Analyzing Program Performance With Sun WorkShop

Forte Developer 6 update 2 (Sun WorkShop 6 update 2)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-7989-10
July 2001, Revision A

Send comments about this document to: docfeedback@sun.com

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road • Palo Alto, CA 94303-4900 USA. All rights reserved.

This product or document is distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. For Netscape™, Netscape Navigator™, and the Netscape Communications Corporation logo™, the following notice applies: Copyright 1995 Netscape Communications Corporation. All rights reserved.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook2, Solaris, SunOS, JavaScript, SunExpress, Sun WorkShop, Sun WorkShop Professional, Sun Performance Library, Sun Performance WorkShop, Sun Visual WorkShop, and Forte are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Sun f90/f95 is derived from Cray CF90™, a product of Cray Inc.

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road • Palo Alto, CA 94303-4900 Etats-Unis. Tous droits réservés.

Ce produit ou document est distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd. La notice suivante est applicable à Netscape™, Netscape Navigator™, et the Netscape Communications Corporation logo™: Copyright 1995 Netscape Communications Corporation. Tous droits réservés.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook2, Solaris, SunOS, JavaScript, SunExpress, Sun WorkShop, Sun WorkShop Professional, Sun Performance Library, Sun Performance WorkShop, Sun Visual WorkShop, et Forte sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

Sun f90/f95 est derivé de CRAY CF90™, un produit de Cray Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Please
Recycle

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	**New Product Name**
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Before You Begin 1

How This Book Is Organized 1

Typographic Conventions 2

Shell Prompts 3

Supported Platforms 3

Accessing Sun WorkShop Development Tools and Man Pages 3

Accessing Sun WorkShop Documentation 5

Accessing Related Documentation 6

Ordering Sun Documentation 6

Sending Your Comments 6

1. Overview of Performance Profiling and Analysis Tools 7

2. Learning to Use the Sampling Collector and Performance Analyzer 9

Setting Up the Examples for Execution 10

System Requirements 10

Choosing Alternative Compiler Options 11

Running the Sampling Collector and Performance Analyzer 11

Example 1: Basic Performance Analysis 13

Collecting Data for `synprog` 14

Simple Metric Analysis 14

Extension Exercise for Simple Metric Analysis 17

The `gprof` Fallacy 17

The Effects of Recursion 19

Loading Dynamically Linked Shared Objects 22

Example 2: OpenMP Parallelization Strategies 23

Collecting Data for `omptest` 24

Comparing Parallel Sections and Parallel Do Strategies 26

Comparing Critical Section and Reduction Strategies 28

Example 3: Locking Strategies in Multithreaded Programs 29

Collecting Data for `mttest` 29

How Locking Strategies Affect Wait Time 30

How Data Management Affects Cache Performance 33

Extension Exercises for `mttest` 36

Example 4: Cache Behavior and Optimization 36

Collecting Data for `cachetest` 37

Setting Up the Display 38

Execution Speed 39

Program Structure and Cache Behavior 39

Program Optimization and Performance 41

3. Collecting Performance Data 45

What the Sampling Collector Collects 45

Clock-Based Data 46

Synchronization Wait Tracing Data 46

Hardware-Counter Overflow Data 47

Sample Points 49

Global Information 50

Where the Data Is Stored 50

Estimating Storage Requirements 51

Controlling Data Collection From Your Program 52

Compiling Your Program for Data Collection and Analysis 54

Limitations on Your Program 55

Collecting Data Using the `collect` Command 55

Data Collection Options 56

Experiment Control Options 58

Output Options 59

Other Options 60

Collecting Data From the Sun WorkShop Integrated Programming Environment 60

Collecting Data Using the `dbx collector` Subcommands 64

Data Collection Subcommands 64

Experiment Control Subcommands 67

Output Subcommand 67

Information Subcommands 68

Obsolete Subcommands 68

Collecting Data From a Running Process 69

Collecting Data From MPI Programs 71

Storing MPI Experiments 71

Running the `collect` Command Under MPI 73

Collecting Data by Starting `dbx` Under MPI 73

4. Analyzing Program Performance Using the Performance Analyzer Graphical Interface 75

Types of Performance Metrics and How to Use Them 76

Timing Metrics 76

Synchronization Delay Metrics 77

Count Metrics 77

How Metrics Are Assigned to Program Structure 78

Function-Level Metrics: Exclusive, Inclusive, and Attributed 78

Interpreting Function-Level Metrics: An Example 79

How Recursion Affects Function-Level Metrics 80

Running the Performance Analyzer 81

Selecting Experiments 83

Loading an Experiment Into the Performance Analyzer 84

Adding Experiments to the Performance Analyzer 84

Dropping Experiments From the Performance Analyzer 85

Filtering the Data to Be Displayed 85

Selecting Experiments, Samples, Threads, and LWPs 85

Selecting Load Objects 87

Examining Metrics for Functions and Load Objects 87

Selecting Metrics and Sort Order for Functions and Load Objects 88

Viewing Summary Metrics for a Function or a Load Object 90

Searching for a Function or Load Object 91

Examining Callers-Callees Metrics for a Function 92

Selecting Metrics and Sort Order in the Callers-Callees Window 94

Examining Annotated Source Code and Disassembly Code 95

Generating and Using a Mapfile 97

Examining Information About Samples 99

Examining Execution Statistics 101

Examining Address-Space Information 102

Printing the Display 103

5. Analyzing Program Performance Using the `er_print` Command Line Interface 105

`er_print` Syntax 106

Metric Lists 106

Function List Commands 109

Callers-Callees List Commands 111

Source and Disassembly Listing Commands 112

Filtering Commands 114

Selection Lists 114

Selection Commands 115

Listing of Selections 116

Metric List Commands 117

Defaults Commands 118

Output Commands 119

Other Display Commands 120

Mapfile Generation Command 120

Control Commands 121

Information Commands 121

6. Understanding the Performance Analyzer and Its Data 123

Interpreting Performance Metrics 123

Clock-Based Profiling 124

Synchronization Wait Tracing 126

Hardware-Counter Overflow Profiling 127

Call Stacks and Program Execution 128
Single-Threaded Execution and Function Calls 128
Explicit Multithreading 131
Parallel Execution and Compiler-Generated Body Functions 132
Mapping Addresses to Program Structure 136
The Process Image 136
Load Objects and Functions 137
Aliased Functions 137
Non-Unique Function Names 138
Static Functions From Stripped Shared Libraries 138
Fortran Alternate Entry Points 139
Inlined Functions 139
Compiler-Generated Body Functions 140
Outline Functions 141
The <Unknown> Function 141
The <Total> Function 142
Annotated Code Listings 142
Annotated Source Code 143
Annotated Disassembly Code 144

7. Manipulating Experiments and Viewing Annotated Code Listings 147
Manipulating Experiments 147
Viewing Annotated Code Listings With `er_src` 148
Other Utilities 150
The `er_archive` Utility 150
The `er_export` Utility 151

A. Profiling Programs With `prof`, `gprof`, and `tcov` 153

Using `prof` to Generate a Program Profile 154

Using `gprof` to Generate a Call Graph Profile 156

Using `tcov` for Statement-Level Analysis 159

Creating `tcov` Profiled Shared Libraries 163

Locking Files 163

Errors Reported by `tcov` Runtime Routines 164

Using `tcov` Enhanced for Statement-Level Analysis 165

Creating Profiled Shared Libraries for `tcov` Enhanced 166

Locking Files 167

`tcov` Directories and Environment Variables 167

Index 169

Figures

FIGURE 3-1 The Sampling Collector Window 61

FIGURE 4-1 Call Tree Illustrating Exclusive, Inclusive, and Attributed Metrics 80

FIGURE 4-2 The Analyzer Window 82

FIGURE 4-3 The Filters Dialog Box 86

FIGURE 4-4 The Function List Metrics Dialog Box 89

FIGURE 4-5 The Summary Metrics Window 90

FIGURE 4-6 The Find Dialog Box 91

FIGURE 4-7 The Callers-Callees Window 92

FIGURE 4-8 The Callers-Callees Metrics Dialog Box 94

FIGURE 4-9 The Create Mapfile Dialog Box 97

FIGURE 4-10 The Overview Display 99

FIGURE 4-11 The Sample Details Window 100

FIGURE 4-12 The Execution Statistics Display 101

FIGURE 4-13 The Address Space Display 102

FIGURE 4-14 The Page Properties Window 103

FIGURE 6-1 Schematic Call Tree for a Multithreaded Program That Contains a Parallel Do Construct 134

FIGURE 6-2 Schematic Call Tree for a Parallel Region With a Worksharing Do Construct 135

Tables

TABLE 3-1 Aliased Hardware Counters Available on SPARC and IA Hardware 48

TABLE 3-2 Environment Variable Settings for Preloading the Library `libcollector.so` 70

TABLE 4-1 Options for the `Analyzer` Command 82

TABLE 5-1 Options for the `er_print` Command 106

TABLE 5-2 Metric Type Characters 107

TABLE 5-3 Metric Visibility Characters 107

TABLE 5-4 Metric Name Strings 108

TABLE 6-1 How Kernel Microstates Contribute to Metrics 124

TABLE 6-2 Annotated Source-Code Metrics 143

TABLE A-1 Performance Profiling Tools 153

Before You Begin

This manual describes the performance analysis tools that are available with the Sun WorkShop™ product.

- The Sampling Collector and Performance Analyzer are a pair of tools that perform statistical profiling of a wide range of performance data and relate it to program structure at the function, source line and instruction level.
- `prof` and `gprof` are tools that perform statistical profiling of CPU usage and provide execution frequencies at the function level.
- `tcov` is a tool that provides execution frequencies at the function and source line levels.

This manual is intended for application developers with a working knowledge of Fortran, C, or C++, the Sun WorkShop integrated programming environment, the Solaris™ operating environment, and UNIX® operating system commands. Some knowledge of performance analysis is helpful but is not required to use the tools. The profiling tools `prof`, `gprof`, and `tcov` do not require a working knowledge of the Sun WorkShop integrated programming environment.

How This Book Is Organized

Chapter 1 introduces the performance analysis tools, briefly discussing what they do and when to use them.

Chapter 2 is a tutorial that demonstrates how to use the Sampling Collector and Performance Analyzer to assess the performance of four example programs.

Chapter 3 describes how to use the Sampling Collector to collect timing data, synchronization delay data, and hardware event data from your program.

Chapter 4 describes how to use the Performance Analyzer graphical interface to analyze the data collected by the Sampling Collector.

Chapter 5 describes how to use the `er_print` command line interface to analyze the data collected by the Sampling Collector.

Chapter 6 describes the process of converting the data collected by the Sampling Collector into performance metrics and how the metrics are related to program structure.

Chapter 7 presents information on the utilities that are provided for manipulating and converting performance experiments and viewing annotated source code and disassembly code without running an experiment.

Appendix A describes the Unix profiling tools `prof`, `gprof`, and `tcov`. These tools provide timing information and execution frequency statistics.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Supported Platforms

This Sun WorkShop™ release supports versions 2.6, 7, and 8 of the Solaris™ *SPARC™ Platform Edition* and Solaris™ *Intel Platform Edition* operating environments.

Accessing Sun WorkShop Development Tools and Man Pages

The Sun WorkShop product components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the Sun WorkShop compilers and tools, you must have the Sun WorkShop component directory in your `PATH` environment variable. To access the Sun WorkShop man pages, you must have the Sun WorkShop man page directory in your `MANPATH` environment variable.

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` and `MANPATH` variables to access this release, see the *Sun WorkShop 6 update 2 Installation Guide* or your system administrator.

Note – The information in this section assumes that your Sun WorkShop 6 update 2 products are installed in the `/opt` directory. If your product software is not installed in the `/opt` directory, ask your system administrator for the path on your system.

Accessing Sun WorkShop Compilers and Tools

Use the steps below to determine whether you need to change your `PATH` variable to access the Sun WorkShop compilers and tools.

To Determine If You Need to Set Your `PATH` Environment Variable

1. **Display the current value of the `PATH` variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing `/opt/SUNWspro/bin/`.**

   If you find the path, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the path, set your `PATH` environment variable by following the instructions in the next section.

To Set Your `PATH` Environment Variable to Enable Access to Sun WorkShop Compilers and Tools

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `PATH` environment variable.**

   `/opt/SUNWspro/bin`

Accessing Sun WorkShop Man Pages

Use the following steps to determine whether you need to change your `MANPATH` variable to access the Sun WorkShop `man` pages.

To Determine If You Need to Set Your `MANPATH` Environment Variable

1. **Request the `workshop` man page by typing:**

```
% man workshop
```

2. **Review the output, if any.**

   If the `workshop`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next section for setting your `MANPATH` environment variable.

To Set Your `MANPATH` Environment Variable to Enable Access to Sun WorkShop Man Pages

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `MANPATH` environment variable.**

   `/opt/SUNWspro/man`

Accessing Sun WorkShop Documentation

You can access Sun WorkShop product documentation at the following locations:

- **The product documentation is available from the documentation index installed with the product on your local system or network.**

  Point your Netscape™ Communicator 4.0 or compatible Netscape version browser to the following file:

  `/opt/SUNWspro/docs/index.html`

  If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- **Manuals are available from the docs.sun.com℠ Web site.**

  The `docs.sun.com` Web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

Accessing Related Documentation

The following table describes related documentation that is available through the `docs.sun.com` Web site.

Document Collection	Document Title	Description
Solaris 8 Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris 8 Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris 8 Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Ordering Sun Documentation

You can order product documentation directly from Sun through the `docs.sun.com` Web site or from Fatbrain.com, an Internet bookstore. You can find the Sun Documentation Center on Fatbrain.com at the following URL:

`http://www.fatbrain.com/documentation/sun`

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

CHAPTER 1

Overview of Performance Profiling and Analysis Tools

Developing high performance applications requires a combination of compiler features, libraries of optimized routines, and tools for performance analysis. *Analyzing Program Performance With Sun WorkShop* describes the tools that are available to help you assess the performance of your code, identify potential performance problems, and locate the part of the code where the problems occur.

This manual deals primarily with the Sampling Collector and Performance Analyzer, a pair of tools that you use to collect and analyze performance data for your application.

- The Sampling Collector collects performance data using a statistical method called profiling. The data can include call stacks, microstate accounting information, thread-synchronization delay data, hardware-counter overflow data, address space data, and summary information for the operating system. See Chapter 3 for detailed information about the Sampling Collector.
- The Performance Analyzer displays the data recorded by the Sampling Collector, so that you can examine the information. The Performance Analyzer processes the data and displays various metrics of performance at the level of the program, the functions, the source lines, and the disassembly instructions. These metrics are classed into three groups: clock-based metrics, synchronization delay metrics, and hardware counter metrics. The Performance Analyzer can also create a mapfile that you can use to improve the order of function loading in the program's address space. See Chapter 4 for detailed information about the Performance Analyzer.

These two tools help to answer the following kinds of questions:

- How much of the available resources does the program consume?
- Which functions or load objects are consuming the most resources?
- Which source lines and disassembly instructions consume the most resources?
- How did the program arrive at this point in the execution?
- Which resources are being consumed by a function or load object?

The main window of the Performance Analyzer displays a list of functions for the program with exclusive and inclusive metrics for each function. The list can be filtered by load object, by thread, by LWP, and by time slice. For a selected function, a subsidiary window displays the callers and callees of the function. This window

can be used to navigate the call tree—in search of high metric values, for example. Two more windows display source code that is annotated line-by-line with performance metrics and interleaved with compiler commentary, and disassembly code that is annotated with metrics for each instruction and interleaved with both source code and compiler commentary if they are available.

The Sampling Collector and Performance Analyzer are designed for use by any software developer, even if performance tuning is not the developer's main responsibility. These tools provide a more flexible, detailed, and accurate analysis than the commonly used profiling tools `prof` and `gprof`, and are not subject to an attribution error in `gprof`.

The following command-line equivalents of the Sampling Collector and Performance Analyzer are available:

- Data can be collected with the `collect` command, which is described in Chapter 3.
- The Sampling Collector can be run from `dbx` using the `collector` subcommands, which are described in Chapter 3.
- The command-line utility `er_print` prints an ASCII version of the various Performance Analyzer displays. See Chapter 5 for more information.
- The command-line utility `er_src` displays source code listings and disassembly code listings that are annotated with compiler commentary but that do not include performance data. See Chapter 7 for more information.

This manual also includes information about the following performance tools:

- `prof` and `gprof`

  `prof` and `gprof` are UNIX® tools for generating profile data and are included with Solaris™ versions 2.6, 7, and 8 of the Solaris SPARC™ *Platform Edition* and Solaris *Intel Platform Edition*.

- `tcov`

  `tcov` is a code coverage tool that reports the number of times each function is called and each source line is executed.

For more information about `prof`, `gprof`, and `tcov`, see Appendix A.

CHAPTER 2

Learning to Use the Sampling Collector and Performance Analyzer

This chapter shows you how to use the Sampling Collector and the Performance Analyzer. Four example programs are provided that illustrate the capabilities of the Performance Analyzer in several different situations.

- Example 1: Basic Performance Analysis. This example uses a C program, `synprog`, that shows how time is attributed to functions, source lines and instructions, and how the Performance Analyzer handles recursive calls and dynamic loading of object modules.
- Example 2: OpenMP Parallelization Strategies. This example uses a Fortran program, `omptest`, that uses OpenMP directives to demonstrate the efficiency of different approaches to parallelization.
- Example 3: Locking Strategies in Multithreaded Programs. This example uses a C program, `mttest`, that uses explicit multithreading to demonstrate the efficiency of different approaches to scheduling of work among threads.
- Example 4: Cache Behavior and Optimization. This example uses a Fortran 90 program, `cachetest`, that demonstrates the effect of memory access on execution speed using hardware counters.

In the examples, these are the general performance questions that are addressed:

- What resources is the program using?
- Where in the program are most of these resources being used?
- How did the program arrive at a certain place in its execution?

If the compiled code is already as optimized as the compiler can make it, you are now using performance analysis for ways to refine the program algorithm itself to make the program execute more efficiently. Once you determine where the program is using the most resources, the Performance Analyzer offers various ways to examine the code so that you can determine why this part of the program is using the most resources.

Note – The data that you see in this chapter might differ from the data that you see when you run the examples for yourself.

Setting Up the Examples for Execution

The information in this section assumes that your Sun WorkShop™ 6 update 2 products are installed in the `/opt` directory. If your product software is not installed in the `/opt` directory, ask your system administrator for the path on your system.

The source code and makefiles for each of the example programs are in the Performance Analyzer example directory.

```
/opt/SUNWspro/examples/WorkShop/analyzer
```

This directory contains a separate subdirectory for each example, named `synprog`, `omptest`, `mttest` and `cachetest`.

To compile the examples with the default options:

1. **Ensure that the Sun WorkShop software directory** `/opt/SUNWspro/bin` **appears in your path.**
2. **Copy the files in one or more of the example subdirectories to your own work directory using the following commands.**

```
% mkdir ~/work-directory
% cp -r /opt/SUNWspro/examples/WorkShop/analyzer/example \
~/work-directory/example
```

Choose *example* from the list of example subdirectory names given in this section. In this tutorial it is assumed that your directory is set up as described in the preceding code box.

3. **Type** `make` **to compile and link the example program.**

```
% cd ~/work-directory/example
% make
```

System Requirements

The following requirements must be met in order to run the example programs as described in this chapter:

- `synprog` has no special requirements.

- omptest has the following requirements:
  - You must run the program on a SPARC machine with at least four CPUs.
  - You must have a Forte™ for High Performance Computing (HPC) license to use the parallelization features of the Fortran 95 compiler.
- mttest requires that you have access to a machine with at least four CPUs.
- cachetest requires that you run the program on UltraSPARC™ III hardware with at least 160 Mbytes of memory.

Choosing Alternative Compiler Options

The default compiler options have been chosen to make the examples work in a particular way. Some of them can affect the performance of the program, such as the -xarch option, which selects the instruction set architecture. This option is set to native so that you use the instruction set that is best suited to your computer. If you want to use a different setting, change the definition of the ARCH environment variable in the makefile.

If you run the examples on a SPARC platform with the default V7 architecture, the compiler generates code that calls the .mul and .div routines from libc.so rather than using integer multiply and divide instructions. The time spent in these arithmetic operations shows up in the <Unknown> function; see "The <Unknown> Function" on page 141 for more information.

The makefiles for all three examples contain a selection of alternative settings for the compiler options in the environment variable OFLAGS, which are commented out. After you run the examples with the default setting, choose one of these alternative settings to compile and link the program to see what effect the setting has on how the compiler optimizes and parallelizes code. For information on the compiler options in the OFLAGS settings, see the *C User's Guide* or the *Fortran User's Guide*.

Running the Sampling Collector and Performance Analyzer

To collect data for the examples you can use the command line or the graphical user interface (GUI). The sections for each example provide specific instructions for both options. This section describes the general procedures for running the Sampling Collector and the Performance Analyzer from the Sun WorkShop GUI. You can collect data using the command line with the collect command or the dbx collector subcommands, both of which are described in Chapter 3.

Some basic features of the Performance Analyzer are also described in this section.

To open the Sun WorkShop main window, type the following on the command line.

```
% workshop &
```

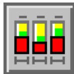
You open the Sampling Collector window from the Debugging window. To open the Debugging window, click the Debugging button in the Sun WorkShop main window.

If you have not previously run the Sun WorkShop GUI, the Debug New Program dialog box is displayed. Enter the name of the example program in the text box or use the list box to navigate to the program. Otherwise, choose Debug ➤ New Program from the Debugging window menu bar to open the Debug New Program dialog box and load the program.

