std %f4, [%o0]	{structure:workStruct_t -}.{double sum_ctr}
4.643 0	[<u>1433</u>] 1000057b0:	inc %g3	
1.481 0	[<u>1433</u>] 1000057b4:	ldx [%g2], %g1	<scalars>.{long_long loop_count}</scalars>
0. 0	[<u>1433</u>] 1000057b8:	cmp %g3, %g1	
1.441 0	[<u>1433</u>] 1000057bc:	bl,a,pt %xcc, <u>0x1000057a8</u>	
2.822 1 501 269 469	[<u>1433</u>] 1000057c0:	ldd [%00], %f2	{structure:workStruct t -}.{double sum ctr}
	1434. }		
0. 0	[<u>1434</u>] 1000057c4 *	 dranch target>	<===<<<
0. 0	[<u>1434</u>] 1000057c4 :	retl	
0. 0	[<u>1434</u>] 1000057c8:	nop	
	1435.		
	1436. void		
	1437. computeC(workStrue	ct_t *x)	
	1438. {		
	<function: compute<="" td=""><td>eC></td><td></td></function:>	eC>	
0. 0	[<u>1438</u>] 100005800*	 dranch target>	<===<<<
0. 0	[<u>1438</u>] 100005800:	sethi %hi(0x100000), %o	5
	1439. long long	i;	
	1440. x->sum_ct	r = 0;	
0. 0	[<u>1440</u>] 100005804:	clrx [%00]	{structure:workStruct_t -}.{double sum_ctr}
0. 0	[<u>1438</u>] 100005808:	or %05, 263, %04	
0. 0	[1438] 10000580c:	sllx %04, 12, %02	

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
dbranch 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 \rightarrow Settings to open the Settings dialog, and select the Views tab in that dialog.

		🚰 Settings	X
Views	Metrics Timeline Source	e/Disassembly Call Tree	Formats Search Path Pathmaps
	<u>S</u> tandard Views	Index Object Vie <u>w</u> s	Memory Object Views
	Functions Timeline Call Tree Source/Disassembly Source Lines Disassembly PCs DataObjects DataLayout Callers-Callees Statistics Experiments	Threads CPUs Samples GCEvents Seconds Processes Experiment IDs Data Size Duration T5_Chip T5_Core Add Custom View	Memory_page_size Memory_page Memory_648_cacheline Memory_328_cacheline Memory_address Memory_in_home_lgrp Memory_lgrp Physical_928_cacheline Physical_328_cacheline Physical_address Add Custom View Machine Model: t5
Exp <u>o</u> rt	<u>I</u> mport		OK <u>A</u> pply Close <u>H</u> elp

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.

🔽 🔤 test.1.er – Oracle Developer Studio Performance Analyzer 🧧 🔲 🔀										
<u>F</u> ile <u>V</u> iews <u>M</u> etrics <u>T</u> ools <u>I</u>	<u>H</u> elp									
🗟 🕼 🗮 😕 🛛 🖓 🚱	🕄 🚾 🛍 💙 🚱 🦳 🙀 Find text in view 🔽 😡 🗌 Match Case									
Vie <u>w</u> s 🔶	Total CPU Time	L1 D-cache Misses	Name	III 🕯						
Source	VALUES	VALUES								
	sec. 🔻	#								
Disassembly	70.940	384 120 120	<total></total>							
Callers-Callees	19.874	384 120 120	Process 1, Thread 10)						
Callers-Callees	19.874	0	Process 1, Thread 11							
Experiments	15.601	0	Process 1, Thread 12	2						
	15.591	0	Process 1, Thread 13	}						
Threads >										
Processes										
Memory_32B_cac										
Memory_address										
More										
F <u>i</u> lters: 2 active filters										
୭୯× =										
2: Memory_address: 1: Functions: Selecte										

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

7. 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.

8. 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.