To open the Sampling Collector window, choose Windows ➤ Sampling Collector from the Debugging window menu bar. The Sampling Collector window is shown in FIGURE 3-1 on page 61. In this window, enter the experiment name, choose the types of data to collect and the sampling interval.

To run the program and collect data, click the Start button in the Sampling Collector window or the Debugging window.

To start the Performance Analyzer, click the Analyzer button in the Sampling Collector window or the Sun WorkShop main window.



The Analyzer window is shown in FIGURE 4-2 on page 82.

When the Performance Analyzer is launched from the Sampling Collector window, the experiment that was last collected is automatically loaded.

If the default data options were used in the Sampling Collector, the main window of the Performance Analyzer displays the Function List with the default clock-based profiling metrics, which are:

- Exclusive user CPU time (the amount of time spent in the function itself), in seconds
- Inclusive user CPU time (the amount of time spent in the function itself and any functions it calls), in seconds

The function list is sorted on exclusive CPU time by default. For a more detailed discussion of metrics, see "Types of Performance Metrics and How to Use Them" on page 76.

Selecting a function in the function list and clicking the Callers-Callees button displays the Callers-Callees window, which gives in formation about the callers and callees of a function. The window is divided into three horizontal panes:

- The middle pane shows data for the selected function.
- The top pane shows data for all functions that call the selected function.
- The bottom pane shows data for all functions that the selected function calls.

In addition to exclusive and inclusive metrics, the Callers-Callees window can display attributed metrics for callers and callees. Attributed metrics are the parts of the inclusive metric of the selected function that are due to calls from a caller or calls to a callee.

Example 1: Basic Performance Analysis

This example is designed to demonstrate the main features of the Performance Analyzer using four programming scenarios:

- "Simple Metric Analysis" on page 14 demonstrates how to use the function list, the annotated source code listing and the annotated disassembly code listing to do a simple performance analysis of two routines that shows the cost of type conversions.
- "The `gprof` Fallacy" on page 17 demonstrates how to use the Callers-Callees list and shows how that is time used in a low-level routine is attributed to its callers. `gprof` is a standard UNIX performance tool that properly identifies the function where the program is spending most of its CPU time, but in this case wrongly reports the caller that is responsible for most of that time. See Appendix A for a description of `gprof`.
- "The Effects of Recursion" on page 19 shows how time is attributed to callers in a recursive sequence for both direct recursive function calls and indirect recursive function calls.
- "Loading Dynamically Linked Shared Objects" on page 22 shows how a function is correctly identified even if it is loaded in different locations at different times.

Collecting Data for `synprog`

Read the instructions in the sections, "Setting Up the Examples for Execution" on page 10 and "Running the Sampling Collector and Performance Analyzer" on page 11, if you have not done so. Compile `synprog` before you begin this example.

To collect data for `synprog` and start the Performance Analyzer from the command line, type the following commands.

```
% cd ~/work-directory/synprog
% collect synprog
% analyzer test.1.er &
```

To collect data for `synprog` and start the Performance Analyzer using the GUI:

1. **Open the Debugging window and load** `synprog`**.**
2. **In the Debugging window, choose Debug ➤ Change Debugging Directory to set the current directory to** ~/*work-directory*/`synprog`**.**

   This ensures that the shared objects `so_syn.so` and `so_syx.so` are found when they are needed.
3. **Choose Windows ➤ Sampling Collector and ensure that the experiment name is** `test.1.er`**.**

   This example uses clock-based metrics and the default sampling interval. You do not need to make changes in the data options in the Sampling Collector window.
4. **Click the Start button.**

   The program runs and the Sampling Collector collects data.
5. **When the program is finished, click the Analyzer button.**

   The experiment `test.1.er` is loaded into the Analyzer window.

   You are now ready to analyze the `synprog` experiment using the procedures in the following sections.

Simple Metric Analysis

This section examines CPU times for two functions, `cputime()` and `icputime()`. Both contain a `for` loop that increments a variable `x` by one. In `cputime()`, `x` is a floating-point variable, but in `icputime()`, `x` is an integer variable.

1. **Locate** `cputime()` **and** `icputime()` **in the Function List display.**

Compare the exclusive user CPU time for the two functions. `cputime()` takes much longer to execute than `icputime()`.

The annotated source listing tells you which lines of code are responsible for the CPU time.

2. **Click `cputime()` to select it, and then click Source.**

   A text editor window opens and displays the annotated source code for function `cputime()`. (You might need to resize the text editor window.)

```
                                    400. int
                                    401. cputime(int k)
                                    402. {
                                    403.         int     i;      /* temp value for loop */
                                    404.         int     j;      /* temp value for loop */
                                    405.         volatile float  x;      /* temp variable for f.p. calculation */
                                    406.         hrtime_t        start;
                                    407.         hrtime_t        vstart;
                                    408.
    0.            0.                409.         start = gethrtime();
    0.            0.                410.         vstart = gethrvtime();
                                    411.
                                    412.         /* Log the event */
    0.            0.                413.         wlog("start of cputime", NULL);
                                    414.
    0.            0.                415.         if(k == 0) {
    0.            0.                416.                 k = 80;
                                    417.         }
    0.            0.                418.         for (i = 0; i < k; i ++) {
    0.            0.                419.                 x = 0.0;
    0.840         0.840             420.                 for(j=0; j<1000000; j++) {
## 2.870         2.870             421.                         x = x + 1.0;
                                    422.                 }
                                    423.         }
                                    424.
                                    425.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
    0.            0.                426.                 "cputime", NULL);
    0.            0.                427.         return 0;
```

Most of the time is used by the loop line and the line in which x is incremented.

3. **Click `icputime()` to select it, and then click Source.**

   The source code for `icputime()` replaces the source code for `cputime()` in the text editor. Examine the loop line and the line in which x is incremented.

```
                                    433. int
                                    434. icputime(int k)
                                    435. {
                                    436.         int     i;      /* temp value for loop */
                                    437.         int     j;      /* temp value for loop */
                                    438.         volatile long   x;      /* temp variable for long calculation */
                                    439.         hrtime_t        start;
                                    440.         hrtime_t        vstart;
                                    441.
    0.            0.                442.         start = gethrtime();
    0.            0.                443.         vstart = gethrvtime();
                                    444.
                                    445.         /* Log the event */
    0.            0.                446.         wlog("start of icputime", NULL);
                                    447.
    0.            0.                448.         if(k == 0) {
    0.            0.                449.                 k = 80;
                                    450.         }
    0.            0.                451.         for (i = 0; i < k; i ++) {
    0.            0.                452.                 x = 0;
    1.150         1.150             453.                 for(j=0; j<1000000; j++) {
    1.420         1.420             454.                         x = x + 1;
                                    455.                 }
                                    456.         }
                                    457.
                                    458.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
    0.            0.010             459.                 "icputime", NULL);
    0.            0.                460.         return 0;
```

The time spent on the loop line is approximately the same as the time spent on the loop line in `cputime()`, but the line in which x is incremented takes much less time to execute than the corresponding line in `cputime()`.

Now examine the annotated disassembly code for these two functions, to see why this is so.

4. **Click `cputime()` in the Function List display, and click Disassembly.**

   The annotated disassembly for `cputime()` is displayed in a text editor window. Scroll down to the instructions for the line of source code in which x is incremented.

```
                                   421.                      x = x + 1.0;
     0.210        0.210                  [ 421]    13944:  ld        [%fp - 16], %f2
 ## 1.610        1.610                  [ 421]    13948:  fstod     %f2, %f4
     0.           0.                     [ 421]    1394c:  ldd       [%l3], %f2
     0.540        0.540                  [ 421]    13950:  faddd     %f4, %f2, %f2
     0.510        0.510                  [ 421]    13954:  fdtos     %f2, %f2
     0.           0.                     [ 421]    13958:  st        %f2, [%fp - 16]
     0.230        0.230                  [ 420]    1395c:  ld        [%fp - 12], %l0
     0.420        0.420                  [ 420]    13960:  add       %l0, 1, %l1
     0.190        0.190                  [ 420]    13964:  cmp       %l1, %l2
     0.           0.                     [ 420]    13968:  bl        0x13944
     0.           0.                     [ 420]    1396c:  st        %l1, [%fp - 12]
     0.           0.                     [ 418]    13970:  ld        [%fp - 8], %l0
     0.           0.                     [ 418]    13974:  add       %l0, 1, %l1
     0.           0.                     [ 418]    13978:  ld        [%fp + 68], %l0
     0.           0.                     [ 418]    1397c:  cmp       %l1, %l0
     0.           0.                     [ 418]    13980:  bl        0x13914
     0.           0.                     [ 418]    13984:  st        %l1, [%fp - 8]
```

   A significant amount of time is spent executing the `fstod` and `fdtos` instructions. These instructions convert the value of `x` from a single floating-point value to a double floating-point value and back again. This must be done so that `x` can be incremented by 1.0, which is a double floating-point constant.

5. **Click `icputime()` in the Function List display and click Disassembly.**

   The annotated disassembly for `icputime()` is displayed in a text editor window. Scroll down to the instructions for the line of source code in which `x` is incremented.

```
                                   454.                      x = x + 1;
     0.210        0.210                  [ 454]    13a84:  ld        [%fp - 16], %l0
     1.210        1.210                  [ 454]    13a88:  add       %l0, 1, %l0
     0.           0.                     [ 454]    13a8c:  st        %l0, [%fp - 16]
     0.240        0.240                  [ 453]    13a90:  ld        [%fp - 12], %l0
     0.670        0.670                  [ 453]    13a94:  add       %l0, 1, %l1
     0.240        0.240                  [ 453]    13a98:  cmp       %l1, %l2
     0.           0.                     [ 453]    13a9c:  bl        0x13a84
     0.           0.                     [ 453]    13aa0:  st        %l1, [%fp - 12]
     0.           0.                     [ 451]    13aa4:  ld        [%fp - 8], %l0
     0.           0.                     [ 451]    13aa8:  add       %l0, 1, %l1
     0.           0.                     [ 451]    13aac:  ld        [%fp + 68], %l0
     0.           0.                     [ 451]    13ab0:  cmp       %l1, %l0
     0.           0.                     [ 451]    13ab4:  bl        0x13a64
     0.           0.                     [ 451]    13ab8:  st        %l1, [%fp - 8]
```

   All that is involved is a load, add, and store operation that takes approximately a third of the time of the corresponding set of instructions in `cputime()`, because no conversions are necessary. The value 1 does not need to be loaded into a register—it can be added directly to `x` by a single instruction.

Extension Exercise for Simple Metric Analysis

Edit the source code for `synprog`, and change `x` to `double` in `cputime()`. What effect does this have on the time? What differences do you see in the annotated disassembly listing?

The `gprof` Fallacy

This section examines how time is attributed from a function to its callers and compares the way attribution is done by the Performance Analyzer with the way it is done by `gprof`.

1. **Click `gpf_work()` and then click Callers-Callees.**

   The Callers-Callees window is displayed. This window is described in "Examining Callers-Callees Metrics for a Function" on page 92 and also in "Running the Sampling Collector and Performance Analyzer" on page 11 of this chapter.

   The Callers pane shows two functions that call the selected function, `gpf_b()` and `gpf_a()`. The Callees pane is empty because `gpf_work()` does not call any other functions. Such functions are called "leaf functions."

   WorkShop Analyzer – Callers–Callees Window – test.1.er

   Sort Metric: Attr. User CPU

Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
3.700	0.	3.700	gpf_b
0.340	0.	0.340	gpf_a
4.040	4.040	4.040	gpf_work

   Metrics... Close Help

   Examine the attributed user CPU time in the Callers pane. Most of the time in `gpf_work()` results from calls from `gpf_b()`. Much less time results from calls from `gpf_a()`.

   To see why `gpf_b()` calls account for over ten times as much time in `gpf_work()` as calls from `gpf_a()`, you must examine the source code for the two callers.

2. **Click `gpf_a()` in the Callers pane to select it.**

   `gpf_a()` becomes the selected function, and moves to the middle pane; its callers appear in the Callers pane, and `gpf_work()`, its callee, appears in the Callees pane.

3. **In the Function List display, where `gpf_a()` is now the selected function, click the Source button.**

   The annotated source code for `gpf_a()` is displayed in a text editor window.

4. **Scroll down so that you can see the code for both `gpf_a()` and `gpf_b()`.**

```
                                  725. void
                                  726. gpf_a()
                                  727. {
                                  728.         hrtime_t        start;
                                  729.         hrtime_t        vstart;
                                  730.         int     i;
                                  731.
   0.           0.                732.         start = gethrtime();
   0.           0.                733.         vstart = gethrvtime();
                                  734.
   0.           0.                735.         for(i = 0; i < 9; i ++) {
   0.           0.340             736.                 gpf_work(1);
                                  737.         }
                                  738.
                                  739.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
   0.           0.                740.                 "gpf_a -- 9 X gpf_work(1)", NULL);
   0.           0.                741. }
                                  742.
                                  743. void
                                  744. gpf_b()
                                  745. {
                                  746.         hrtime_t        start;
                                  747.         hrtime_t        vstart;
                                  748.
   0.           0.                749.         start = gethrtime();
   0.           0.                750.         vstart = gethrvtime();
                                  751.
   0.           3.700             752.         gpf_work(10);
                                  753.
                                  754.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
   0.           0.                755.                 "gpf_b -- 1 X gpf_work(10)", NULL);
   0.           0.                756. }
```

   `gpf_a()` calls `gpf_work()` ten times with an argument of 1, whereas `gpf_b()` calls `gpf_work()` only once, but with an argument of 10. The arguments from `gpf_a()` and `gpf_b()` are passed to the formal argument `amt` in `gpf_work()`.

   Now examine the code for `gpf_work()`, to see why the way the callers call `gpf_work()` makes a difference.

5. **Scroll down one screen in the text editor, to display the code for `gpf_work()`.**

```
                                  758. void
                                  759. gpf_work(int amt)
                                  760. {
                                  761.         int     i;
                                  762.         int     imax;
                                  763.
   0.           0.                764.         imax = 4* amt * amt;
                                  765.
   0.           0.                766.         for(i = 0; i < imax; i ++) {
                                  767.                 volatile float x;
                                  768.                 int j;
   0.           0.                769.                 x = 0.0;
   0.790        0.790             770.                 for(j=0; j<200000; j++) {
## 3.250        3.250             771.                         x = x + 1.0;
                                  772.                 }
                                  773.         }
   0.           0.                774. }
```

Examine the line in which the variable imax is computed: imax is the upper limit for the following for loop.The time spent in gpf_work() thus depends on the square of the argument amt. So ten times as much time is spent on one call from a function with an argument of 10 (400 iterations) than is spent on ten calls from a function with an argument of 1 (10 instances of 4 iterations).

In gprof, however, the amount of time spent in a function is estimated from the number of times the function is called, regardless of how the time depends on the function's arguments or any other data that it has access to. So for an analysis of synprog, gprof incorrectly attributes ten times as much time to calls from gpf_a() as it does to calls from gpf_b(). This is the gprof fallacy.

The Effects of Recursion

This section demonstrates how the Performance Analyzer assigns metrics to functions in a recursive sequence. In the data collected by the Sampling Collector, each instance of a function call is recorded, but in the analysis, the metrics for all instances of a given function are aggregated. The synprog program contains two examples of recursive calling sequences:

- Function recurse() demonstrates direct recursion. It calls function real_recurse(), which then calls itself until a test condition is met. At that point it performs some work that requires user CPU time The flow of control returns through successive calls to real_recurse() until it reaches recurse().
- Function bounce() demonstrates indirect recursion. It calls function bounce_a(), which checks to see if a test condition is met. If it is not, it calls function bounce_b(). bounce_b() in turn calls bounce_a(). This sequence continues until the test condition in bounce_a() is met. Then bounce_a() performs some work that requires user CPU time, and the flow of control returns through successive calls to bounce_b() and bounce_a() until it reaches bounce().

In either case, exclusive metrics belong only to the function in which the actual work is done, in these cases real_recurse() and bounce_a(). These metrics are passed up as inclusive metrics to every function that calls the final function.

First, examine the metrics for recurse() and real_recurse():

1. **In the Function List display, find function recurse() and click it.**

   Instead of scrolling the function list you can use the search feature:

   a. **Choose View ➤ Find from the menu bar.**

      The Find dialog box is displayed.

   b. **Enter the function name, recurse, in the text box.**

c. **Click Apply until the function `recurse()` is highlighted in the Function List display.**

d. **Click Cancel to close the dialog box.**

Function `recurse()` shows inclusive user CPU time, but its exclusive user CPU time is zero because all `recurse()` does is execute a call to `real_recurse()`.

Note – For some runs, `recurse()` might show a small exclusive CPU time value. Because profiling experiments are statistical in nature, the experiment that you run on `synprog` might record one or two profile events in `recurse()`. However, the exclusive time due to these events is much less than the inclusive time.

---

2. **Click Callers-Callees.**

   The Callers-Callees window shows that `recurse()` calls one function, `real_recurse()`.

3. **Click `real_recurse()` in the callees pane.**

WorkShop Analyzer – Callers–Callees Window – test.1.er

Sort Metric: Attr. User CPU	Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
	1.850	1.850	1.850	real_recurse
	0.	0.	1.850	recurse
	1.850	1.850	1.850	real_recurse

Metrics... Close Help

The Callers-Callees window now displays information for `real_recurse()`:

- Both `recurse()` and `real_recurse()` appear in the callers pane as callers of `real_recurse()`. You would expect this, because after `recurse()` calls `real_recurse()`, `real_recurse()` calls itself recursively.
- In order to simplify the display, `real_recurse()` does not appear in the callee pane as its own callee.
- Exclusive metrics as well as inclusive metrics are displayed in the center function pane for `real_recurse()`, where the actual user CPU time is spent. The exclusive metrics are passed back to `recurse()` as inclusive metrics.

- Exclusive metrics are also displayed for `real_recurse()` in the caller pane. If a function generates exclusive metrics, the Performance Analyzer displays them for that function anywhere it appears in the Callers-Callees window.

Now examine what happens in the indirect recursive sequence.

1. **Find function `bounce()` in the Function List display and click it.**

   Function `bounce()` shows inclusive user CPU time, but its exclusive user CPU time is zero. This is because all `bounce()` does is execute a call to `bounce_a()`.

2. **Click Callers-Callees.**

   The Callers-Callees window shows that `bounce()` calls only one function, `bounce_a()`.

3. **Click `bounce_a()`.**

WorkShop Analyzer – Callers–Callees Window – test.1.er

Sort Metric: Attr. User CPU

Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
0.750	0.	0.750	bounce_b
0.	0.	0.750	bounce
0.750	0.750	0.750	bounce_a
0.	0.	0.750	bounce_b

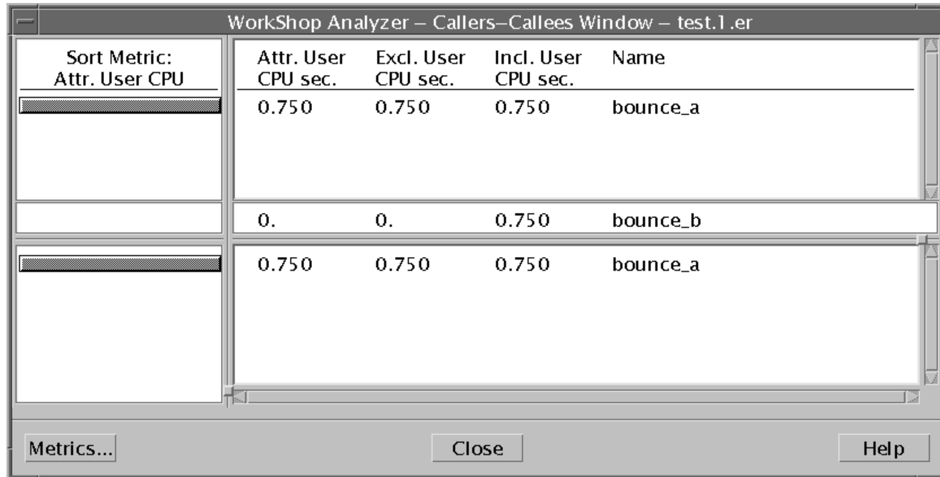
Metrics... Close Help

The Callers-Callees window now displays information for `bounce_a()`:

- Both `bounce()` and `bounce_b()` appear in the callers pane as callers of `bounce_a()`.
- In addition, `bounce_b()` appears in the callee pane. If a function calls an intermediate function instead of calling itself recursively, that intermediate function does appear in the callee pane.
- Exclusive as well as inclusive metrics are displayed for `bounce_a()`, where the actual user CPU time is spent. These are passed up to the functions that call `bounce_a()` as inclusive metrics.

4. **Click** `bounce_b()`.

   The Callers-Callees window now displays information for `bounce_b()`.



`bounce_b()` is shown as both calling and being called by `bounce_a()`. Exclusive as well as inclusive metrics for `bounce_a()` appear in both the caller and the callee panes because if a function generates exclusive metrics, the Performance Analyzer displays the metrics for that function anywhere the function appears in the Callers-Callees window.

Loading Dynamically Linked Shared Objects

This section demonstrates how the Performance Analyzer handles calls to functions that are part of a dynamically linked shared object that may be loaded at different places at different times.

The `synprog` directory contains two dynamically linked shared objects, `so_syn.so` and `so_syx.so`. In the course of execution, `synprog` first loads `so_syn.so` and makes a call to one of its functions, `so_burncpu()`. Then it unloads `so_syn.so`, loads `so_syx.so` at what happens to be the same address, and makes a call to one of the `so_syx.so` functions, `sx_burncpu()`. Then, without unloading `so_syx.so`, it loads `so_syn.so` again—at a different address, because the address where it was first loaded is still being used by another shared object—and makes another call to `so_burncpu()`.

The functions `so_burncpu()` and `sx_burncpu()` perform identical operations, as you can see if you examine their source code. Therefore they should take the same amount of user CPU time to execute.

The addresses at which the shared objects are loaded are determined at run time, and the run-time loader chooses where to load the objects.

This rather artificial exercise demonstrates that the same function can be called at different addresses at different points in the program execution, that different functions can be called at the same address, and that the Performance Analyzer deals correctly with this behavior, aggregating the data for a function regardless of the address at which it appears:

- **Scroll the Function List display so that you can see the metrics for both** `sx_burncpu()` **and** `so_burncpu()`**.**

`so_burncpu()` performs operations identical to those of `sx_burncpu()`. The user CPU time for `so_burncpu()` is almost exactly twice the user CPU time for `sx_burncpu()` because `so_burncpu()` was executed twice. The Performance Analyzer recognized that the same function was executing and aggregated the data for it, even though it appeared at two different addresses in the course of program execution.

Example 2: OpenMP Parallelization Strategies

The Fortran program `omptest` uses OpenMP parallelization and is designed to test the efficiency of parallelization strategies for two different cases:

- The first case compares the use of a `PARALLEL SECTIONS` directive with a `PARALLEL DO` directive for a section of code in which two arrays are updated from another array. This case illustrates the issue of balancing the work load across the threads.
- The second case compares the use of a `CRITICAL SECTION` directive with a `REDUCTION` directive for a section of code in which array elements are summed to give a scalar result. This case illustrates the cost of contention among threads for memory access.

See the *Fortran Programming Guide* for background on parallelization strategies and OpenMP directives. When the compiler identifies an OpenMP directive, it generates special functions and calls to the threads library. These functions appear in the Performance Analyzer display. For more information, see "Parallel Execution and Compiler-Generated Body Functions" on page 132 and "Compiler-Generated Body Functions" on page 140. Messages from the compiler about the actions it has taken appear in the annotated source and disassembly listings.

Collecting Data for `omptest`

Read the instructions in the sections, "Setting Up the Examples for Execution" on page 10 and "Running the Sampling Collector and Performance Analyzer" on page 11, if you have not done so. Compile `omptest` before you begin this example.

In this example you generate two experiments: one that is run with 4 CPUs and one that is run with 2 CPUs.

To collect data for `omptest` and start the Performance Analyzer from the command line, type the following commands in the C shell.

```
% cd ~/work-directory/omptest
% setenv PARALLEL 4
% collect -o omptest.1.er omptest
% setenv PARALLEL 2
% collect -o omptest.2.er omptest
% unsetenv PARALLEL
% analyzer omptest.1.er &
% analyzer omptest.2.er &
```

If you are using the Bourne shell or the Korn shell, type the following commands.

```
$ cd ~/work-directory/omptest
$ PARALLEL=4; export PARALLEL
$ collect -o omptest.1.er omptest
$ PARALLEL=2; export PARALLEL
$ collect -o omptest.2.er omptest
$ unset PARALLEL
$ analyzer omptest.1.er &
$ analyzer omptest.2.er &
```

The `collection` commands are included in the makefile, so in any shell you can type the following commands.

```
$ cd ~/work-directory/omptest
$ make collect
$ analyzer omptest.1.er &
$ analyzer omptest.2.er &
```

To collect data for `omptest` and start the Performance Analyzer using the GUI:

1. **Open the Debugging window or choose Debug ➤ New Program from the Debugging window menu bar and load `omptest`.**

2. **Click Environment Variables in the Debug New Program dialog box, or choose Debug ➤ Edit Run Parameters and click Environment Variables in the Edit Run Parameters dialog box.**

   The Environment Variables dialog box is displayed. Set the `PARALLEL` environment variable to select 4 CPUs for this experiment:

   a. **Type `PARALLEL` in the Name text box and type `4` in the Value text box.**

   b. **Click Add, then click OK.**

3. **Choose Windows ➤ Sampling Collector from the Debugging window menu bar and enter the experiment name `omptest.1.er`.**

   This example uses clock-based metrics and the default sampling interval. No changes in the data options need to be made in the Sampling Collector window.

4. **Click the Start button.**

   The program runs and the Sampling Collector collects data.

5. **When the program is finished, click the Analyzer button.**

   The experiment `omptest.1.er` is loaded into the Analyzer window.

6. **In the Debugging window, choose Debug ➤ Edit Run Parameters and click Environment Variables in the Edit Run Parameters dialog box.**

   Select 2 CPUs for the second experiment:

   a. **Click the `PARALLEL` environment variable in the list box.**

   b. **Change the value to `2` in the Value text box.**

   c. **Click Change, then click OK.**

7. **Choose Windows ➤ Sampling Collector.**

   The experiment name is automatically updated to `omptest.2.er`. No changes are needed in any of the settings.

8. **Click the Start button.**

   The program runs and the Sampling Collector collects data.

9. **When the program is finished, click the Analyzer button.**

   The experiment `omptest.2.er` is loaded into another Analyzer window.

10. **After completing the two experiments, delete the `PARALLEL` environment variable using the Environment Variables dialog box.**

You are now ready to analyze the omptest experiment using the procedures in the following sections.

Comparing Parallel Sections and Parallel Do Strategies

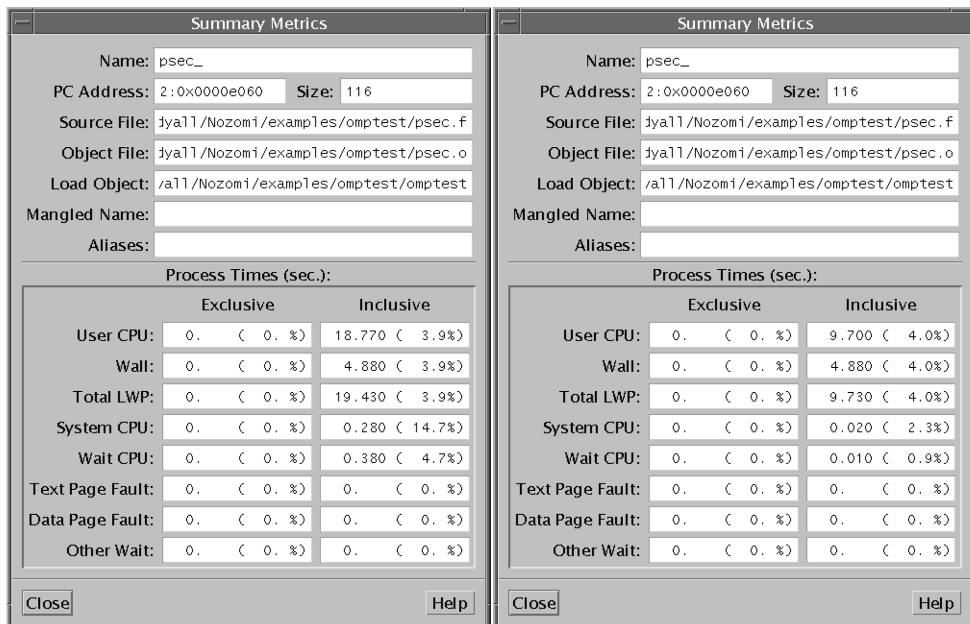
This section compares the performance of two routines, psec_() and pdo_(), that use the PARALLEL SECTIONS directive and the PARALLEL DO directive. The performance of the routines is compared as a function of the number of CPUs.

To compare the four-CPU run with the two-CPU run, you must have two Analyzer windows, with omptest.1.er loaded into one, and omptest.2.er loaded into the other.

1. **In each Analyzer window, click the line containing psec_ in the Function List display, then choose View ➤ Show Summary Metrics.**

   The function is at the end of the list. There are other functions which start with psec_ that have been generated by the compiler.

2. **Position the Summary Metrics windows so that you can compare the metrics displayed in them.**



The data for the four-CPU run is on the left in this figure.

3. **Compare the inclusive metrics for user CPU time, wall clock time, and total LWP time.**

   For the two-CPU run, the ratio of wall clock time to either user CPU time or total LWP is about 1 to 2, which indicates relatively efficient parallelization.

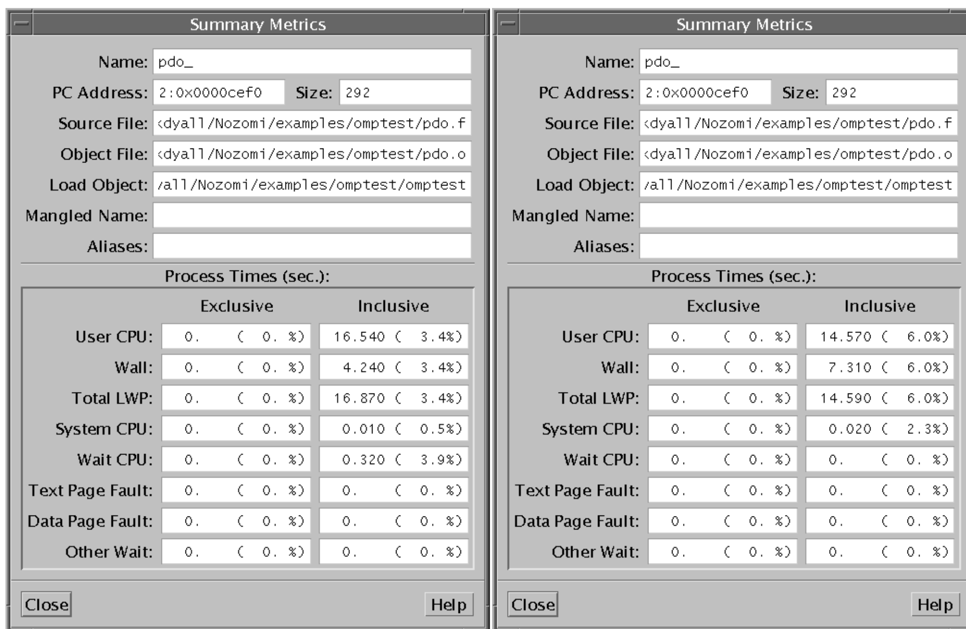
   For the four-CPU run, `psec_()` takes about the same wall clock time as for the two-CPU run, but both the user CPU time and the total LWP time are higher. There are only two sections within the `psec_()` `PARALLEL SECTION` construct, so only two threads are required to execute them, using only two of the four available CPUs at any given time. The other two threads are spending CPU time waiting for work. Because there is no more work available, the time is wasted.

4. **In each Analyzer window, click the line containing `pdo_` in the Function List display.**

   The data for `pdo_()` is now displayed in the Summary Metrics windows.

5. **Compare the inclusive metrics for user CPU time, wall-clock time, and total LWP.**

   The user CPU time for `pdo_()` is about the same as for `psec_()`. The ratio of wall-clock time to user CPU time is about 1 to 2 on the two-CPU run, and about 1 to 4 on the four-CPU run, indicating that the `pdo_()` parallelizing strategy scales much more efficiently on multiple CPUs, taking into account how many CPUs are available and scheduling the loop appropriately.



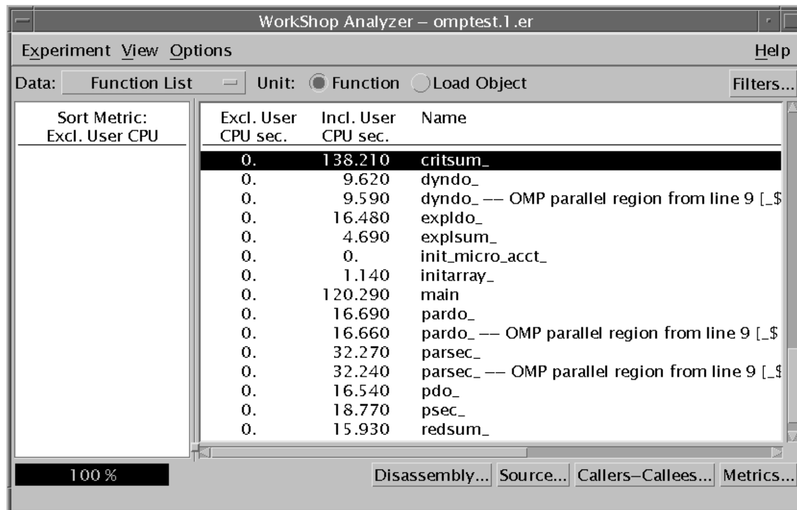
The data for the four-CPU run is on the left in this figure.

Comparing Critical Section and Reduction Strategies

This section compares the performance of two routines, `critsec_()` and `reduc_()`, in which the `CRITICAL SECTIONS` directive and `REDUCTION` directive are used. In this case, the parallelization strategy deals with an identical assignment statement embedded in a pair of `do` loops. Its purpose is to sum the contents of three two-dimensional arrays.

```
t = (a(j,i)+b(j,i)+c(j,i))/k
sum = sum+t
```

1. **For the four-CPU experiment, `omptest.1.er`, locate `critsum_()` and `redsum_()` in the Function List display.**



2. **Compare the inclusive user CPU time for the two functions.**

   The inclusive user CPU time for `critsum_()` is enormous, because `critsum_()` uses a critical section parallelization strategy. Although the summing operation is spread over all four CPUs, only one CPU at a time is allowed to add its value of `t` to `sum`. This is not a very efficient parallelization strategy for this kind of coding construct.

   The inclusive user CPU time for `redsum_()` is much smaller than for `critsum_()`. This is because `redsum_()` uses a reduction strategy, by which the partial sums of `(a(j,i)+b(j,i)+c(j,i))/k` are distributed over multiple processors, after which these intermediate values are added to `sum`. This strategy makes much more efficient use of the available CPUs.

Example 3: Locking Strategies in Multithreaded Programs

The `mttest` program emulates the server in a client-server, where clients queue requests and the server uses multiple threads to service them, using explicit threading. Performance data collected on `mttest` demonstrates the sorts of contentions that arise from various locking strategies, and the effect of caching on execution time.

The executable `mttest` is compiled for explicit multithreading, and it will run as a multithreaded program on a machine with multiple CPUs or with one CPU. There are some interesting differences and similarities in its performance metrics between a multiple CPU system and a single CPU system.

Collecting Data for `mttest`

Read the instructions in the sections, "Setting Up the Examples for Execution" on page 10 and "Running the Sampling Collector and Performance Analyzer" on page 11, if you have not done so. Compile `mttest` before you begin this example.

In this example you generate two experiments: one that is run with 4 CPUs and one that is run with 1 CPU. The experiments record synchronization wait tracing data as well as clock-based data.

To collect data for `mttest` and start the Performance Analyzer from the command line, type the following commands.

```
% cd ~/work-directory/mttest
% collect -s on -o mttest.1.er mttest
% analyzer mttest.1.er &
% collect -s on -o mttest.2.er mttest -u
% analyzer mttest.2.er &
```

The `collect` commands are included in the makefile, so instead you can type the following commands.

```
% cd ~/work-directory/mttest
% make collect
% analyzer mttest.1.er &
% analyzer mttest.2.er &
```

To collect data for `mttest` and start the Performance Analyzer using the GUI:

1. **Open the Debugging window or choose Debug ➤ New Program from the Debugging window menu bar and load** `mttest`**.**
2. **Choose Windows ➤ Sampling Collector from the Debugging window menu bar and enter the experiment name** `mttest.1.er`**.**
3. **Ensure that the Synchronization Wait Tracing check box is checked.**
4. **Click the Start button to collect data.**

   The program runs and the Sampling Collector collects data.
5. **When the program is finished, click the Analyzer button.**

   The experiment `mttest.1.er` is loaded into the Analyzer window.
6. **Choose Debug ➤ Edit Run Parameters from the Debugging window menu bar.**

   Enter the value `-u` in the Arguments box and click OK. This option forces `mttest` to run on a single processor.
7. **Choose Windows ➤ Sampling Collector from the Debugging window menu bar.**

   Ensure that the experiment name is `mttest.2.er` and that the Synchronization Wait Tracing check box is checked.
8. **Click the Start button.**

   The program runs and the Sampling Collector collects data.
9. **When the program is finished, click the Analyzer button.**

   The experiment `mttest.2.er` is loaded into another Analyzer window.

   After you have loaded the two experiments, position the two Analyzer windows so that you can see them both.

   You are now ready to analyze the `mttest` experiment using the procedures in the following sections.

How Locking Strategies Affect Wait Time

1. **In the Function List display for the four-CPU experiment,** `mttest.1.er`**, scroll down to the data for** `lock_local()` **and** `lock_global()`**.**

   Both functions have approximately the same inclusive user CPU time. This indicates that the two functions are doing the same amount of work. However, `lock_global()` spends a lot of time in synchronization waiting, whereas `lock_local()` spends no time in synchronization waiting.

WorkShop Analyzer – mttest.1.er

Experiment View Options Help

Data: Function List Unit: Function Load Object Filters...

Excl. User CPU sec.	Incl. User CPU sec.	Incl. Sync Wait sec.	Incl. Sync Wait Count	Name
0.	4.620	6.993	7	cond_timeout_glol
0.	0.	6.988	3	cond_wait
0.	0.	0.001	30	dump_arrays
0.	0.	0.000	1	fprintf
0.	0.	0.000	1	init_micro_acct
0.	4.660	6.993	3	lock_global
0.	4.610	0.	0	lock_local
0.	4.620	0.	0	lock_none
0.	4.710	35.493	70	locktest
0.	4.710	35.493	73	main
0.	0.	7.027	11	mutex_lock
0.	4.640	0.	0	nothreads
0.	0.	0.001	30	printf
0.	4.610	5.807	4	read_write
0.	0.	0.000	2	resolve_symbols

Sort Metric: Excl. User CPU

100 %

Disassembly... Source... Callers–Callees... Metrics...

The annotated source code for the two functions shows why this is so.

2. **Click the line of data for `lock_global()`, then click the Source button.**

   The annotated source code for `lock_global()` appears in a text editor window.

```
                                        830. void
                                        831. lock_global(Workblk *array, struct scripttab *k)
                                        832. {
                                        833.         /* acquire the global lock */
                                        834.
                                        835. #ifdef SOLARIS
   0.        0.        6.993       3    836.         mutex_lock(&global_lock);
                                        837. #endif
                                        838. #ifdef POSIX
                                        839.         pthread_mutex_lock(&global_lock);
                                        840. #endif
                                        841. #ifdef LWP
                                        842.         _lwp_mutex_lock(&global_lock);
                                        843. #endif
```

`lock_global()` uses a global lock to protect all the data. Because of the global lock, all running threads must contend for access to the data, and only one thread has access to it at a time. The rest of the threads must wait until the working thread releases the lock to access the data. This line of source code is responsible for the synchronization wait time.

3. **Click the line of data for `lock_local()`, and then click the Source button.**

   The annotated source code for `lock_local()` appears in the text editor window.

```
                                       923. void
                                       924. lock_local(Workblk *array, struct scripttab *k)
                                       925. {
                                       926.            /* acquire the local lock */
                                       927. #ifdef SOLARIS
  0.        0.        0.        0      928.            mutex_lock(&(array->lock));
                                       929. #endif
                                       930. #ifdef POSIX
                                       931.            pthread_mutex_lock(&(array->lock));
                                       932. #endif
                                       933. #ifdef LWP
                                       934.            _lwp_mutex_lock(&(array->lock));
                                       935. #endif
```

   `lock_local()` only locks the data in a particular thread's work block. No thread can have access to another thread's work block, so each thread can proceed without contention or time wasted waiting for synchronization. The synchronization wait time for this line of source code, and hence for `lock_local()`, is zero.

   Before you continue to the next instruction, you can dismiss the text editor windows for the annotated source code.

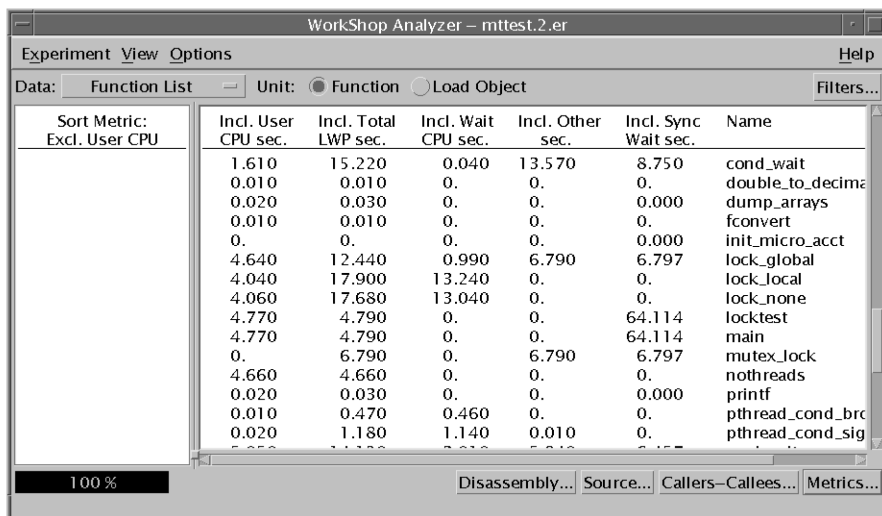
4. **In the Function List display for the one-CPU experiment, `mttest.2.er`, click Metrics.**

   The Metrics dialog box is displayed. Make the following changes:

   a. **Deselect Exclusive User CPU Time and Inclusive Synchronization Wait Counts in the Show Time column.**

   b. **Select Inclusive Total LWP Time, Inclusive Wait CPU Time and Inclusive Other Wait Time in the Show Time column.**

5. **Scroll down to the data for `lock_local()` and `lock_global()`.**



Incl. User CPU sec.	Incl. Total LWP sec.	Incl. Wait CPU sec.	Incl. Other sec.	Incl. Sync Wait sec.	Name
1.610	15.220	0.040	13.570	8.750	cond_wait
0.010	0.010	0.	0.	0.	double_to_decima
0.020	0.030	0.	0.	0.000	dump_arrays
0.010	0.010	0.	0.	0.	fconvert
0.	0.	0.	0.	0.000	init_micro_acct
4.640	12.440	0.990	6.790	6.797	lock_global
4.040	17.900	13.240	0.	0.	lock_local
4.060	17.680	13.040	0.	0.	lock_none
4.770	4.790	0.	0.	64.114	locktest
4.770	4.790	0.	0.	64.114	main
0.	6.790	0.	6.790	6.797	mutex_lock
4.660	4.660	0.	0.	0.	nothreads
0.020	0.030	0.	0.	0.000	printf
0.010	0.470	0.460	0.	0.	pthread_cond_brc
0.020	1.180	1.140	0.010	0.	pthread_cond_sig

As in the four-CPU experiment, both functions have the same inclusive user CPU time, and therefore are doing the same amount of work. The synchronization behavior is also the same as on the four-CPU system: `lock_global()` uses a lot of time in synchronization waiting but `lock_local()` does not.