9. Select the Memory_32B_cacheline view in the Views navigation panel.

🔽 🔄 test. 1. er	- Oracle Develope	r Studio Performance Analyzer	
File Views Metrics Tools	<u>H</u> elp	,	
S 🕼 🤮 👋 🏹 🚱 🧕		Find: Find text in view 🔽 💫 🔍 M	lat <u>c</u> h Case
Vie <u>w</u> s 📀	L1 D-cache	Name	III
Source	VALUES		
Disassambly	#	▼	
Disassembly	1 504 470 470	<total></total>	
Callers-Callees	3 201 001	<unknown></unknown>	
	733 029 229	Memory_32B_cacheline_0x0000000100143200	
Experiments	708 240 240	Memory_32B_cachetine 0x0000000100143220	
Threads			
Processes			
Memory_32B_cac >			
Memory_address			
More			
Filters: 1 active filter			
୭৫× ≡			
1: Functions: Selecte			

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 97
- "Setting Up the mttest Sample Code" on page 99
- "Collecting Data from mttest for Synchronization Tracing Tutorial" on page 100
- "Examining the Synchronization Tracing Experiment for mttest" on page 100

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 OracleDeveloperStudio12.5-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.l.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.



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.

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.

Name	Options for Name Sort This Metric By	Þ		Sync Wa INCLUSIVE sec.	Sync \$\$ INCLUSIVE #
<tota< th=""><th>Move This Metric To</th><th>•</th><th>В</th><th>Before Metric 🕨</th><th>60</th></tota<>	Move This Metric To	•	В	Before Metric 🕨	60
	Other Metric Options		А	After Metric 🕩	Total CPU Time
	Format	•	E		Name
nttest	More Metrics	•	nii	ng	Sync Wait Time
	Metrics Settings	Ctrl+Shift-M	-		Sync Wait Count
			-		

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 clockprofiling 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.

Total CPU Time		Sync Wait Time	Sync Wait	Name	III	Selection De	etails					
			Count			Name:	do worl	k				
ST EXCLUSIVE	1 INCLUSIVE	CLUSIVE 11				PC Address:	2:0x00	004B50				
sec.	sec. 🔻	sec.	#			Circo	200	004000				
196.788	196.788	137.687	60	<total></total>	۸	Size:	300					
0.660	196.788	56.814	23	do work		Source File:	mttest	. c				
0.	184.819	56.814	22	_lwp_start		Object File: I	nd as	test.l.er/a	archiv	es/m	ttest_gV8	BvCyH
0.	70.519	0.	0	cache_trash	ш	Load Object:	mttest	(found as	test.	l.er	/archives	a∕mtte
70.519	70.519	0.	0	computeB		angled Name						
0.	35.435	0.	0	cache_trash_odd	ш	angica Name.						
0.	35.085	0.	0	cache_trash_even		Allases:						
0.741	29.891	υ.	U	Trylock_global	ш			21 Exc	lusivo		tt Inc	clusive
4.223	17.192	0.	U	mutex_trylock		Total Thread	d Time	0,660	(0)	208-1	252 507	(75
10.000	12.909	0.	0	toke deferred direct		Total Inicat	u mine.	0.000	(0.2	20-07	200.007	(75
12.909	12.909	0.	0	computed	•	Total CPU	J Time:	0.660	(0.3	34%)	196.788	(100
11.970	11.970	17.051	11	cond timeout global		User CPL	J Time:	0.660	(0.3	34%)	196.788	(100
0.	11.978	90.972	38	start		System CPI	J Time:	0.	(0.	%)	Θ.	(0
0.	11.968	00.072	0	calladd		Tran CPI	I Time	0	(0	%)	Θ	(0
11 968	11.968	0.	0	computeA		Data Baga Foul	t Time.	0	(0	0.)	0	(0
2,642	11,968	0.	0	computeE		Jaca Fage Faul	it nine:	0.	(0,	-6)	0.	(0
11.968	11,968	0.	0	computeG		Text Page Faul	t Time:	0.	(0.	%)	Ο.	(0
0.	11.968	17.938	3	cond global		rnel Page Faul	t Time:	0.	(0.	%)	Ο.	(0
0.	11.968	0.	0	lock none		Stoppe	d Time:	0.	(0.	%)	0.	(0
0.	11.968	80.872	37	locktest		Wait CPI	I Time	0	(0	8)	۵	(0
0.	11.968	80.872	38	main		Clear	o Time.	0.	(0.	0.)	0.	(0
11.958	11.958	Ο.	0	compute		Siee	p nime:	0.	(0.	-8)	U.	(0
11.958	11.958	Ο.	O	computeD		User Loc	k Time:	0.	(0.	%)	56.810	(100
11.958	11.958	Θ.	0	computeE		Sync Wai	t Time:	0.	(0.	%)	56.814	(41
11.958	11.958	Θ.	0	computeI	-	Sync Wait	Count:	0	(0.	%)	23	(38
4	11 050	17 000		list sists								
Called-by / Ca	Called-by / Calls											
do_work												
Total CP	do_work	≡ Tot	al C × do	work								
ATTRIBUTED	is called by	ATTR	IBUTED cal	Is								
sec. 🔻	-	se	c. 🔻									
184.819	_lwp_start	7	0.519 cad	he_trash	-							
11.968	locktest	2	9.891 try	/lock_global								
		1	1.978 cor	d_timeout_global								
4		1	1.968 cal	ladd	-	4			_	_	_	h
-		1	1 069 000	d alobal	1.1							

3. 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.