However, total LWP time for `lock_global()` is actually less than for `lock_local()`. This is because of the way each locking scheme schedules the threads to run on the CPU. The global lock set by `lock_global()` allows each thread to execute in sequence until it has run to completion. The local lock set by `lock_local()` schedules each thread onto the CPU for a fraction of its run and then repeats the process until all the threads have run to completion. In both cases, the threads spend a significant amount of time waiting for work. The threads in `lock_global()` are waiting for the lock. This wait time appears in the Inclusive Synchronization Wait Time metric and also the Other Wait Time metric. The threads in `lock_local()` are waiting for the CPU. This wait time appears in the Wait CPU Time metric.

6. **Click Metrics, then click Default and OK to reload the default metrics for the rest of the example.**

How Data Management Affects Cache Performance

1. **Scroll the Function List display to the data for `ComputeA()` and `ComputeB()` in both Analyzer windows.**

   In the one-CPU experiment, `mttest.2.er`, the inclusive user CPU time for `ComputeA()` is almost the same as for `ComputeB()`.

WorkShop Analyzer – mttest.2.er

Experiment View Options Help

Data: Function List Unit: Function Load Object Filters...

Sort Metric: Excl. User CPU

Excl. User CPU sec.	Incl. User CPU sec.	Incl. Sync Wait sec.	Incl. Sync Wait Count	Name
62.970	62.970	96.388	65	<Total>
5.170	5.170	0.	0	addone
4.660	4.660	0.	0	compute
4.630	4.630	0.	0	computeG
4.610	4.610	0.	0	computeC
4.540	4.540	0.	0	computeH
4.420	4.420	0.	0	computeJ
4.230	4.230	0.	0	computeD
4.230	4.230	0.	0	computeI
4.120	4.120	0.	0	computeB
4.060	4.060	0.	0	computeA
4.040	4.040	0.	0	computeE
3.570	8.740	0.	0	computeF
3.290	3.290	0.	0	mutex_trylock
2.670	62.920	32.274	18	do_work

100 %

Disassembly... Source... Callers–Callees... Metrics...

In the four-CPU experiment, `mttest.1.er`, `ComputeB()` uses much more inclusive user CPU time than `ComputeA()`.

WorkShop Analyzer – mttest.1.er

Experiment View Options Help

Data: Function List Unit: Function Load Object Filters...

Sort Metric: Excl. User CPU

Excl. User CPU sec.	Incl. User CPU sec.	Incl. Sync Wait sec.	Incl. Sync Wait Count	Name
82.000	82.000	63.471	101	<Total>
20.140	20.140	0.	0	computeB
6.480	6.480	0.	0	addone
4.690	4.690	0.	0	computeD
4.660	4.660	0.	0	computeC
4.640	4.640	0.	0	compute
4.620	4.620	0.	0	computeA
4.620	4.620	0.	0	computeG
4.620	4.620	0.	0	computeH
4.610	4.610	0.	0	computeE
4.610	4.610	0.	0	computeJ
4.590	4.590	0.	0	computeI
4.190	4.190	0.	0	mutex_trylock
3.520	10.000	0.	0	computeF
3.300	82.000	27.944	21	do_work

100 %

Disassembly... Source... Callers–Callees... Metrics...

The remaining instructions apply to the four-CPU experiment, `mttest.1.er`.

2. **Click `ComputeA()`, then click the Source button. Scroll down in the text editor so that the source for both `ComputeA()` and `ComputeB()` is displayed.**

```
                                    1344. void
                                    1345. computeA(double *x)
                                    1346. {
                                    1347.          int i,j;
      0.          0.                1348.          *x = 0;
      4.620       4.620             1349.          for (i = 0; i < 20000000; i++) { *x = *x + 1.0; }
      0.          0.                1350. }
                                    1351.
                                    1352. void
                                    1353. computeB(double *x)
                                    1354. {
                                    1355.          int i,j;
      0.          0.                1356.          *x = 0;
   ## 20.140     20.140             1357.          for (i = 0; i < 20000000; i++) { *x = *x + 1.0; }
      0.          0.                1358. }
```

(The annotated source code is shown above with only User CPU time displayed. In your display there will be other metrics listed unless you choose to deselect them.)

The code for these functions is identical: a loop adding one to a variable. All the user CPU time is spent in this loop. To find out why `ComputeB()` uses more time than `ComputeA()`, you must examine the code that calls these two functions.

When you have examined the source code, dismiss the text editor windows.

3. **In the Function List display, click `ComputeA()`, then click the Callers-Callees button.**

The Callers-Callees window is displayed, showing the selected function, `ComputeA()`, in the middle display pane, and its caller in the upper pane.

4. **Click caller `lock_none()`.**

   `lock_none()` is now the selected function in the Function List display as well as the Callers-Callees window.

5. **Click the Source button in the Function List display.**

   The annotated source code for `lock_none()` is displayed in a text editor window.

6. **Scroll down so that the code for `cache_trash()`, which is the function that calls `ComputeB()`, is also visible.**

```
                          793. void
                          794. lock_none(Workblk *array, struct scripttab *k)
                          795. {
   0.        0.           796.         array->ready = array->start;
   0.        0.           797.         array->vready = array->vstart;
                          798.
   0.        0.           799.         array->compute_ready = array->ready;
   0.        0.           800.         array->compute_vready = array->vready;
                          801.
                          802.         /* do some work on the current array */
   0.        4.620        803.         (k->called_func)(&array->list[0]);
                          804.
   0.        0.           805.         array->compute_done = gethrtime();
   0.        0.           806.         array->compute_vdone = gethrvtime();
                          807.
   0.        0.           808. }
                          809.
                          810. /* cache_trash: multiple threads refer to adjacent words,
                          811.  *       causing false sharing of cache lines, and trashing
                          812.  */
                          813. void
                          814. cache_trash(Workblk *array, struct scripttab *k)
                          815. {
   0.        0.           816.         array->ready = array->start;
   0.        0.           817.         array->vready = array->vstart;
                          818.
   0.        0.           819.         array->compute_ready = array->ready;
   0.        0.           820.         array->compute_vready = array->vready;
                          821.
                          822.         /* use a datum that will share a cache line with others */
   0.       20.140        823.         (k->called_func)(&element[array->index]);
                          824.
   0.        0.           825.         array->compute_done = gethrtime();
   0.        0.           826.         array->compute_vdone = gethrvtime();
   0.        0.           827. }
                          828.
```

   (The annotated source code is shown with only the User CPU time displayed. In your display there will be other metrics listed unless you choose to deselect them.)

   Both `ComputeA()` and `ComputeB()` are called by reference using a pointer, so their names do not appear in the source code.

   You can verify that `cache_trash()` is the caller of `ComputeB()` by selecting `ComputeB()` in the Function List display then clicking Callers-Callees.

7. **Compare the calls to `ComputeA()` and `ComputeB()`.**

   `ComputeA()` is called with a double in the thread's work block as an argument (`&array->list[0]`), that can be read and written to directly without danger of contention with other threads.

   `ComputeB()`, however, is called with a series of doubles that occupy successive words in memory (`&element[array->index]`). Whenever a thread writes to one of these addresses in memory, any other threads that have that address in their cache

must delete the data, which is now out-of-date. If one of the threads needs the data again later in the program, the data must be copied back into the data cache from memory, even if the data item that is needed has not changed. The resulting cache misses, which are attempts to access data not available in the data cache, waste a lot of CPU time. This explains why `ComputeB()` uses much more user CPU time than `ComputeA()` in the four-CPU experiment.

In the one-CPU experiment, only one thread is running at a time and no other threads can write to memory. The running thread's cache data never becomes invalid. No cache misses or resulting copies from memory occur, so the performance for `ComputeB()` is just as efficient as the performance for `ComputeA()` when only one CPU is available.

Extension Exercises for `mttest`

1. If you are using a computer that has hardware counters, run the four-CPU experiment again and collect data for one of the cache hardware counters, such as cache misses or stall cycles. You can combine the information from this new experiment with the previous experiment by choosing Experiment ➤ Add in the Analyzer window instead of clicking the Analyzer button in the Sampling Collector window.
2. The makefile contains optional settings for compilation variables that are commented out. Try changing some of these options and see what effect the changes have on program performance. The compilation variables to try are:
   - `THREADS` – Select the threads model.
   - `OFLAGS` – Compiler optimization flags

Example 4: Cache Behavior and Optimization

This example addresses the issue of efficient data access and optimization. It uses two implementations of a matrix-vector multiplication routine, `dgemv`, which is included in standard BLAS libraries. Three copies of the two routines are included in the program. The first copy is compiled without optimization, to illustrate the effect of the order in which elements of an array are accessed on the performance of the routines. The second copy is compiled with `-O3`, and the third with `-fast`, to illustrate the effect of compiler loop reordering and optimization.

This example illustrates the use of hardware counters and compiler commentary for performance analysis.

Collecting Data for cachetest

Read the instructions in the sections, "Setting Up the Examples for Execution" on page 10 and "Running the Sampling Collector and Performance Analyzer" on page 11, if you have not done so. Compile cachetest before you begin this example.

In this example you generate several experiments with data collected from different hardware counters, as well as an experiment that contains clock-based data.

To collect data for cachetest and start the Performance Analyzer from the command line, type the following commands.

```
% cd ~/work-directory/cachetest
% collect -o cachetest.1.er cachetest
% collect -o cachetest.2.er -h fpadd,,fpmul cachetest
% collect -o cachetest.3.er -h cycles,,insts1 cachetest
% collect -o cachetest.4.er -h dcstall cachetest
% analyzer cachetest.1.er cachetest.2.er &
```

The collect commands have been included in the makefile, so instead you can type the following commands.

```
% cd ~/work-directory/cachetest
% make collect
% analyzer cachetest.1.er cachetest.2.er
```

To collect data for cachetest and start the Performance Analyzer using the GUI:

1. **Open the Debugging window or choose Debug ➤ New Program from the Debugging window menu bar and load** cachetest.
2. **Choose Windows ➤ Sampling Collector from the Debugging window menu bar and enter the experiment name** cachetest.1.er.
3. **Click the Start button.**

   The program runs and the Sampling Collector collects clock-based data for the first experiment.
4. **When the program is finished, click the Analyzer button.**

   The experiment cachetest.1.er is loaded into the Analyzer window.
5. **In the Sampling Collector window, click the first hardware counter check box and select or enter in the hardware counter text box the counter name "**fpadd**".**

6. **Click the second hardware counter check box and select or enter in the hardware counter text box the counter name "`fpmul`".**
7. **Click the Start button.**

   The program runs and the Sampling Collector collects hardware counter data for the second experiment.
8. **Select or enter in the hardware counter text boxes the counter names "`cycles`" and "`insts1`".**
9. **Click the Start button.**

   The program runs and the Sampling Collector collects hardware counter data for the third experiment.
10. **Deselect the second hardware counter and select or enter in the first hardware counter text box the counter name "`dcstall`".**
11. **Click the Start button.**

    The program runs and the Sampling Collector collects hardware counter data for the fourth experiment.
12. **In the Analyzer window, choose Experiment ➤ Add and add `cachetest.2.er`.**

    See "Adding Experiments to the Performance Analyzer" on page 84 for details.

    The Analyzer windows that are displayed show data for exclusive metrics only. This is different from the default, and has been set in a local defaults file. See "Defaults Commands" on page 118 for more information.

Setting Up the Display

To limit the information in the display to the routines in the actual program and exclude the system calls:

1. **Choose View ➤ Select Load Objects Included.**

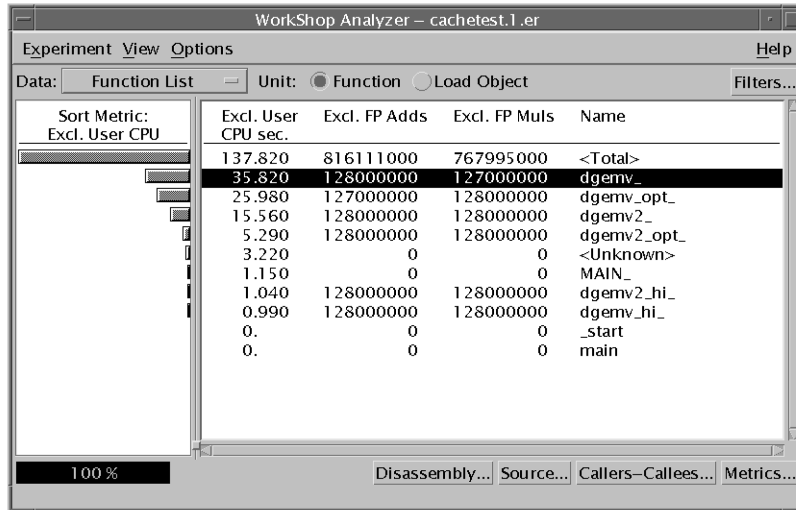
   All of the load objects are currently selected.
2. **Click the Clear All button, select `cachetest` in the load object list, and then click OK.**

   The path name is included in the names of the load objects. When you have finished, only the data from the routines in the main load object is displayed.

   You are now ready to analyze the `cachetest` experiment using the procedures in the following sections.

Execution Speed

1. **For each of the six functions,** `dgemv`**,** `dgemv2`**,** `dgemv_opt`**,** `dgemv2_opt`**,** `dgemv_hi`**, and** `dgemv2_hi`**, add the FP Adds and FP Muls counts and divide by the User CPU time and 10^6.**



The numbers obtained are the MFLOPS counts for each routine. All of the subroutines have the same number of floating-point instructions issued but use different amounts of CPU time. (The variation between the counts is due to counting statistics.) The performance of `dgemv2` is better than that of `dgemv`, the performance of `dgemv2_opt` is better than that of `dgemv_opt`, but the performance of `dgemv2_hi` and `dgemv_hi` are about the same.

2. **Compare the MFLOPS counts obtained here with the MFLOPS values printed by the program.**

The values computed from the data are the same (to within the statistical variation) as the values printed for the first experiment. The printed values for the rest of the experiments are lower because of the overhead for the collection of the hardware counter data. The overhead for the first experiment, which only collects clock-based data, is minimal.

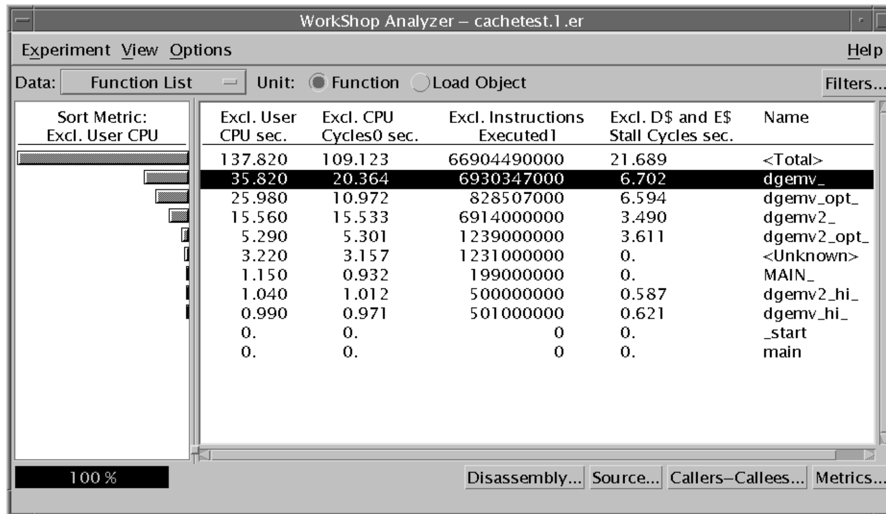
Program Structure and Cache Behavior

In this section, we examine the reasons why `dgemv2` has better performance than `dgemv`.

1. **Choose Experiment ➤ Drop and drop** `cachetest.2.er`**.**

2. **Choose Experiment ➤ Add and add `cachetest.3.er` and `cachetest.4.er`.**

   You must open the Add Experiment dialog box once for each experiment. You might also need to resize the window so that you can see the function names.



3. **Compare the values for User CPU time and CPU Cycles.**

   There is a difference for `dgemv` because of DTLB (data translation lookaside buffer) misses. The system clock is still running while the CPU is waiting for a DTLB miss to be resolved, but the cycle counter is turned off. The difference for `dgemv2` is negligible, indicating that there are few DTLB misses.

4. **Compare the D- and E-cache stall times for `dgemv` and `dgemv2`.**

   There is less time spent waiting for the cache to be reloaded in `dgemv2` than in `dgemv`, because in `dgemv2` the way in which data access occurs makes more efficient use of the cache.

   To see why, we examine the annotated source code. First, to limit the data in the display we remove most of the metrics.

5. **Click Metrics and deselect the metrics for Instructions Executed and CPU Cycles.**

6. **Click `dgemv`, click Source and resize and scroll the edit display so that you can see the source for both `dgemv` and `dgemv2`.**

```
   Excl. User   Excl. D$ and E$
   CPU sec.     Stall Cycles sec.
                                      1. !------------------------------------------------------------
                                      2. ! Standard BLAS interface:      A(1:m) = B(1:m,1:n) * C(1:n)
                                      3. !------------------------------------------------------------
     0.           0.                  4.       SUBROUTINE dgemv (transa, m, n, alpha, b, ldb,     &
                                      5.      &                   c, incc, beta, a, inca)
                                      6.       CHARACTER (KIND=1) :: transa
                                      7.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                      8.       REAL      (KIND=8) :: alpha, beta
                                      9.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                     10.       INTEGER               :: i, j
                                     11.
     0.           0.                 12.       a(1:m) = 0.0
                                     13.
     0.           0.                 14.       DO i = 1, m
     0.           0.004              15.          DO j = 1, n
## 33.940         6.696              16.             a(i) = a(i) + b(i,j) * c(j)
    1.880         0.002              17.          END DO
     0.           0.                 18.       END DO
                                     19.
     0.           0.                 20.       RETURN
     0.           0.                 21.       END
                                     22. !------------------------------------------------------------
     0.           0.                 23.       SUBROUTINE dgemv2 (transa, m, n, alpha, b, ldb,    &
                                     24.      &                    c, incc, beta, a, inca)
                                     25.       CHARACTER (KIND=1) :: transa
                                     26.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                     27.       REAL      (KIND=8) :: alpha, beta
                                     28.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                     29.       INTEGER               :: i, j
                                     30.
     0.           0.                 31.       a(1:m) = 0.0
                                     32.
     0.           0.                 33.       DO j = 1, n         ! <=-----\ swapped loop indices
    0.010         0.001              34.          DO i = 1, m      !    <=--/
   14.080         3.489              35.             a(i) = a(i) + b(i,j) * c(j)
    1.470         0.000              36.          END DO
     0.           0.                 37.       END DO
                                     38.
     0.           0.                 39.       RETURN
     0.           0.                 40.       END
```

The loop structure in the two routines is different. Because the code is not optimized, the data in the array in `dgemv` is accessed by rows, with a large stride (in this case, 4000). This is the cause of the DTLB and cache misses. In `dgemv2`, the data is accessed by column, with a unit stride. Since the data for each loop iteration is contiguous, a large segment can be mapped and loaded into cache and there will be cache misses only when this segment has been used and another is required.

Program Optimization and Performance

In this section we examine the effect of two different optimization options on the program performance, `-O3` and `-fast`. The transformations that have been made on the code are indicated by compiler commentary messages, which appear in the annotated source code.

1. **In the Function List display, click Metrics and select Show Value for Instructions Executed and Show Time for CPU Cycles.**

2. **Compare the metrics for `dgemv_opt` and `dgemv2_opt` with the metrics for `dgemv` and `dgemv2`.**

   The source code is identical to that in `dgemv` and `dgemv2`. The difference is that they have been compiled with the `-O3` compiler option. Both routines show about the same decrease in CPU time, whether measured by User CPU time or by CPU cycles, but in neither routine is the cache behavior improved. There are fewer instructions executed overall, and there are fewer instructions executed in `dgemv_opt` than in `dgemv`, but the performance gain is similar.

3. **Click `dgemv_opt`, click Source and resize and scroll the edit display so that you can see the source for both `dgemv_opt` and `dgemv2_opt`.**

```
    0.                4.        SUBROUTINE dgemv_opt (transa, m, n, alpha, b, ldb,     &
                      5.       &                   c, incc, beta, a, inca)
                      6.        CHARACTER (KIND=1) :: transa
                      7.        INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                      8.        REAL      (KIND=8) :: alpha, beta
                      9.        REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                     10.        INTEGER            :: i, j
                     11.

                  Loop below pipelined with steady-state cycle count = 2 before unrolling
                  Loop below unrolled 3 times
                  Loop below has 0 loads, 2 stores, 0 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs per iteration
    0.               12.        a(1:m) = 0.0
                     13.
    0.               14.        DO i = 1, m

                  Loop below unrolled 4 times
                  Loop below has 4 loads, 0 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
                  Loop below pipelined with steady-state cycle count = 4 before unrolling
    0.               15.           DO j = 1, n
## 25.980            16.              a(i) = a(i) + b(i,j) * c(j)
                     17.           END DO
                     18.        END DO
                     19.
                     20.        RETURN
                     21.        END
                     22. !-----------------------------------------------------------
    0.               23.        SUBROUTINE dgemv2_opt (transa, m, n, alpha, b, ldb,    &
                     24.       &                    c, incc, beta, a, inca)
                     25.        CHARACTER (KIND=1) :: transa
                     26.        INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                     27.        REAL      (KIND=8) :: alpha, beta
                     28.        REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                     29.        INTEGER            :: i, j
                     30.

                  Loop below pipelined with steady-state cycle count = 2 before unrolling
                  Loop below unrolled 3 times
                  Loop below has 0 loads, 2 stores, 0 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs per iteration
    0.               31.        a(1:m) = 0.0
                     32.
    0.               33.        DO j = 1, n       ! <=-----\ swapped loop indices

                  Loop below pipelined with steady-state cycle count = 6 before unrolling
                  Loop below unrolled 3 times
                  Loop below has 4 loads, 2 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
    0.               34.           DO i = 1, m     !     <=--/
    5.290            35.              a(i) = a(i) + b(i,j) * c(j)
                     36.           END DO
                     37.        END DO
                     38.
                     39.        RETURN
                     40.        END
```

   (The annotated source code is shown above with only User CPU time displayed. In your display there will be other metrics listed unless you choose to deselect them.)

The compiler commentary shows that the initialization loop and the inner loop in each routine has been unrolled and pipelined. The inner loop in `dgemv_opt` has the same large stride as in `dgemv`, so the cache behavior is the same.

4. **In the Function List display locate `dgemv_hi` and `dgemv2_hi`.**

   The source code is identical to that in `dgemv` and `dgemv2`. The difference is that they have been compiled with the `-fast` compiler option. Now both routines have the same CPU time and the same cache performance, and a smaller CPU time than `dgemv_opt` and `dgemv2_opt`.

5. **Click `dgemv_hi`, click Source and resize and scroll the edit display so that you can see the source for all of `dgemv_hi`.**

```
   0.                 4.        SUBROUTINE dgemv_hi (transa, m, n, alpha, b, ldb,     &
                      5.       &                   c, incc, beta, a, inca)
                      6.        CHARACTER (KIND=1) :: transa
                      7.        INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                      8.        REAL      (KIND=8) :: alpha, beta
   0.                 9.        REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                     10.        INTEGER              :: i, j
                     11.

                 Loop below has 0 loads, 1 stores, 1 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs per iteration
                 Loop below pipelined with steady-state cycle count = 1 before unrolling
                 Loop below unrolled 8 times
   0.                12.        a(1:m) = 0.0
                     13.

                 Loop below interchanged with loop on line 15
                 Loop below unrolled and jammed
                 Loop below pipelined with steady-state cycle count = 5 before unrolling
                 Loop below unrolled 8 times
                 Loop below has 5 loads, 1 stores, 5 prefetches, 4 FPadds, 4 FPmuls, and 0 FPdivs per iteration
   0.                14.        DO i = 1, m

                 Loop below unrolled and jammed
                 Loop below interchanged with loop on line 14
   0.                15.           DO j = 1, n
## 0.990             16.              a(i) = a(i) + b(i,j) * c(j)
                     17.           END DO
                     18.        END DO
                     19.
                     20.        RETURN
                     21.        END
```

   The compiler has done much more work to optimize this routine. In particular, it has interchanged the loops on lines 14 and 15, created a loop that has 4 floating-point add and 4 floating-point multiply operations per loop cycle, and inserted prefetch instructions to improve the cache behavior.

6. **Scroll down to see the source for `dgemv2_hi`.**

   The compiler commentary messages are the same as for `dgemv_hi` except for the loop interchange. The code generated by the compiler for the two versions of the routine is now essentially the same.

7. **In the Function List display, click Disassembly.**

   Compare the disassembly with that for `dgemv` or `dgemv_opt`. There are many more instructions generated for `dgemv_hi`, but the number of instructions executed is the smallest of the three versions of the routine. Optimization can produce more instructions, but the instructions are used more efficiently and executed less frequently.

CHAPTER 3

Collecting Performance Data

This chapter introduces the Sampling Collector, describes the data it collects, and explains how to use it. There are three ways to collect data: using the Sun WorkShop Debugging window, using the `collector` command within `dbx`, or using the `collect` command from the command line.

This chapter covers the following topics.

- What the Sampling Collector Collects
- Where the Data Is Stored
- Estimating Storage Requirements
- Controlling Data Collection From Your Program
- Compiling Your Program for Data Collection and Analysis
- Limitations on Your Program
- Collecting Data Using the `collect` Command
- Collecting Data From the Sun WorkShop Integrated Programming Environment
- Collecting Data Using the `dbx collector` Subcommands
- Collecting Data From a Running Process
- Collecting Data From MPI Programs

What the Sampling Collector Collects

The Sampling Collector collects data about specific events from a program and the kernel on which the program is running. In addition, the Sampling Collector collects global data and records markers for data grouping.

The data collected from each event is called a profile packet and the process of collecting the data is called profiling. The data collected when a marker is recorded is called a sample packet, and the process of recording them is called taking samples. The event data is converted by the Performance Analyzer into performance metrics.

All profile packets contain the following information:

- A header identifying the data
- A high-resolution timestamp

- A thread ID
- A lightweight process (LWP) ID
- A copy of the call stack

For more information on threads and lightweight processes, see Chapter 6.

In addition to the common data, each profile packet contains information specific to the data type. The three types of data that the Sampling Collector can gather are:

- Clock-based data
- Synchronization wait tracing data
- Hardware-counter overflow data

Clock-Based Data

In clock-based profiling, the state of each LWP is stored at regular time intervals. This time interval is called the profiling interval. The information is stored in an integer array: one element of the array is used for each of the ten microaccounting states maintained by the kernel.The data collected is converted by the Performance Analyzer into times spent in each state, with a resolution of the profiling interval.

The profiling interval must be a multiple of the system clock resolution. The default resolution is 10 milliseconds. If you want to do profiling at higher resolution, you can change the system clock rate to give a resolution of 1 millisecond. If you have root privilege, you can do this by adding the following line to the file `/etc/system`, and then rebooting.

```
set hires_tick=1
```

See the *Solaris Tunable Parameters Reference Manual* for more information.

Synchronization Wait Tracing Data

In multithreaded programs, the synchronization of tasks performed by different threads can cause delays in execution of your program, because one thread might have to wait for access to data that has been locked by another thread, for example. These events are called synchronization delay events and can be recorded by the Sampling Collector. The process of collecting and recording these events is called synchronization wait tracing. The time spent waiting for the lock is called the wait time.

Synchronization delay events are collected by tracing calls to the functions in the threads library, `libthread.so`. In parallel programs that use the Sun Message Passing Interface (MPI) library, MPI blocking calls are another class of synchronization event. Calls to the MPI library are traced in the same way as calls to the threads library.

Events are only recorded if their wait time exceeds a threshold value, which is given in microseconds. A threshold value of 0 means that all synchronization delay events are traced, regardless of wait time. The default threshold is determined by running a calibration test, in which calls are made to the threads library without any synchronization delay. The threshold is the average time for these calls multiplied by an arbitrary factor (currently 6). This procedure prevents the recording of events for which the wait times are due only to the call itself and not to a real delay. As a result, the amount of data is greatly reduced, but the count of synchronization events can be significantly underestimated.

Hardware-Counter Overflow Data

Hardware-counter overflow profiling records a profile packet when a designated hardware counter of the CPU on which the LWP is running overflows. The counter is reset and continues counting. Hardware counters are commonly used to keep track of events like instruction-cache misses, data-cache misses, CPU cycles, floating-point operations, and instructions executed. When the counter reaches the overflow value, the Sampling Collector records a profile packet that includes the overflow value and the counter type.

The UltraSPARC III hardware and the IA hardware have two registers that can be used to count events. The Sampling Collector can collect data from both registers. For each register the Sampling Collector allows you to select the type of counter to monitor for overflow, and to set an overflow value for the counter. Some hardware counters can use either register, others are only available on a particular register. Consequently, not all combinations of hardware counters can be chosen in a single experiment.

Hardware Counter Lists

Hardware counters are system-specific, so the choice of counters available to you depends on the system you are using. For convenience, the performance tools provide aliases for a number of counters that are likely to be in common use. You can obtain a list of available hardware counters from the Sampling Collector. (See "Collecting Data Using the `collect` Command" on page 55, "Collecting Data From the Sun WorkShop Integrated Programming Environment" on page 60 or "Collecting Data Using the `dbx collector` Subcommands" on page 64).

The entries in the counter list for aliased counters are formatted as in the following example.

```
CPU Cycles (cycles = Cycle_cnt/0) 1000003  h=200003
```

The first field, "CPU Cycles", is the name of the corresponding Performance Analyzer metric. The aliased counter name, "cycles", is in parentheses to the left of the "=" sign. The field to the right of the "=" sign, "Cycle_cnt/0", contains the internal name, Cycle_cnt, as it is used by cputrack(1) and the register number on which that counter can be used. The next field is the default overflow value, and the last field is the default high-resolution overflow value.

The aliased counters that are available on both SPARC and IA hardware are given in TABLE 3-1.

TABLE 3-1 Aliased Hardware Counters Available on SPARC and IA Hardware

Standard Counter Name	Metric Name	Description
cycles	CPU Cycles0	CPU cycles (counted on register 0)
cycles1	CPU Cycles1	CPU cycles (counted on register 1)
insts	Instructions Executed0	Instructions executed (counted on register 0)
insts1	Instructions Executed1	Instructions executed (counted on register 1)

The non-aliased entries in the counter list are formatted as in the following example.

```
Cycle_cnt Events (reg. 0) 1000003  h=200003
```

"Cycle_cnt" gives the internal name as used by cputrack(1). The string "Cycle_cnt Events" is the name of the Performance Analyzer metric for this counter. The register on which the event can be counted is given next, in parentheses. The next field is the default overflow value, and the last field is the default high-resolution overflow value.

In the counter list, the aliased counters appear first, then all the counters available on register 0, then all the counters available on register 1. The aliased counters appear twice, with and without the alias. In the non-aliased list, they can have different overflow values.

Limitations on Hardware-Counter Overflow Profiling

There are several limitations on hardware counter overflow profiling:

- You can only collect hardware-counter overflow data on processors that are known to the performance tools, such as UltraSPARC III processors and some IA processors. On other processors, hardware-counter overflow profiling is disabled.
- You cannot collect hardware-counter overflow data with versions of the operating environment that precede the Solaris 8 release.
- You can record data for at most two hardware counters in an experiment. To record data for more than two hardware counters or for counters that use the same register you must run separate experiments.
- You cannot collect hardware-counter overflow data and clock-based data in the same experiment.
- You cannot collect hardware-counter overflow data on a system while `cpustat`(1) is running, because `cpustat` takes control of the counters and does not let a user process use the counters.
- You should not try to use the hardware counters in your own code via the `libcpc`(3) API if you intend to do hardware-counter overflow profiling. If you do, the results of any performance experiment are undefined.

Sample Points

The Sampling Collector has the ability to group the data into samples. Each sample represents a time interval within an experiment. Dividing the data into samples is a convenient way of analyzing different portions of your code or examining how the performance of your code varies over time.

The sample boundary is marked by recording a sample packet, which consists of a header, a timestamp, execution statistics from the kernel and, if requested, address space data. The data recorded at sample points is global to the program and is not converted into performance metrics.

Sample packets are recorded in the following circumstances:

- At a breakpoint set in the Debugging window or in `dbx`
- At the end of a sampling interval, if you have selected periodic sampling
- When you choose Collect ➤ New Sample or click the New Sample button in the Debugging window, if you have selected manual sampling
- At a call to `collector_sample`, if you have put calls to this routine in your code (see "Controlling Data Collection From Your Program" on page 52)
- When a specified signal is delivered, if you have used the `-l` option with the `collect` command (see "Experiment Control Options" on page 58)

- When collection is terminated

The sampling interval is specified as an integer in units of seconds. The default value is 1 second.

Global Information

Global information about your program is recorded with each sample packet. It includes the following kinds of data:

- **Execution statistics.** Include page fault and I/O data, context switches, and a variety of page residency (working-set and paging) statistics. This information appears in the Execution Statistics display of the Performance Analyzer. (See "Examining Execution Statistics" on page 101.)
- **Address-space data (optional).** Consists of page-referenced and page-modified information for every segment of the application's address space. This information appears in the Address Space display of the Performance Analyzer. (See "Examining Address-Space Information" on page 102.)

Where the Data Is Stored

The data collected during one execution of your application is called an experiment. The default name for a new experiment is `test.1.er`. The suffix `.er` is mandatory: if you give a name that does not have it, an error message is displayed and the name is not accepted.

If you choose a name with the format *experiment*`.`*n*`.er`, where *n* is a positive integer, the Sampling Collector automatically increments *n* by one in the names of subsequent experiments—for example, `mytest.1.er` is followed by `mytest.2.er`, `mytest.3.er`, and so on. The Sampling Collector also increments *n* if the experiment already exists, and continues to increment *n* until it finds an experiment name that is not in use. If the experiment name does not contain *n* and the experiment exists, the Sampling Collector inserts `.`*n* before `.er` in the experiment name with *n*=1, and increments *n* if necessary to find a name that is not in use.

Experiments can be collected into groups. The group is defined in an experiment group file, which is stored by default in the current directory. The experiment group file is a plain text file with an experiment name on each line. The default name for an experiment group file is `test.erg`. If the name does not end in `.erg`, an error is displayed and the name is not accepted. Once you have created an experiment group, any experiments you run with that group name are added to the group. Experiment groups can be specified with the `collect` command (see "Collecting

Data Using the `collect` Command" on page 55) or the `dbx collector` subcommands (see "Collecting Data Using the `dbx collector` Subcommands" on page 64).

The default name is different for experiments collected from MPI programs, which create one experiment for each MPI process. The default experiment name is `test.`*m*`.er`, where *m* is the MPI rank of the process. If you specify an experiment group *group*`.erg`, the experiment name will be *group.m*`.er`. If you specify an experiment name, it will override these defaults. See "Collecting Data From MPI Programs" on page 71 for more information.

Each experiment is a data structure that cannot simply be manipulated with Unix commands. Utilities to copy, move and delete experiments have been provided. See "Manipulating Experiments" on page 147 for details.

In addition to the experiment data, the experiment contains archives of the load objects used by the program. These archives contain the addresses, sizes and names of each object file and each function in the load object, as well as the address of the load object and a time stamp for its last modification.

Experiments are stored by default in the current directory. If this directory is on a networked file system, storing the data takes longer than on a local file system, and can distort the performance data. You should always try to record experiments on a local file system if possible.

Estimating Storage Requirements

In this section some guidelines are given for estimating the amount of disk space needed to record an experiment. The profile packets contain data that depends on the type of experiment (clock-based profiling, synchronization wait tracing, hardware counter profiling) and data that depends on the program structure (the call stack). The amount of data that depends on the experiment type is approximately 50 to 100 bytes. The call stack data consists of return addresses for each call, and contains 4 bytes (8 bytes on 64-bit SPARC architecture) per address. Profile packets are recorded for each LWP in the experiment.

For a clock-based profiling experiment with a profiling interval of 10ms and a small call stack, such that the packet size is 100 bytes, data would be recorded at a rate of 10 kbytes/sec per LWP. For a hardware counter overflow profiling experiment collecting data for CPU cycles and instructions executed on a 750MHz processor with an overflow value of 1000000 and a packet size of 100 bytes, data would be recorded at a rate of 150 kbytes/sec per LWP. Applications that have call stacks with

a depth of hundreds of calls could easily record data at ten times these rates. Because of the high data transfer rate, you should try to record experiments to a local file system rather than a networked file system.

To ensure that you can store the experiment and that the recording of the experiment does not significantly distort the performance of your program because of the volume of data stored, you should make a careful choice of the parameters that control the rate of data collection, such as the profiling interval, the synchronization wait tracing threshold and the hardware counter overflow values.

Your estimate of the size of the experiment should be based on the number of LWPs used and the expected execution time with profiling enabled. It should also take into account the disk space used by the archive files (see the previous section). If you are not sure how much space you need, try running your experiment for a short time. From this experiment you can obtain the size of the archive files, which are independent of the data collection time, and scale the size of the profile files to obtain an estimate of the size for the full-length experiment.

As well as allocating disk space, the Sampling Collector allocates buffers in memory to store the profile data before writing it to disk. There is currently no way to specify the size of these buffers. If the Sampling Collector runs out of memory, you should try to reduce the amount of data collected.

Controlling Data Collection From Your Program

The `libcollector.so` dynamic shared library contains some API routines that you can use in your program to control data collection. The routines are written in C and their definitions are as follows.

```
void collector_sample(char *name);
void collector_pause(void);
void collector_resume(void);
void collector_terminate_expt(void);
```

To access the API routines from C or C++, include the following statement.

```
#include "libcollector.h"
```

There is currently no explicit Fortran interface. If you want to use the interface directly from a Fortran program, insert the following directive at the top of each subprogram that calls the API routines.

```
!$PRAGMA C(function-list)
```

In this directive, *function-list* is a comma-separated list of the names of the API routines you want to use in that subprogram. When you link your program, link with `-lcollector`. See the *Fortran User's Guide* for information on compiler directives and the *Fortran Programming Guide* for information on the C-Fortran interface.

The C include file contains macros that bypass the calls to the real API routines if data is not being collected. In this case the routines are not dynamically loaded. In Fortran the API routines are dynamically loaded when they are called, and there is an overhead associated with the dynamic loading. If you want to avoid this overhead when you are not collecting performance data, you can use conditional compilation for the calls to the API routines and the C pragma directive.

To collect performance data you must run your program using the Sampling Collector, as described later in this chapter. Inserting calls to the API routines does not enable data collection.

The descriptions of the four API routines follow.

`collector_sample(char *name)`

Record a sample packet and label the sample with the given name. The name is not currently used by the Performance Analyzer. If you want to use `collector_sample` from Fortran, you should pass an integer 0 as an argument, which will be interpreted by the C routine as a null pointer.

`collector_pause()`

Turn off the actual writing of data to the experiment. The experiment remains open.

`collector_resume()`

Turn the actual writing of data to the experiment back on.

`collector_terminate_expt()`

Terminate the experiment whose data is being collected. No further data is collected, but the program continues to run normally.

Compiling Your Program for Data Collection and Analysis

You can collect and analyze data for a program compiled with any options, but you will not have access to all the features of the Performance Analyzer unless you use the `-g` compiler option (`-g0` for C++ to ensure that front-end inlining is enabled). When this option is used the compiler generates symbol tables that are used by the Performance Analyzer to obtain source line numbers and file names and print compiler commentary messages. Without this option you cannot view annotated source code listings or compiler commentary, and you might not have all function names in the main Performance Analyzer display.

When you compile your program, you must not disable dynamic linking, which is done with the `-dn` and `-Bstatic` compiler options. If you try to collect data for a program that is entirely statically linked, the Sampling Collector prints an error message and does not collect data. This is because the collector library, among others, is dynamically loaded when you run the Sampling Collector.

If you compile your program with optimization turned on at some level, the compiler can rearrange the order of execution so that it does not strictly follow the sequence of lines in your program. The Performance Analyzer can analyze experiments collected on optimized code, but the data it presents at the disassembly level is often difficult to relate to the original source code lines.

If you compile a C program on an IA platform with an optimization level of 4 or 5, the Sampling Collector is unable to reliably unwind the call stack. As a consequence, only the exclusive metrics for a function are reliable. If you compile a C++ program on an IA platform, you can use any optimization level, as long as you do not use the `-noex` (or `-features=no@except`) compiler option to disable C++ exceptions. If you do use this option the Sampling Collector is unable to reliably unwind the call stack, and only the exclusive metrics for a function are reliable.

Limitations on Your Program

If you want to collect performance data on your program, there are some restrictions on which system utilities you can use. The following is a list of situations in which the Sampling Collector can fail.

- Programs that use `SIGPROF`, or mask it, or install a handler for it can fail for either clock-based profiling or hardware-counter overflow profiling.
- Programs that use `SIGEMT`, or mask it, or install a handler for it can fail for hardware-counter overflow profiling.
- Programs that use entry points in `libcpc`(3) can fail for hardware-counter overflow profiling.
- Programs that use `setitimer`(2) for either `ITIMER_PROF` or `ITIMER_REALPROF` can fail for either clock-based profiling or hardware-counter overflow profiling.