4. Switch to the Callers-Callees view.

liews d		Total CPU	Sync	Sync	Name	
Velcome		Time	Wait Time	Wait Count	hunc	
verview		ATTRIBUTED	ATTRIBUTED	ATTRIBUTED		
		104 010	56 914	# 22	lyp start	
unctions		11.968	0.000	1	locktest	
imeline						
all Tree	îĘ					
ource						
isassembly						
allers-Callees			_			
allers-Callees		<	Add	<u>R</u> emove	S <u>e</u> t Head	et Center Set Tail
allers-Callees		0.660	<u>A</u> dd 0.	<u>R</u> emove	S <u>e</u> t Head S do_work	et Center Set Tail
allers-Callees) xperiments hreads rocesses	¢	0.660	Add 0.	<u>R</u> emove	S <u>e</u> t Head S do_work	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory 32B cac	¢	0.660	Add Q.	<u>R</u> emove	<u>Set</u> Head <u>S</u> do_work	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_32B_cac	÷	0.660 70.519 29.901	<u>A</u> dd 0. 0.	Remove 0	Sgt Head S do_work	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_328_cac	÷	0.660 70.519 29.891 11.978	0. 0. 0. 17. 951	<u>Remove</u> 0 0 0	Sgt Head S do_work cache_trash trylock_global cond timeout olobal	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_32B_cac ilters 0 C X =	÷	0.660 70.519 29.881 11.978 11.968	0. 0. 0. 17.951 0.	0 0 0 0 0 11 0	Set Head S do_work cache_trash trylock_global cond_timeout_global calladd	et Center Set Tail
allers-Callees > xperiments hreads rocesses temory_32B_cac jlters g c x = add a filter, select a row	÷	0.660 0.660 70.519 29.891 11.978 11.968 11.968	0. 0. 0. 17.951 0. 17.938	Remove 0 0 11 0 3	Sgt Head S do_work cache_trash trylock_global cond_timeout_global calladd cond_global	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_32B_cac jters @ C X = o add a filter, select a row rom a view (such as	÷	0.660 0.660 70.519 29.891 11.978 11.968 11.968 11.968	▲dd 0. 0. 17.951 0. 17.938 0.	Remove 0 0 0 0 0 0 0 0 0	Sgt Head S do_work cache_trash trylock_global cond_timeout_global calladd cond_global lock_none	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_32B_cac ilters C C X c o add a filter, select a row rom a view (such as unctions) and then click on	Ŷ	0.660 0.660 70.519 29.891 11.978 11.968 11.968 11.968 11.958	0. 0. 0. 17.951 0. 17.938 0. 17.936	Remove 0 0 0 0 0 0 0 3	Sgt Head S do_work cache_trash trylock_global cond_timeout_global calladd cond_global lock_none lock_global	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_32B_cac ilters o add a filter, select a row rom a view (such as unctions) and ther, click on te toolbar Filters icon.	Ŷ	0.660 0.660 70.519 29.891 11.978 11.968 11.968 11.968 11.958	0. 0. 0. 17.951 0. 17.938 0. 17.938 0.	Remove 0 0 0 0 0 3 0 3 0	Sgt Head S do_work cache_trash trylock_global cond_timeout_global calladd cond_global lock_global lock_global lock_jlocal	et Center Set Tail
allers-Callees > xperiments hreads rocesses lemory_32B_cac ilters o cd a filter, select a row rom a view (such as unctions) and then click on he toolbar Filters icon.	↓ Ĵ	0.660 0.660 70.519 29.891 11.978 11.968 11.968 11.958 11.958	0. 0. 0. 17.951 0. 17.938 0. 17.938 0. 0.	Remove 0 0 0 11 0 3 0 3 0 0	Sgt Head S do_work cache_trash trylock_global cond_timeout_global calladd cond_global lock_none lock_global lock_local nothreads	et Center Set Tail
allers-Callees > xperiments + hreads - rocesses - lemory_32B_cac - jlters - O C X = o add a filter, select a row rom a view (such as unctions) and then click on he toolbar Filters icon.	Ţ Ţ	0.660 70.519 29.891 11.978 11.968 11.968 11.958 11.958 11.958	0. 0. 0. 17.951 0. 17.938 0. 17.936 0. 0. 2.988	Remove 0 0 0 0 0 0 0 3 0 3 0 2	Sgt Head S do_work cache_trash trylock_global cond_timeout_global calladd cond_global lock_none lock_global lock_local nothreads sema_global	et Center Set Tail

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.