Collecting Data Using the `collect` Command

To run the Sampling Collector from the command line using the `collect` command, type the following.

```
% collect collect-options program program-arguments
```

In this example *collect-options* are the `collect` command options, *program* is the name of the program you want to collect data on, and *program-arguments* are its arguments.

Note – If static linking is enforced when your program is compiled, the `collect` command fails and displays an error message.

If no command arguments are given, the default is to turn on clock-based profiling with a profiling interval of 10 milliseconds.

To obtain a list of options and a list of the names of any hardware counters that are available for profiling, type the `collect` command with no arguments.

```
% collect
```

For a description of the list of hardware counters, see "Hardware-Counter Overflow Data" on page 47. See also "Limitations on Hardware-Counter Overflow Profiling" on page 49.

Periodic sampling is not available with the `collect` command. Sample points are only recorded at the start and the end of the process, unless the program contains sampling calls using the `libcollector`(3) API (see "Controlling Data Collection From Your Program" on page 52), or the `-l` option is set (see "Experiment Control Options" on page 58).

Data Collection Options

These options control the types of data that are collected. See "What the Sampling Collector Collects" on page 45 for a description of the data types.

If no data collection options are given, the default is `-p on`, which enables clock-based profiling with the default profiling interval of 10 milliseconds. The default is turned off by the `-h` option but not by any of the other data collection options.

If clock-based profiling is explicitly disabled, and neither synchronization wait tracing nor hardware counter overflow profiling is enabled, the `collect` command prints an error message and exits.

`-a`

Collect address-space data. It is collected at sample points only (see "Sample Points" on page 49).

Note – This feature is not available on IA hardware.

`-h` *counter*`[,`*threshold*`[,`*counter2*`[,`*threshold2*`]]]`

Collect hardware counter overflow profiling data. The counter names *counter* and *counter2* can be one of the following:

- A standard counter name

- An internal name, as used by `cputrack`(1). If the counter can use either event register, the event register to be used can be specified by appending /0 or /1 to the internal name.

If two counters are specified, they must use different registers. If they do not use different registers, the `collect` command prints an error message and exits.

To obtain a list of available counters, type `collect` with no arguments. A description of the counter list is given in the section "Hardware Counter Lists" on page 47.

The overflow values can be specified using *threshold* and *threshold2*. They can take the following values:

- `h` – The high-resolution value for the chosen counter is used.
- *number* – The overflow value. Must be a positive integer.
- 0 or a null string – The default overflow value is used.

The default is the normal threshold, which is predefined for each counter and which appears in the counter list. See also "Limitations on Hardware-Counter Overflow Profiling" on page 49.

The `-h` option and the `-p` option are incompatible. The `collect` command prints an error message and exits if both are turned on. Using the `-h` option turns off the default `-p on` option.

`-n`

Run the target without using the Sampling Collector. All other options are ignored. Useful to simplify scripts for conditional data collection: for example, for collecting data from some but not all MPI processes in a job.

`-p` *option*

Collect clock-based profiling data. The allowed values of *option* are:

- `off` – Turn off clock-based profiling.
- `on` – Turn on clock-based profiling with the default profiling interval of 10 milliseconds.
- *value* – Set the profiling interval to *value*, given in milliseconds. The value should be a multiple of the system clock resolution. If it is larger but not a multiple it is rounded down. If it is smaller, a warning message is printed and it is set to the system clock resolution. See "Clock-Based Data" on page 46 for information on enabling high-resolution profiling.

Collecting clock-based profiling data is the default action of the `collect` command.

The `-h` option and the `-p` option are incompatible. The `collect` command fails with an error message if both are turned on.

`-s` *option*

Collect synchronization wait tracing data. The allowed values of *option* are:

- `all` – Turn on synchronization wait tracing with a zero threshold. This option will force all synchronization events to be recorded.
- `calibrate` – Turn on synchronization wait tracing and set the threshold value by calibration at runtime. (Equivalent to `on`.)
- `off` – Turn off synchronization wait tracing.
- `on` – Turn on synchronization wait tracing with the default threshold, which is to set the value by calibration at runtime. (Equivalent to `calibrate`.)
- *value* – Set the threshold to *value*, given as a positive integer in microseconds.

Experiment Control Options

`-l` *signal*

Record a sample packet when the signal named *signal* is delivered to the process.

The signal can be specified by the full signal name, by the signal name without the initial letters `SIG`, or by the signal number. Do not use a signal that is used by the program or that would terminate execution. Suggested signals are `SIGUSR1` and `SIGUSR2`. Signals can be delivered to a process by the `kill`(1) command.

If you use both the `-l` and the `-y` options, you must use different signals for each option.

See the `signal`(3HEAD) man page for more information about signals.

`-x`

Leave the target process stopped on exit from the exec system call in order to allow a debugger to attach to it. If you attach `dbx` to the process, the `dbx` command `ignore PROF` will ensure that collection signals are passed on to the `collect` command.

-y *signal*[,r]

Control recording of data with the signal named *signal*. Whenever the signal is delivered to the process, it switches between the paused state, in which no data is recorded, and the recording state, in which data is recorded. Sample points are always recorded, regardless of the state of the switch.

The signal can be specified by the full signal name, by the signal name without the initial letters `SIG`, or by the signal number. Do not use a signal that is used by the program or that would terminate execution. Suggested signals are `SIGUSR1` and `SIGUSR2`. Signals can be delivered to a process by the `kill`(1) command.

If you use both the `-l` and the `-y` options, you must use different signals for each option.

When the `-y` option is used, the Sampling Collector is started in the recording state if the optional `r` argument is given, otherwise it is started in the paused state. If the `-y` option is not used, the Sampling Collector is started in the recording state.

See the `signal`(3HEAD) man page for more information about signals.

Output Options

-d *directory-name*

Place the experiment in directory *directory-name*. This option only applies to individual experiments and not to experiment groups. If the directory does not exist, the `collect` command prints an error message and exits.

-g *group-name*

Make the experiment part of experiment group *group-name*. If *group-name* does not end in `.erg`, the `collect` command prints an error message and exits. If the group exists, the experiment is added to it. The experiment group is placed in the current directory unless *group-name* includes a path.

-o *experiment-name*

Use *experiment-name* as the name of the experiment to be recorded. If *experiment-name* does not end in `.er`, the `collect` command prints an error message and exits. See "Where the Data Is Stored" on page 50 for more information on experiment names and how the Sampling Collector handles them.

Other Options

-V

Print the current version of the `collect` command. No further arguments are examined, and no further processing is done.

-v

Print the current version of the `collect` command and detailed information about the experiment being run.

Collecting Data From the Sun WorkShop Integrated Programming Environment

---

Note – You can access information on how to use the Sampling Collector window in the online help. Click Help in any window to see the online help for that window.

To collect data from the Sun WorkShop Integrated Programming Environment:

1. **Load your program into the Debugging window.**
2. **Ensure that runtime checking is turned off (the default).**

   Icons appear on the Sessions tab or in the status area of the Debugging window that indicate whether memory use or access checking is turned on. Alternatively, if the Check menu shows Enable rather than Disable items, run time checking is off. If you try to enable run-time checking when you have the Sampling Collector window open, an error message is displayed.
3. **From the Debugging window menu bar, choose Windows ➤ Sampling Collector.**
4. **Use the Collect Data radio buttons to enable or disable data collection.**
   - If you select "On", the Sampling Collector is enabled. It remains active after your program has finished running, and for each subsequent run it creates a new experiment. This button is selected by default when you open the window.
   - If you select "Off", the Sampling Collector is disabled and collects and stores no data until you select "On" or reopen the Sampling Collector window.

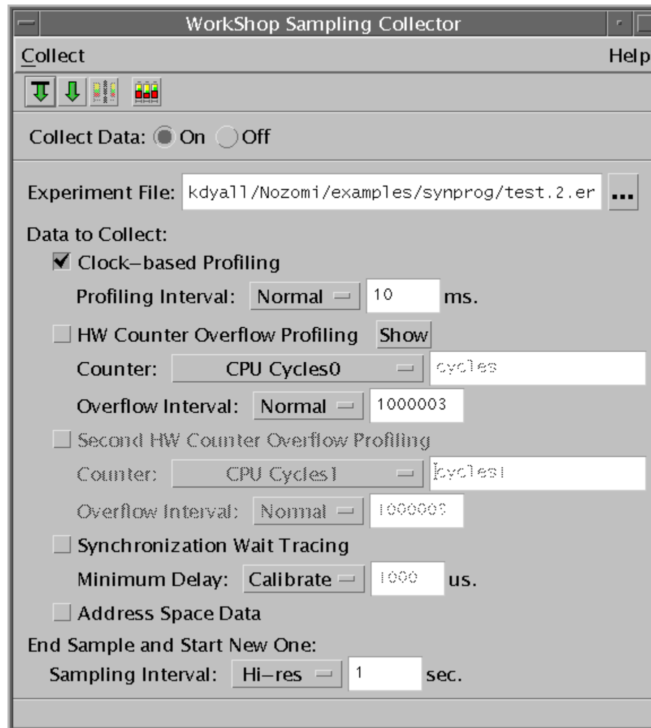


FIGURE 3-1 The Sampling Collector Window

5. **Use the Experiment File text box and browse button to name the experiment.**

   The default experiment name provided by the Sampling Collector is `test.1.er`. If you want to change it, enter the name in the text box or click the browse button to display a file selection window from which you can navigate to a directory and choose a name. The name must be unique, and it must end in `.er`.

   If you use the format *name*`.`*n*`.er` for your experiment name, the Sampling Collector automatically increments the integer *n* in the names of subsequent experiments by one. For example, `test.1.er` is followed by `test.2.er`. If you enter an experiment name that does not end in `.er`, a warning message is displayed, and the last valid name is redisplayed in the text box. See "Where the Data Is Stored" on page 50 for more information on experiment names and how the Sampling Collector handles them.

6. **To collect clock-based profiling data, ensure that the Clock-Based Profiling check box is selected. (This check box is selected by default.)**

   To set the profiling interval, choose one item from the Profiling Interval list box:

   - Normal – 10 millisecond profiling interval.
   - Hi-res – 1 millisecond profiling interval.
   - Custom – Set your own interval in milliseconds.

The profiling interval should be a multiple of the system clock resolution (see "Clock-Based Data" on page 46 for more information). If it is larger but not a multiple it is rounded down. If it is smaller, a warning message is printed and it is set to the system clock resolution. See "Clock-Based Data" on page 46 for information on enabling high-resolution profiling.

7. **To collect information about hardware-counter overflows, click the HW Counter Overflow Profiling check boxes, choose the counters and overflow values.**

   To choose the first hardware counter, do one of the following:

   - Choose one of the common hardware counters from the Counter list box.
   - Choose Other from the Counter list box and type the name of the counter in the Counter text box. To list the available counters, click Show. The list will appear in the Debugging window output pane. See "Hardware-Counter Overflow Data" on page 47 for a description of the list format.

Note – Hardware counters are platform dependent, so the list of available counters differs from system to system. On systems that do not support hardware-counter overflow profiling, this option is disabled.

   To set the overflow value, choose one of these items from the Overflow Interval options menu:

   - Normal – Choose the normal value.
   - Hi-res – Choose the high resolution value.
   - Custom – Set your own value in the text box.

   Both the Normal and Hi-res values depend on the hardware counter. You can obtain these values from the hardware-counter list (see "Hardware-Counter Overflow Data" on page 47).

   When the first hardware counter has been selected, the data entry functions for the second hardware counter are activated. To choose a second hardware counter, follow the same procedure as for the first hardware counter.

   Most hardware counters count on only one register. If you choose a second counter, you must ensure that the second counter counts on a different register from the first. If you do not, an error message is displayed. The register number is listed in the hardware counter list.

8. **To collect synchronization wait tracing data, click the Synchronization Wait Tracing check box.**

   To specify the threshold beyond which tracing begins, choose one of these items from the Minimum Delay options menu:

   - Calibrate – The threshold is determined at run time. See "Synchronization Wait Tracing Data" on page 46 for details.
   - 1000 microseconds.

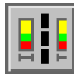
- 100 microseconds.
- All – Sets the threshold to zero so that all synchronization waits are traced.
- Custom – Set your own threshold in microseconds in the text box.

9. **To collect information about memory allocation in the address space, click the Address Space Data check box.**

Note – This feature is not available on IA hardware.

---

10. **To set the sampling interval, choose an item from the Sampling Interval options menu:**
    - Normal – 10-second interval.
    - Hi-res – 1-second interval.
    - Custom – Set your own interval in seconds.
    - Manual – Record a sampling point by either choosing Collect ➤ New Sample in the Sampling Collector window, or clicking the New Sample button.



Note – You must use manual sampling if you are using the asynchronous IO library, `libaio.so`.

---

11. **To run your program and collect data, do one of the following:**
    - Choose Collect ➤ Start from the Sampling Collector window.
    - If you want to start from the beginning of the program, click the Start button in the tool bar of either the Sampling Collector window or the Debugging window.

- If you want to continue execution of a paused program, click the Continue button in the tool bar of either the Sampling Collector window or the Debugging window,

Collecting Data Using the dbx collector Subcommands

To run the Sampling Collector from dbx:

1. **Load your program into dbx by typing the following command.**

```
% dbx program
```

2. **Use the collector command to enable data collection, select the data types, and set any optional parameters.**

```
(dbx) collector subcommand
```

To get a listing of available collector subcommands, type:

```
(dbx) help collector
```

You must use one collector command for each subcommand.

3. **Set up any dbx options you wish to use and run the program.**

If a subcommand is incorrectly given, a warning message is printed and the subcommand is ignored. A complete listing of the collector subcommands follows.

Data Collection Subcommands

The following subcommands control the types of data that are collected by the Sampling Collector. They are ignored with a warning if an experiment is active.

address_space { on | off }

Enables or disables collection of address-space data (pages that have been referenced and modified). The default is off.

Note – This feature is not available on IA hardware.

`hwprofile` *option*

Controls the collection of hardware-counter overflow profiling data. If you attempt to enable hardware-counter overflow profiling on systems that do not support it, `dbx` returns a warning message and the command is ignored. The allowed values for *option* are:

- `on` – Turns on hardware-counter overflow profiling. The default action is to collect data for the `cycles` counter at the normal overflow value.
- `off` – Turns off hardware-counter overflow profiling.
- `list` – Returns a list of available counters See "Hardware-Counter Overflow Data" on page 47 for a description of the list. If your system does not support hardware-counter overflow profiling, dbx returns a warning message.
- `counter` *name value* `[` *name2 value2* `]` – Selects the hardware counter *name*, and sets its overflow value to *value*; optionally selects a second hardware counter *name2* and sets its overflow value to *value2*. An overflow value of 0 is interpreted as the default overflow value. The two counters must use different registers. If they do not, a warning message is printed and the command is ignored.

The default value of *option* is `off`.

If clock-based profiling is already enabled, the `hwprofile` subcommand prints a warning message and turns off clock-based profiling.

See also "Limitations on Hardware-Counter Overflow Profiling" on page 49.

`profile` *option*

Controls the collection of clock-based profiling data. The allowed values for *option* are:

- `on` – Turns on clock-based profiling. If an experiment is active, this option is ignored and a warning is displayed.
- `off` – Turns off clock-based profiling.
- `timer` *value* – Sets the profiling interval to *value* milliseconds. The default setting is 10 ms. The value should be a multiple of the system clock resolution. If the value is larger than the system clock resolution but not a multiple it is rounded down. If the value is smaller than the system clock resolution it is set to the system clock resolution. In both cases a warning message is printed. See "Clock-Based Data" on page 46 to find out how to enable high-resolution profiling.

The default value of *option* is `on`.

If hardware-counter overflow profiling has been turned on using the `hwprofile` subcommand, the `profile` subcommand prints a warning message and turns hardware-counter overflow profiling off.

`sample` *option*

Controls the sampling mode. The allowed values for *option* are:

- `periodic` – Select periodic sampling. Must be used in conjunction with the `period` option.
- `manual` – Select manual sampling.
- `period` *value* – Sets the sampling interval to *value*, given in seconds.

The default value of *option* is `periodic`, with a sampling interval *value* of 1 second.

Note – You must use manual sampling if you are using the asynchronous IO library, `libaio.so`.

`synctrace` *option*

Controls the collection of synchronization wait tracing data. The allowed values for *option* are

- `on` – Turns on synchronization wait tracing.
- `off` – Turns off synchronization wait tracing.
- `threshold` *value* – Sets the threshold for the minimum synchronization delay. The allowed values for *value* are `calibrate`, to use a calibrated threshold determined at runtime, or a value given in microseconds. Setting *value* to `0` (zero) causes the Sampling Collector to trace all events, regardless of wait time. The default setting is `calibrate`.

The default value of *option* is `off`.

Experiment Control Subcommands

`disable`

Disables data collection. If a process is running and collecting data, it terminates the experiment and disables data collection. If a process is running and data collection is disabled, it is ignored with a warning. If no process is running, it disables data collection for subsequent runs.

`enable`

Enables data collection. If a process is running but data collection is disabled, it enables data collection and starts a new experiment. If a process is running and data collection is disabled, it is ignored with a warning. If no process is running, it enables data collection for subsequent runs.

You can enable and disable data collection as many times as you like during the execution of any process. Each time you enable data collection, a new experiment is created.

`pause`

Suspends the collection of data, but leaves the experiment open. Sample points are still recorded. This subcommand is ignored if data collection is already paused.

`resume`

Resumes data collection after a `pause` has been issued. This subcommand is ignored if data collection is active.

Output Subcommand

`store` *option*

Governs where the experiment is stored. This command is ignored with a warning if an experiment is active. The allowed values for *option* are:

- `directory` *directory-name* – Sets the directory where the experiment is stored. This subcommand is ignored with a warning if the directory does not exist.

- `filename` *experiment-name* – Sets the name of the experiment. If the experiment name does not end in `.er`, the subcommand is ignored with a warning. See "Where the Data Is Stored" on page 50 for more information on experiment names and how the Sampling Collector handles them.
- `group` *group-name* – Sets the name of the experiment group. If the group name does not end in `.erg`, the subcommand is ignored with a warning. If the group already exists, the experiment is added to the group.

Information Subcommands

`show`

Shows the current setting of every Sampling Collector control.

`status`

Reports on the status of any open experiment.

Obsolete Subcommands

`close`

Synonym for `disable`.

`enable_once`

Formerly used to enable data collection for one run only. This command is ignored with a warning.

`quit`

Synonym for `disable`.

Collecting Data From a Running Process

The Sampling Collector allows you to collect data from a running process. If the process is already under the control of dbx (either in the command line version or in the Debugging window), you can pause the program and enable data collection using the methods described in previous sections.

If the process is not under the control of dbx, you can attach dbx to it, collect performance data, and then detach from the process, leaving it to continue.

To collect data from a running process that is not under the control of dbx:

1. **Determine the program's process ID (PID).**

   If you started the program from the command line and put it in the background, its PID will be printed to standard output by the shell. Otherwise you can determine the program's PID by typing the following.

```
% ps -ef | grep program-name
```

2. **Attach to the process.**

   - From the Debugging window, choose Debug ➤ Attach Process and select the process using the dialog box. Use the online help for instructions.
   - From dbx, type the following.

```
(dbx) attach program-name pid
```

   If dbx is not already running, type the following.

```
% dbx program-name pid
```

   See the debugger manual, *Debugging a Program With* dbx, for more details on attaching to a process. Attaching to a running process will pause the process.

3. **Start data collection.**

   - From the Debugging window, choose Windows ➤ Sampling Collector and use the dialog box to set up the sampling parameters, then choose Execute ➤ Continue or use the continue button.
   - From dbx, use the collector command to set up the sampling parameters and the cont command to resume the process.

4. **Detach from the process.**

   When you have finished collecting data, pause the program and then detach the process from dbx.

   - From the Debugging window choose Execute ➤ Detach Process.
   - From dbx, type the following.

   ```
   (dbx) detach
   ```

   If you want to collect synchronization wait tracing data, you must preload the Sampling Collector library, libcollector.so, before you run your program, because the library provides wrappers to the real synchronization routines that enable data collection to take place.

   To preload libcollector.so, you must set both the name of the library and the path to the library using environment variables. Use the environment variable LD_PRELOAD to set the name of the library. Use the environment variable LD_LIBRARY_PATH to set the path to the library. If you are using SPARC-V9 64-bit architecture, you must also set the environment variable LD_LIBRARY_PATH64. If you have already defined these environment variables, add the new values to them. The values of the environment variables are shown in TABLE 3-2.

TABLE 3-2 Environment Variable Settings for Preloading the Library libcollector.so

Environment variable	Value
LD_PRELOAD	libcollector.so
LD_LIBRARY_PATH	/opt/SUNWspro/lib
LD_LIBRARY_PATH64	/opt/SUNWspro/lib/v9

If your Sun WorkShop software is not installed in /opt/SUNWspro, ask your system administrator for the correct path. You can set the full path in LD_PRELOAD, but doing this can create complications when using SPARC-V9 64-bit architecture.

Note – Remove the LD_PRELOAD and LD_LIBRARY_PATH settings after the run, so they do not remain in effect for other programs that are started from the same shell.

If you want to collect data from an MPI program that is already running, you must attach a separate instance of dbx to each process and enable the Sampling Collector for each process. When you attach dbx to the processes in an MPI job, each process will be halted and restarted at a different time. The time difference could change the interaction between the MPI processes and affect the performance data you collect. To minimize this problem, one solution is to use pstop(1) to halt all the processes.

However, once you attach dbx to the processes, you must restart them from dbx, and there will be a timing delay in restarting the processes, which can affect the synchronization of the MPI processes. See also “Collecting Data From MPI Programs” on page 71.