	🔄 test	.1.er - Orac	le Develope	r Studio Performance Analyzer 📃 🔲 🛽
<u>F</u> ile <u>V</u> iews <u>M</u> etrics <u>T</u> ools	<u>H</u> elp			
🖾 🕼 🕾 👋 🛛 🏹 🚱	$ $ \leq >			Find: Find text in view 🔽 😽 🗛 🗔 Mat <u>c</u> h Case
Vie <u>w</u> s 📀	Total CPU	Sync	Sync	mttest.c
Welcome	Time	Time	Count	
	11 INCLUSIVE	CLUSIVE 11	11 INCLUSIVE	
Overview	sec.	sec.	#	
Functions				956. /* lock_global: use a global lock to process array's data *.▲ 957. void
Timeline				958. lock_global(Workblk *array, struct scripttab *k)
				<function: lock_global=""></function:>
Call Tree	0.	0.	O	959. {
				960. /* acquire the global lock */
Source				961.
Disassembly				962. #ITORT SULARIS
,				965. mulex_lock(aglobal_lock);
Callers-Callees				965 #ifdef POSTX
Fun estimante	0.	17.954	3	966. pthread mutex lock(&global lock):
Experiments				967. #endif
Threads				968.
	0.	0.	0	969. array->ready = gethrtime();
Processes	0.	0.	0	970. array->vready = gethrvtime();
				971.
More	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 \widehat{V} 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 \rightarrow 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. You can see that the thread state bar is green for that thread which means all its time was spent in User CPU Time, with none in User Lock Time. 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 \sim 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 \sim 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 workblock, 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.

Click the Compare Experiments button [□] on the tool bar or choose File → Compare Experiments.

The Compare Experiments dialog box opens.

Compare Experiments	X							
Comparing experiments. The Comparison experiment(s) are compared against the Baseline experiment. Most data views support comparing experiments. When you compare experiments, Performance Analyzer displays data from the experiments or groups in adjacent columns. The Advanced option allows grouping (aggregating) of experiments before comparing them.								
More More	∜ <u>R</u> everse							
Baseline Experiment:								
test.1.er	▼							
Comparison Experiment <u>1</u> :	•							
A <u>d</u> vanced OK Cancel	<u>H</u> elp							

The test.l.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, popup 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.

4. 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		11 INCLUSIVE		1 INCLUSIVE		11 INCLUSIVE		
Baseline	Exp. 1	Baseline	Exp. 1	Baseline	Exp. 1	Baseline	Exp. 1	
sec.	sec.	sec.	sec.	sec.	sec.	#	# 🔻	
198.799	200.700	198.799	200.700	139.386	138.784	58	169	<total></total>
0.560	0.570	198.799	200.690	56.878	56.896	21	131	do work
Θ.	0.	186.811	188.702	56.878	56.896	21	126	lwp_start
Θ.	0.	Θ.	0.	0.000	0.003	1	77	fetch_work
Θ.	0.	11.988	11.998	82.507	81.888	37	43	start
Θ.	Θ.	11.988	11.998	82.507	81.888	37	43	main
Θ.	0.	11.988	11.998	82.507	81.888	36	41	locktest
Θ.	0.	Θ.	0.	82.507	81.888	36	36	thread work
Θ.	Θ.	11.998	11.998	17.969	17.972	11	27	cond timeout global
Θ.	0.	11.978	11.978	17.964	17.967	3	15	cond global
Θ.	Θ.	11.968	11.978	17.954	17.961	3	4	lock global
Θ.	0.	11.968	11.988	Ο.	0.	0	4	lock_local
Θ.	Θ.	11.998	11.968	2.991	2.992	3	4	sema global
Θ.	Ο.	Ο.	0.	0.000	0.000	1	2	resolve symbols
Θ.	0.	0.010	0.	Θ.	0.	0	0	collector_write_packet
Θ.	Θ.	Θ.	0.010	Θ.	Θ.	O	O	cond timedwait
Θ.	0.010	Ο.	0.010	Ο.	0.	0	0	lwp_park
Ο.	0.	Θ.	0.	Θ.	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.