Collecting Data From MPI Programs

The Sampling Collector can collect performance data from multi-process programs that use the Sun Message Passing Interface (MPI) library. The MPI library is included in Sun HPC ClusterTools™. Use ClusterTools 3.1 or a compatible version. To launch the parallel jobs, use the Sun™ Cluster Runtime Environment (CRE) command mprun. See the Sun High-Performance Computing (HPC) ClusterTools 3.1 documentation in the Sun HPC 3.1 AnswerBook™ Collection for more information. For information about MPI and the MPI standard, see the MPI web site http://www.mcs.anl.gov/mpi.

Because of the way MPI and the Sampling Collector are implemented, each MPI process records a separate experiment. Each experiment must have a unique name. Where and how the experiment is stored will depend on the kinds of file systems that are available to your MPI job. Issues about storing experiments are discussed in the next section.

To collect data from MPI jobs, you can either run the collect command under MPI or start dbx under MPI and use the dbx collector subcommands. Each of these options is discussed in subsequent sections.

Storing MPI Experiments

Because multiprocessing environments can be complex, there are some issues about storing MPI experiments you should be aware of when you collect performance data from MPI programs. These issues concern the efficiency of data collection and storage, and the naming of experiments. See “Where the Data Is Stored” on page 50 for information on naming experiments, including MPI experiments.

Each MPI process that collects performance data creates its own experiment. When an MPI process creates an experiment, it locks the experiment directory. All other MPI processes must wait until the lock is released before they can use the directory. Thus, if you store the experiments on a file system that is accessible to all MPI processes, the experiments are created sequentially, but if you store the experiments on file systems that are local to each MPI process, the experiments are created concurrently.

If you store the experiments on a common file system and specify an experiment name in the standard format, *experiment*.*n*.er, the experiments have unique names. The value of *n* is determined by the order in which MPI processes obtain a lock on the experiment directory, and cannot be guaranteed to correspond to the MPI rank of the process. If you attach dbx to MPI processes in a running MPI job, *n* will be determined by the order of attachment.

If you store the experiments on a local file system and specify an experiment name in the standard format, the names are not unique. For example, suppose you ran an MPI job on a machine with 4 single-processor nodes labelled node0, node1, node2 and node3. Each node has a local disk called /scratch, and you store the experiments in directory *username* on this disk. The experiments created by the MPI job have the following full path names.

```
node0:/scratch/username/test.1.er
node1:/scratch/username/test.1.er
node2:/scratch/username/test.1.er
node3:/scratch/username/test.1.er
```

The full name including the node name is unique, but in each experiment directory there is an experiment named test.1.er. If you move the experiments to a common location after the MPI job is completed, you must make sure that the names remain unique. For example, to move these experiments to your home directory, which is assumed to be accessible from all nodes, and rename the experiments, type the following commands.

```
rsh node0 'er_mv /scratch/username/test.1.er test.0.er'
rsh node1 'er_mv /scratch/username/test.1.er test.1.er'
rsh node2 'er_mv /scratch/username/test.1.er test.2.er'
rsh node3 'er_mv /scratch/username/test.1.er test.3.er'
```

For large MPI jobs, you might want to move the experiments to a common location using a script. Do not use the Unix commands cp or mv; see “Manipulating Experiments” on page 147 for information on how to copy and move experiments.

If you do not specify an experiment name, the Sampling Collector uses the MPI rank to construct an experiment name with the standard form *experiment.n*.er, but in this case *n* will be the MPI rank. The stem, *experiment*, is the stem of the experiment group name if you specify an experiment group, otherwise it is test. The experiment names are unique, regardless of whether you use a common file system or a local file system. Thus, if you use a local file system to record the experiments and copy them to a common file system, you will not have to rename the experiments when you copy them and reconstruct any experiment group file.

If you do not know which local file systems are available to you, use the `df -lk` command or ask your system administrator. You should always make sure that the experiments are stored in a directory that already exists, that is uniquely defined and that is not in use for any other experiment. You should also make sure that the file system has enough space for the experiments. See "Estimating Storage Requirements" on page 51 for information on how to estimate the space needed.

Note – If you copy or move experiments between computers or nodes you cannot view the annotated source code or disassembly code unless you have access to the load objects that were used to run the experiment or a copy with the same path and timestamp.

Running the `collect` Command Under MPI

To collect data with the `collect` command under the control of MPI, use the following syntax.

```
% mprun -np n collect [collect-arguments] program-name [program-arguments]
```

Here, *n* is the number of processes to be created by MPI. This procedure creates *n* separate instances of `collect`, each of which records an experiment. Read the section "Where the Data Is Stored" on page 50 for information on where and how to store the experiments.

To ensure that the sets of experiments from different MPI runs are stored separately, you can create an experiment group with the `-g` option for each MPI run. The experiment group should be stored on a file system that is accessible to all MPI processes. Creating an experiment group also makes it easier to load the set of experiments for a single MPI run into the Performance Analyzer. An alternative to creating a group is to specify a separate directory for each MPI run with the `-d` option.

Collecting Data by Starting `dbx` Under MPI

To start `dbx` and collect data under the control of MPI, use the following syntax.

```
% mprun -np n dbx program-name < collection-script
```

Here, *n* is the number of processes to be created by MPI and *collection-script* is a `dbx` script that contains the commands necessary to set up and start data collection. This procedure creates *n* separate instances of `dbx`, each of which records an experiment on one of the MPI processes. If you do not define the experiment name, the experiment will be labelled with the MPI rank. Read the section "Storing MPI Experiments" on page 71 for information on where and how to store the experiments.

You can name the experiments with the MPI rank by using the collection script and a call to `MPI_Comm_rank()` in your program. For example, in a C program you would insert the following line.

```
ier = MPI_Comm_rank(MPI_COMM_WORLD,&me);
```

In a Fortran program you would insert the following line.

```
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
```

If this call was inserted at line 17, for example, you could use a script like this.

```
stop at 18
run program-arguments
rank=$[me]
collector enable
collector store filename experiment.$rank.er
cont
quit
```

CHAPTER 4

Analyzing Program Performance Using the Performance Analyzer Graphical Interface

This chapter discusses the Performance Analyzer, its performance metrics, and how to use it. The Performance Analyzer analyzes the program performance data that is collected by the Sampling Collector. It provides you with options for examining and manipulating the experiment data, so that you can identify program execution bottlenecks and analyze and improve program performance.

This chapter covers the following topics.

- Types of Performance Metrics and How to Use Them
- How Metrics Are Assigned to Program Structure
- Running the Performance Analyzer
- Selecting Experiments
- Filtering the Data to Be Displayed
- Examining Metrics for Functions and Load Objects
- Examining Callers-Callees Metrics for a Function
- Examining Annotated Source Code and Disassembly Code
- Generating and Using a Mapfile
- Examining Information About Samples
- Examining Execution Statistics
- Examining Address-Space Information
- Printing the Display

For an introduction to using the Performance Analyzer and examples of how you might use the Performance Analyzer to fine-tune an application, see Chapter 2.

For a more detailed description of how the Performance Analyzer analyzes data and relates it to program structure, see Chapter 6.

Types of Performance Metrics and How to Use Them

The Performance Analyzer reads the raw data collected by the Sampling Collector and converts it into measures of program performance called metrics. The three types of profiling data are converted into three types of metrics: clock-based data is converted into timing metrics, synchronization wait tracing data is converted into synchronization metrics and hardware-counter overflow data is converted into count metrics.

Timing Metrics

The following metrics are computed from clock-based data:

- **User CPU time.** LWP time spent running in user mode on the CPU.
- **Wall time.** LWP time spent in LWP 1. This is the "wall clock time."
- **Total LWP time.** Sum of all LWP times.
- **System CPU time.** LWP time running in kernel mode on the CPU or in a trap state.
- **Wait CPU time.** LWP time spent waiting for the CPU.
- **Text page fault time.** LWP time spent waiting for a text page.
- **Data page fault time.** LWP time spent waiting for a data page.
- **Other wait time.** LWP time spent waiting for a lock or for a kernel page, or spent sleeping or stopped.

For multithreaded experiments, times other than wall clock time are summed across all LWPs. Wall time as defined is not meaningful for multiple-program multiple-data (MPMD) programs.

Timing metrics tell you where your program spent time in several categories and can be used to improve the performance of your program.

- High user CPU time tells you where the program did most of the work. It can be used to find the parts of the program where there may be the most gain from redesigning the algorithm.
- High system CPU time tells you that your program is spending a lot of time in calls to system routines.
- High wait CPU time tells you that there are more threads ready to run than there are CPUs available, or that other processes are using the CPUs.

- High text page fault time means that the code generated by the linker is organized in memory so that calls or branches cause a new page to be loaded. Creating and using a mapfile (see "Generating and Using a Mapfile" on page 97) can fix this kind of problem.
- High data page fault time indicates that access to the data is causing new pages to be loaded. Reorganizing the data structure or the algorithm in your program can fix this problem.

Synchronization Delay Metrics

The following metrics are computed from synchronization wait tracing data:

- **Synchronization delay events.** The number of calls to a synchronization routine where the wait time exceeded the prescribed threshold.
- **Synchronization wait time.** Total of wait times that exceeded the prescribed threshold.

From this information you can determine if functions or load objects are either frequently left on hold, or experience unusually long wait times when they do make a call to a synchronization routine. High synchronization wait times indicate contention among threads. You can reduce the contention by redesigning your algorithms, particularly restructuring your locks so that they cover only the data for each thread that needs to be locked.

Count Metrics

Hardware-counter overflow profiling data is converted by the Performance Analyzer into count metrics. For counters that count in cycles, the metrics reported are converted to times; for counters that do not count in cycles, the metrics reported are event counts.

Some of the more common hardware counters count instruction-cache misses, data-cache misses, cycles, cache stall cycles, floating-point operations, instructions issued or executed. High counts of cache misses, for example, indicate that restructuring your program to improve data or text locality or to increase cache reuse can improve program performance.

How Metrics Are Assigned to Program Structure

Metrics are assigned to program instructions using the call stack that is recorded with the event-specific data. Each instruction is mapped to a line of source code and the metrics assigned to that instruction are also assigned to the line of source code. See Chapter 6 for a more detailed explanation of how this is done. See "Examining Annotated Source Code and Disassembly Code" on page 95 for information on how the Performance Analyzer presents metrics at the source code level and the instruction level.

In addition to source code and instructions, metrics are assigned to higher level objects: functions and load objects. The call stack contains information on the sequence of function calls made to arrive at the instruction address recorded when a profile was taken. The Performance Analyzer uses the call stack to compute metrics for each function in the program. These metrics are called function-level metrics.

Function-Level Metrics: Exclusive, Inclusive, and Attributed

The Performance Analyzer computes three types of function-level metrics: exclusive metrics, inclusive metrics and attributed metrics.

- Exclusive metrics for a function are calculated from events which occur inside the function itself: they exclude metrics coming from calls to other functions.
- Inclusive metrics are calculated from events which occur inside the function and any functions it calls: they include metrics coming from calls to other functions.
- Attributed metrics tell you how much of a metric came from calls from or to another function: they attribute metrics to another function.

For a function at the bottom of a particular call stack, the exclusive and inclusive metrics are the same, because the function makes no calls to other functions.

Exclusive and inclusive metrics are also computed for load objects. Exclusive metrics for a load object are calculated by summing the function-level metrics over all functions in the load object. Inclusive metrics for load objects are calculated in the same way as for functions.

Exclusive and inclusive metrics for a function give information about all paths through the function. Attributed metrics give information about particular paths through a function. They show how much of a metric came from a particular function call. The two functions involved in the call are described as a *caller* and a *callee*. For each function in the call tree:

- The attributed metrics for a function's callers tell you how much of the function's inclusive metric was due to calls from each caller. The attributed metrics for the callers add up to the function's inclusive metric.
- The attributed metrics for a function's callees tell you how much of the function's inclusive metric came from calls to each callee. Their sum plus the function's exclusive metric equals the function's inclusive metric.

Comparison of attributed and inclusive metrics for the caller or the callee gives further information:

- The difference between a caller's attributed metric and its inclusive metric tells you how much of the metric came from calls to other functions and from work in the caller itself.
- The difference between a callee's attributed metric and its inclusive metric tells you how much of the callee's inclusive metric came from calls to it from other functions.

To locate places where you could improve the performance of your program:

- Use exclusive metrics to locate functions that have high metric values.
- Use inclusive metrics to determine which call sequence in your program was responsible for high metric values.
- Use attributed metrics to trace a particular call sequence to the function or functions that are responsible for high metric values.

Interpreting Function-Level Metrics: An Example

Exclusive, inclusive and attributed metrics are illustrated in FIGURE 4-1, which contains a fragment of a call tree. The focus is on the central function, function C. There may be calls to other functions which do not appear in this figure.

Function C calls two functions, function E and function F, and attributes 10 units of its inclusive metric to function E and 10 units to function F. These are the callee attributed metrics. Their sum (10+10) added to the exclusive metric of function C (5) equals the inclusive metric of function C (25).

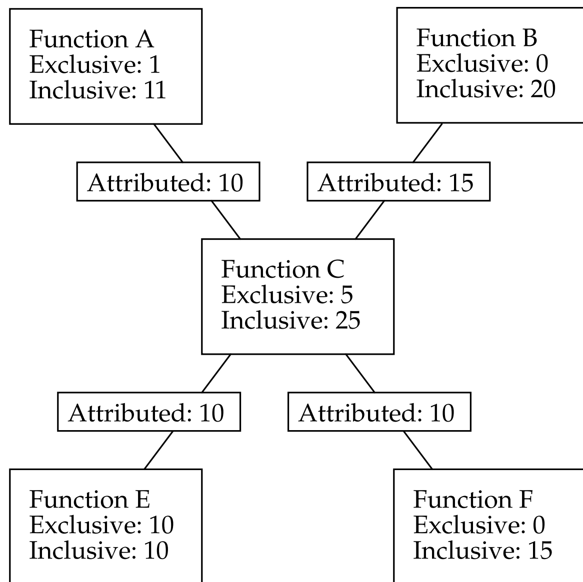


FIGURE 4-1 Call Tree Illustrating Exclusive, Inclusive, and Attributed Metrics

The callee attributed metric and the callee inclusive metric are the same for function E but different for function F. This means that function E is only called by function C but function F is called by some other function or functions. The exclusive metric and the inclusive metric are the same for function E but different for function F. This means that function F calls other functions, but function E does not.

Function C is called by two functions: function A and function B, and attributes 10 units of its inclusive metric to function A and 15 units to function B. These are the caller attributed metrics. Their sum (10+15) equals the inclusive metric of function C.

The caller attributed metric is equal to the difference between the inclusive and exclusive metric for function A, but it is not equal to this difference for function B. This means that function A only calls function C, but function B calls other functions besides function C. (In fact, function A might call other functions but the time is so small that it does not appear in the experiment.)

How Recursion Affects Function-Level Metrics

Recursive function calls, whether direct or indirect, complicate the calculation of metrics. The Performance Analyzer displays metrics for a function as a whole, not for each invocation of a function: the metrics for a series of recursive calls must therefore be compressed into a single metric. This does not affect exclusive metrics, which are calculated from the function at the bottom of the call stack (the "leaf function"), but it does affect inclusive and attributed metrics.

Inclusive metrics are computed by adding the exclusive metric for the leaf function to the inclusive metric of the functions in the call stack. To ensure that the metric is not counted multiple times in a recursive call stack, the exclusive metric for the leaf function is only added to the inclusive metric for each unique function.

Attributed metrics are computed from inclusive metrics. In the simplest case of recursion, a recursive function has two callers: itself and another function (the initiating function). If all the work is done in the final call, the inclusive metric for the recursive function will be attributed to itself and not to the initiating function. This is because the inclusive metric for all the higher invocations of the recursive function are regarded as zero to avoid multiple counting of the metric. The initiating function, however, correctly attributes to the recursive function as a callee the portion of its inclusive metric due to the recursive call.

Running the Performance Analyzer

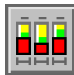
The Performance Analyzer can be started from the command line or from the Sun WorkShop integrated programming environment.

To start the Performance Analyzer from the Sun WorkShop integrated programming environment, you have three options:

- **Choose Tools ➤ Analyzer from the menu bar.**

  A menu is displayed from which you can choose to open a new Analyzer window, load a particular experiment or edit an experiment list. See “Tools Menu” in the Main Window section of the Sun WorkShop online help for details.

- **Click the Analyzer button in the tool bar of the main window.**

  The Performance Analyzer automatically loads the last experiment you analyzed. If you have not previously analyzed an experiment, the Load Experiment dialog box is displayed to enable you to select an experiment.

- **Click the Analyzer button in the tool bar of the Sampling Collector window.**

  The Performance Analyzer automatically loads the most recent experiment that was collected.

To start the Performance Analyzer from the command line, use the `analyzer(1)` command. The syntax of the `analyzer` command is shown here.

```
% analyzer [-s session-id] [-V] [experiment-name [, experiment-name2 … ]]
```

See "Where the Data Is Stored" on page 50 for information on experiment names. If you omit the experiment name, the Load Experiment dialog box is displayed when the Performance Analyzer starts. If you give more than one experiment name, the data for all experiments are added in the Performance Analyzer. You can load an experiment group by giving the group name instead of an experiment name.

The options for the `analyzer` command are described in TABLE 4-1.

TABLE 4-1 Options for the `Analyzer` Command

`-s` *session-name*	Attaches a user-defined identifier to the Performance Analyzer instance
`-V`	Prints the version number of the Performance Analyzer to `stdout`

The Analyzer window is the main display that you see when you open the Performance Analyzer. The window contains a menu bar, upper and lower tool bars, and a data display pane which is divided into two panels.

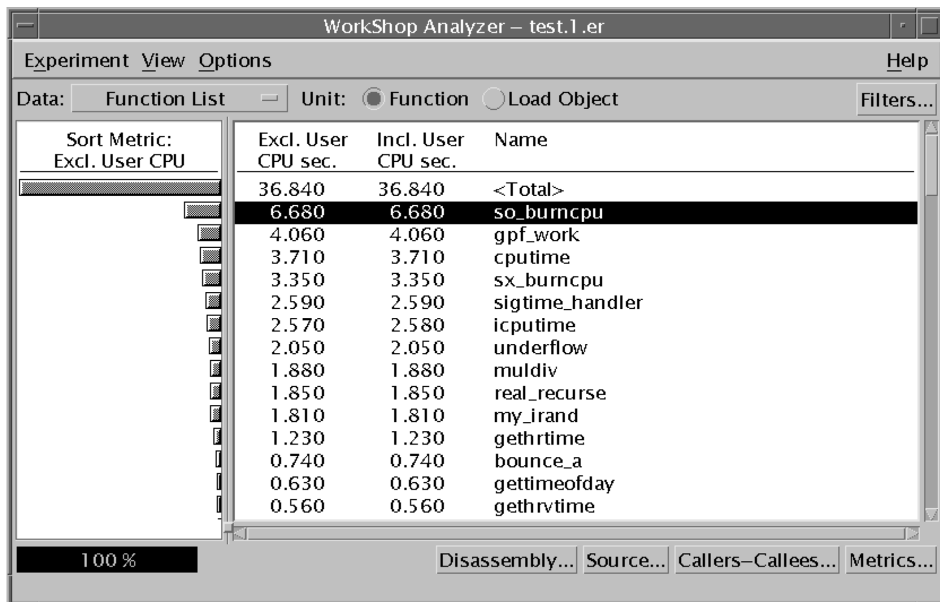


FIGURE 4-2 The Analyzer Window

The Performance Analyzer has four main data types which can be displayed: Function List, Overview, Address Space and Execution Statistics. These are discussed in later sections. You can select them from the Data menu in the upper tool bar. The majority of the data and functionality is available from the Function List display. Function List data is displayed by default when you open the Performance Analyzer.

The Performance Analyzer computes a single set of performance metrics for the data which is loaded. The data can come from a single experiment, from a predefined experiment group or from several experiments.

To compare two selections of metrics from the same set, you can open a new Analyzer window by choosing View ➤ Open New Window from the menu bar. To dismiss this window, choose Experiment ➤ Close from the menu bar in the new window.

To compute and display more than one set of metrics—if you want to compare two experiments, for example—you must start an instance of the Performance Analyzer for each set.

If you are running multiple instances of the Performance Analyzer, they will share an editor window unless you tag each instance with a separate session name. The editor window is used to display annotated source code or annotated disassembly code. Tagging the Performance Analyzer instances enables you to compare annotated source or disassembly code for different experiments in different editor windows. You can only tag Performance Analyzer instances using the `analyzer` command.

To exit the Performance Analyzer, choose Experiment ➤ Exit from the Analyzer window menu bar.

Note – Information on how to use all the windows in the Performance Analyzer and them is available in the online help. Click Help in any window to see the online help for that window.

Selecting Experiments

The Performance Analyzer allows you to compute metrics for a single experiment, from a predefined experiment group or from several experiments. This section tells you how to load, add and drop experiments from the Performance Analyzer.

Loading an Experiment Into the Performance Analyzer

Loading an experiment clears all experiment data from the Performance Analyzer and reads in a new set of data. (It has no effect on the experiments as stored on disk.)

To load an experiment or an experiment group into the Performance Analyzer at any time:

1. **Choose Experiment ➤ Load from the Analyzer window menu bar to open the Load Experiment dialog box.**

   If you start the Performance Analyzer from Sun WorkShop, or if you do not give an experiment name when you start the Performance Analyzer from the command line, the Load Experiment dialog box opens automatically.