		test.1.er,	Oracle De	eveloper Stud	io Performa	nce Analyz	er		📃 🔲 🛛
<u>File Views Metrics Tools</u>	<u>H</u> elp								
🖬 🕼 🛱 🌄 🛛 🖓 🚱						Find: Find f	text in viev	v 🔽 🔍 🔍 🛛	Mat <u>c</u> h Case
Vie <u>w</u> s 🔶		Total CP	U Time	Sync Wa	it Time	Sync Wa	it Count	Name	Ш
Timeline		ATTRIB	UTED	ATTRIE	UTED	ATTRIE	BUTED		
		Baseline	Exp. 1	Baseline	Exp. 1	Baseline	Exp. 1		
Call Tree		106 011	199 702	56 070	56 006	# •	# 126	lwn_start	
Source		11.988	11.988	0.	0.	0	5	locktest	
Disassembly									
Callers-Callees	12								
Experiments									
Threads									
Processes									
More		<	≥	Add Rer	nove	S <u>e</u> t Head	<u>S</u> et Ce	nter Set Tail	
Filters		0.560	0.570	Θ.	0.	0	0	do work	
-267 =									
To add a filter, select a row									
Trom a view (such as									
the toolbar Filters icon									
		11.998	11.998	17.969	17.972	11	27	cond timeout ald	bal
		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	
		Θ.	Θ.	0.000	0.003	1	77	fetch work	
		72.481	74.322	0.	Θ.	0	0	cache trash	
	îe	11.978	11.968	0.	Θ.	0	0	calladd	
Compare		11.968	11,988	0.	Ο.	0	4	lock local	
12.2 +/- 11Y IA =		11.968	12.018	0.	0.	0	0	lock none	
12.3 T/- UK +I =		11.978	11.978	0.	0	0	Ö	nothreads	
Baseline: test.1.er		29,921	29,921	0.	0.	0	õ	trylock global	
Evp 1: test 2 or		201021	10.021		2.	0	v	Let A cook_dcopare	
chp. 1. test. 2.el									
Local Host: Rem	ote Ho	st: Workin	g Directory:/r	nttest Comp	are: on Filt	ers: off 🥖	Warning		

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.

112 Oracle Developer Studio 12.5: Performance Analyzer Tutorials • June 2016

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 113
- "Additional Tutorials" on page 114
- "More Information" on page 115

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 12.5: 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:

	🔄 Connect to Remote Host	×
H <u>o</u> st Name:	yourhost 💌	
Authentication:	Public key, Password	<u>M</u> anage
<u>U</u> ser Name:	your user name	
<u>P</u> assword:	•••••	
Installation Path:	Path to bin directory where Studio is installed on 'yourhost'	
Connection Status	1	
	Connect Cancel Close	e <u>H</u> elp

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.

Additional Tutorials

As mentioned in the "Introduction to the Performance Analyzer Tutorials", there are several other tutorials in the PerformanceAnalyzer subdirectory of the samples zip file you downloaded in "Getting the Sample Code for the Tutorials" on page 10. The following list gives more information on each of these tutorials:

cachetest	The cachetest tutorial explores the effects of compiler optimization on the performance of a code, the presentation of compiler commentary from the Oracle Developer Studio compilers to help you understand the optimizations.
ksynprog	The ksynprog tutorial explores running tasks that trigger a path through the kernel and collecting performance data on the resulting program.

omptest	The omptest tutorial explores using the OpenMP parallelization directives and the resulting performance characteristics that can be examined in Performance Analyzer.
synprog	The synprog tutorial explores a simple program that does a number of tasks and exhibits some performance characteristics or features of the Performance Analyzer.

More Information

The following resources give more information on Performance Analyzer and the related datacollection tools:

- Integrated help system in Performance Analyzer
- Oracle Developer Studio 12.5: Performance Analyzer
- Articles and white papers available on the Oracle Developer Studio developer portal (http: //www.oracle.com/technetwork/server-storage/solarisstudio/).
- Chapter 5, "Performance Analysis Tools" in *What's New in the Oracle Developer Studio* 12.5 *Release*
- "Performance Analyzer and er_print Utility Limitations" in *Oracle Developer Studio 12.5: Release Notes*