2. **In the Load Experiment dialog box, double-click the experiment or experiment group name in the list box, or enter the name in the text box.**

   Click the Help button to obtain information on navigating the dialog box.

Adding Experiments to the Performance Analyzer

Adding an experiment to the Performance Analyzer reads a set of data into a new storage location in the Performance Analyzer and recomputes all the metrics. The data for each experiment is stored separately, but the metrics displayed are the combined metrics for all experiments. This capability is useful when you have to record data for the same program in separate runs—for example, if you want timing data and hardware counter data for the same program.

To add an experiment to any experiments already loaded into the Performance Analyzer:

1. **Choose Experiment ➤ Add from the Analyzer window menu bar to open the Add Experiment dialog box.**

2. **In the Add Experiment dialog box, double-click the experiment you want to add, or type the name of the experiment in the text box.**

To examine the data collected from an MPI run, open one experiment in the Performance Analyzer, then add the others, so you can see the data for all the MPI processes in aggregate. If you have defined an experiment group, loading the experiment group has the same effect.

Dropping Experiments From the Performance Analyzer

Dropping an experiment clears the data for that experiment from the Performance Analyzer, and recomputes the metrics. (It has no effect on the experiment files.)

To drop an experiment from the Performance Analyzer:

1. **Choose Experiment ➤ Drop to open the Drop Experiment dialog box.**
2. **In the list box, click the experiment you want to drop from the Performance Analyzer.**
3. **Click either Apply to drop the experiment and leave the dialog box open, or OK to drop the experiment and close the dialog box.**

If you have loaded an experiment group, you can only drop individual experiments, not the whole group.

You can drop an experiment from the Performance Analyzer only if more than one experiment is loaded. If only one experiment is loaded, the Drop option is disabled.

Filtering the Data to Be Displayed

You can process information more efficiently if you can focus on the part of your program where you think a problem may be occurring. The Performance Analyzer allows you to filter the experiment information in several ways:

- By experiments, samples, threads, and LWPs
- By load objects

Selecting Experiments, Samples, Threads, and LWPs

You can limit the information in the Performance Analyzer displays by specifying only certain experiments, samples, threads, and LWPs for which to display metrics. Metrics in the Function List and Overview displays appear only for those samples, threads, and LWPs that you select.

To limit the information in the displays, use the following procedure. Some steps can be omitted, and some steps can be repeated.

1. **Click the Filters button in the Analyzer window.**

   The Filters dialog box is displayed.

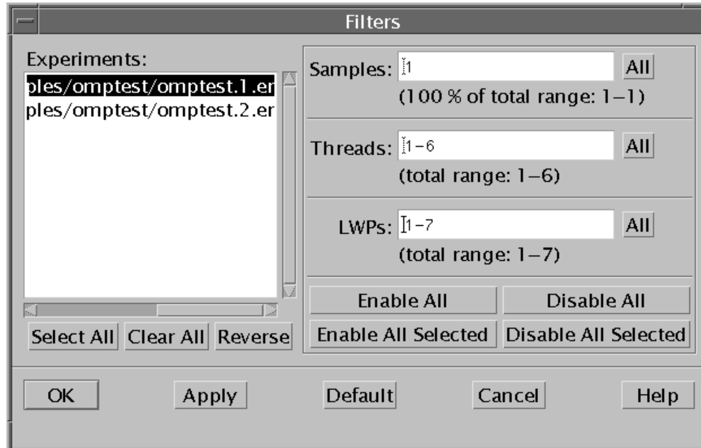


FIGURE 4-3 The Filters Dialog Box

2. **To specify the samples, threads, and LWPs for which you want to display data, enter a comma-separated list in the appropriate text box, or click All.**

   The list can contain ranges of numbers, which are specified by two numbers separated by a hyphen. You can select samples, threads, and LWPs in any number and any combination, but you should ensure that the selection of threads and LWPs makes sense for your application.

3. **To select experiments, use the Experiments list box and the three buttons below it.**

   - Select All selects all experiments.
   - Clear All deselects all experiments.
   - Reverse selects the unselected experiments and deselects the selected experiments.

4. **To apply the choice of samples, threads, and LWPs to experiments, use the four buttons below the text entry boxes.**

   - Enable All applies the choice of samples, threads, and LWPs to all experiments, whether they are selected or not.
   - Enable All Selected applies the choice of samples, threads, and LWPs to the experiments selected in the Experiments list box.
   - Disable All disables the display of data for all experiments, whether they are selected or not. If you make this choice, no data is displayed.

- Disable All Selected disables the display of data for the experiments selected in the Experiments list box. The choice of samples, threads, and LWPs is ignored.

Each experiment can have its own selection of threads, samples and LWPs. If you specify a selection for an experiment more than once, the last selection supersedes all others.

5. **To apply a selection to the displays and make another selection, click Apply.**
6. **To apply the current selection to the displays and exit the Filters dialog box, click OK.**

Selecting Load Objects

For purposes of performance analysis, you probably do not want to display information about all the load objects in your program; for example, you might want to see only the metrics that apply to your program files, and not to any system libraries. The Performance Analyzer allows you to specify which load objects you want to examine metrics for in the Function List and Overview displays.

To select one or more load objects for which to display information:

1. **Choose View ➤ Select Load Objects Included to open the Select Load Objects Included dialog box.**
2. **In the list box, click the files you do not want to display to deselect them. If a file that you want to display is not selected, click it to select it. You can also use the Select All and Clear All buttons select or deselect all the load objects listed.**
3. **Click OK to apply your selections and close the Select Load Objects Included dialog box.**

Examining Metrics for Functions and Load Objects

When you open the Performance Analyzer, the Analyzer window displays Function List data by default. Function List data consists of metrics specific to functions and load objects. The data display pane is divided into two display panels:

- The left display panel contains a histogram representation of the metric on which the data is sorted.

- The right display panel shows a table of function or load-object metrics. The name of the function or load object to which the data in that row applies is shown to the right of each row in the table.

To choose between the display of data for functions and for load objects, use the Function and Load Object radio buttons in the upper tool bar.

The default metrics shown in the Function List display are read from a defaults file. In the absence of any user defaults files, the system defaults file is read. A defaults file can be stored in a user's home directory, where it will be read each time the Performance Analyzer is started, or in any other directory, where it will be read when the Performance Analyzer is started from that directory. The user defaults files, which must be named `.er.rc`, can contain selected `er_print` commands. See "Defaults Commands" on page 118 for more details. The selection of metrics to be displayed, the order of the metrics and the sort metric can be specified in the defaults file. The system defaults file specifies the following metrics to be displayed if they are present in an experiment:

- Exclusive user CPU time
- Inclusive user CPU time
- Exclusive hardware-counter overflow profiling metrics
- Inclusive hardware-counter overflow profiling metrics
- Inclusive synchronization wait time
- Inclusive synchronization wait counts

For each function or load-object metric displayed, the Function List system defaults select a value in seconds or in counts, depending on the metric. The lines of the display are sorted on the first metric in the default list.

For C++ programs, you can display the long or the short form of a function name. To select the form, choose Options ➤ Set Function Names Format. The default is long. This choice can also be set up in the defaults file.

Selecting Metrics and Sort Order for Functions and Load Objects

If you suspect that your program performance is being affected by a particular problem, you can limit what appears in the Function List display to metrics reflecting only that problem.

- **To change the selection of metrics, metric units, sort order and column ordering in the Function List display, click the Metrics button.**

The Function List Metrics dialog box is displayed.

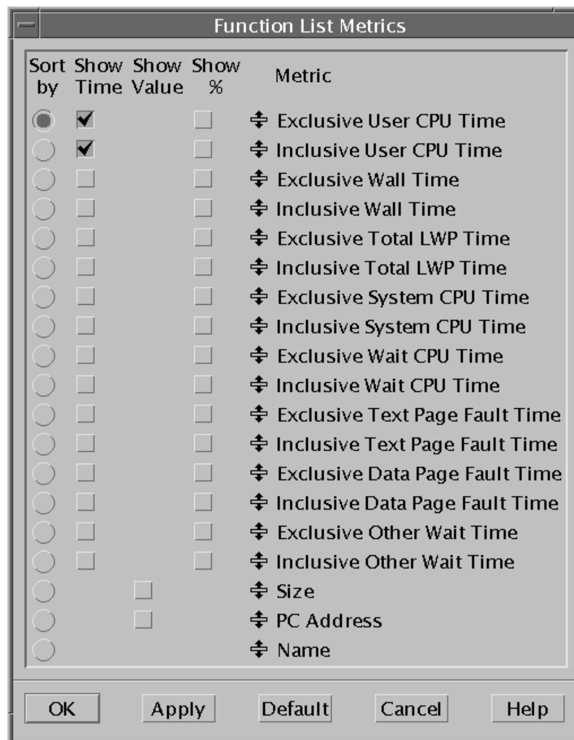


FIGURE 4-4 The Function List Metrics Dialog Box

The Function List Metrics dialog box lists the metrics which are available in the experiment set you have loaded into the Performance Analyzer. You can choose to display any combination of these. In addition, you can choose to display the size, in bytes, and the program-counter address. The names of functions or load objects are always displayed.

The order of the list determines the order of the columns of metrics in the Analyzer window data display pane, and is initially determined by the defaults file.

All metrics are available as either a time in seconds or a count, and as a percentage of the total program metric. Hardware counter metrics where the count is in cycles are available as a time, a count, and a percentage.

1. **To choose the metrics you want to be displayed and their units, click the appropriate check boxes in the “Show time”, “Show value” and “Show %” columns.**

2. **To specify the sort order, click the appropriate radio button in the “Sort by” column.**

3. **To change the order of the metrics list, drag the icon next to the name of the metric that you want to move and drop it onto the metric above which you want it to appear.**

4. **To apply your selections to the Function List display, click OK to apply the new selections and close the Metrics dialog box, or click Apply to apply the new selections and keep the dialog box open.**

Viewing Summary Metrics for a Function or a Load Object

You can use the Summary Metrics window to view the entire set of metrics and other information for a selected function or load object in table form instead of as part of the Function List display.

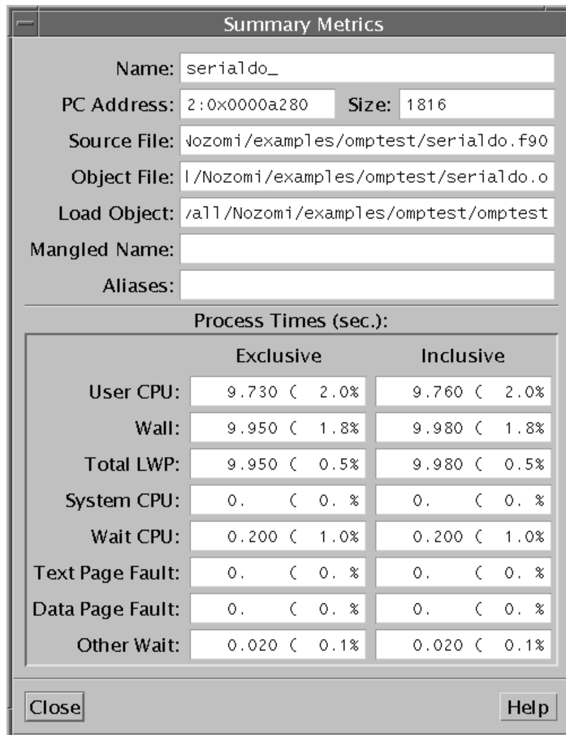


FIGURE 4-5 The Summary Metrics Window

To see summary metrics for a function or load object:

1. **Use the Unit radio buttons to select function display or load-object display.**

2. **Click the function or load object in the Function List display.**

3. **Choose View ➤ Show Summary Metrics to open the Summary Metrics window.**

   The Summary Metrics window is divided into two sections. The upper section lists the name, address and size of the selected function or load object, and for functions, the source file, object file, and load object where code for the function resides. The lower section displays all the metrics recorded in the experiment, exclusive and inclusive, for the selected function or load object. The display is not affected by the selection of metrics in the Function List display.

Note – All data in the Summary Metrics window can be copied to the clipboard and pasted into any text editor.

Searching for a Function or Load Object

The Performance Analyzer includes a search tool that you can use to locate a function or load object in the Function List display.

To search for a particular function or load object:

1. **Choose View ➤ Find to display the Find dialog box.**

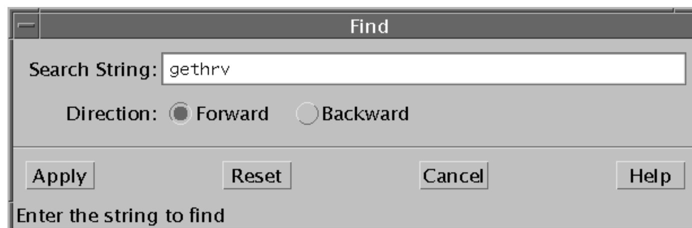


FIGURE 4-6 The Find Dialog Box

2. **Type the string you want to find in the Search String text box.**

Note – The Performance Analyzer Find feature uses UNIX regular expressions. Thus, where `c` is any character, `c*` does not indicate the string consisting of `c` followed by zero or more other characters, but zero or more instances of `c`. For a complete description of UNIX regular expressions, see the `regexp`(5) man page.

---

3. **Specify the search direction by clicking one of the Direction radio buttons. The default is Forward.**
4. **Click Apply.**

   If the search is successful, the row of data for the function that you searched on is highlighted in the Function List display and the list is scrolled so that the function is at the top of the pane.
5. **To search for other function names matching the search string, click Apply.**

   The search loops back to the beginning of the list when it reaches the end.
6. **To reset the string in the Search String text box to the string which was last found, click Reset.**

Examining Callers-Callees Metrics for a Function

To examine caller and callee metrics for a selected function:

1. **Click the function you want to select in the Function List display.**
2. **Click the Callers-Callees button in the lower tool bar.**

WorkShop Analyzer – Callers–Callees Window – test.1.er

Sort Metric: Attr. User CPU	Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
	4.060	0.	36.840	commandline
	0.	0.	4.060	gpf
	3.720	0.	3.720	gpf_b
	0.340	0.	0.340	gpf_a

Metrics... Close Help

FIGURE 4-7 The Callers-Callees Window

The Callers-Callees window contains a center pane with information about the selected function, an upper pane (the Callers pane) with information about the function's caller, and a lower pane (the Callees pane) with information about the function's callees, if any. Each of these panes is divided into two panels:

- The left panel contains a histogram representation of the metric on which the data is sorted.
- The right panel shows a table of function metrics; to the right of each row in the table is the name of the function to which the data in that row applies.

The Callers-Callees window can display exclusive, inclusive and attributed metrics for the selected function, its callers and its callees, if the supporting data was collected by the Sampling Collector. Each of these metrics can be displayed as a time or an event count, and as a percentage. Metrics for hardware counters which count in cycles can be displayed as a time, an event count and a percentage. For attributed metrics, for both callers and callees, the percentage given is the percentage that the attributed metric contributes to the function's inclusive metric. For exclusive and inclusive metrics, it is the percentage of the total program metric.

The default metrics are derived from the metrics in the Function List display. If these have not been changed, the following metrics are displayed if they are present in an experiment:

- Attributed user CPU time
- Exclusive user CPU time
- Inclusive user CPU time
- Attributed hardware-counter overflow profiling metrics
- Exclusive hardware-counter overflow profiling metrics
- Inclusive hardware-counter overflow profiling metrics
- Attributed synchronization wait time
- Inclusive synchronization wait time
- Attributed synchronization wait counts
- Inclusive synchronization wait counts

For each function displayed, the metric is displayed in seconds or in counts, depending on the metric. The lines of the display are sorted on the first attributed metric in the default list.

You can navigate through your program's structure by clicking on a function in either the Caller pane or the Callee pane. The display recenters on the newly selected function. By observing exclusive, inclusive, and attributed metrics, you can locate any function that is responsible for large amounts of a given metric.

For recursive function calls, the function can be its own caller and its own callee. To simplify the display in the Callers-Callees window, the recursive function is shown as a caller of itself, but not as a callee.

Selecting Metrics and Sort Order in the Callers-Callees Window

The Callers-Callees window has its own dialog box for selecting metrics and sort order, because in addition to exclusive and inclusive metrics, it displays attributed metrics. To specify the data displayed in the Callers-Callees window and its sort order, click Metrics in the Callers-Callees window. The Callers-Callees Metrics dialog box is displayed.

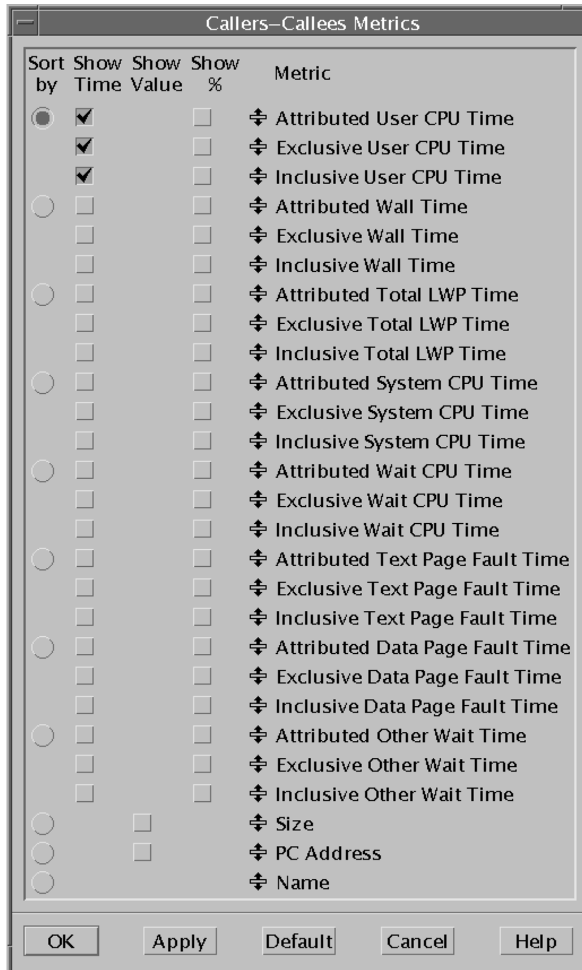


FIGURE 4-8 The Callers-Callees Metrics Dialog Box

The Callers-Callees Metrics dialog box lists the metrics that are supported by the experiment set you have loaded into the Performance Analyzer. You can choose to display any combination of these. In addition, you can choose to display the following information:

- Size, in bytes
- Program-counter (PC) address

The names of functions are always displayed.

The order of the list determines the order of the columns of metrics in the Callers-Callees Window data display pane.

- **To choose the metrics you want to display and their units, click the appropriate check boxes in the "Show time", "Show value" and "Show %" columns.**
- **To specify the sort order, click the appropriate radio button in the "Sort by" column.**

  For any metric, you can sort only on the attributed value.

- **To change the order of the metrics list, drag the icon next to the name of the metric that you want to move and drop it onto the metric above which you want it to appear.**
- **To apply your selections to the Callers-Callees display, click OK to close the dialog box, or click Apply to keep the dialog box open.**

Examining Annotated Source Code and Disassembly Code

When you have identified the function or functions that are causing performance problems, you can generate source code or disassembly code annotated with performance metrics, so you can identify the actual lines or instructions that are causing the problem.

To display annotated source code or disassembly code for a function:

1. **Click the line containing the function in the Function List display to select it.**
2. **Click Source or Disassembly in the lower tool bar of the Analyzer window.**

   Your text editor opens, showing the code for the selected function, with performance metrics for each line of code or instruction displayed to the left. To select a text editor, choose Options ➤ Text Editor Options.

The metrics displayed in the annotation are those that appear in the Function List display at the time you invoke the source code or disassembly code display. To change the metrics, use the Function List Metrics dialog box to change the Function List metrics, then reinvoke the annotated source code or disassembly code display.

The lines that contain metric values that are greater than a specified percentage of the maximum metric value for each column are highlighted, and the entry point for the function you selected is also highlighted.

If you choose to display annotated disassembly code and the source for your program is available, the source code appears interleaved with the disassembly code listing.

Information on where to find the source code is stored in the load objects used by your program. When the experiment is completed, an archive file is generated for each load object which among other things contains the path to the load object and the time stamp of its last modification. These are checked when annotated source or disassembly code displays are requested. If you copy an experiment from one computer to another you will not be able to view the annotated source code or annotated disassembly code unless you have access to the load objects which were used to run the experiment or a copy with the same path and time stamp.

When annotated source code is requested, the Performance Analyzer looks for the file containing the selected function under the absolute path name as recorded in the executable and stored in the experiment. If it is not there, it tries to find a file of the same basename in the current working directory, and use it. If you have moved the source files, or the experiment was recorded on a different file system, you can put a symbolic link from the current directory to the actual source location in order to see the annotated source code.

For both source and disassembly listings, compiler commentary is also interleaved with the source code to which it is related. The types of compiler commentary which are displayed can be set up in a defaults file, which contains `er_print` commands. See “Source and Disassembly Listing Commands” on page 112 for commands to select compiler commentary types and “Defaults Commands” on page 118 to find out about defaults files.

Annotated source and disassembly code are discussed further in Chapter 6. The possible metric values that can appear in the annotation are explained in TABLE 6-2.

Generating and Using a Mapfile

Using the data from the experiment, the Performance Analyzer can generate a mapfile that you can use with the static linker (`ld`) to create an executable with a smaller working-set size, more effective instruction cache behavior, or both. The mapfile provides the linker with an order in which it loads the routines.

To create the mapfile:

1. **Use the Sampling Collector to collect performance data (see "Collecting Data From the Sun WorkShop Integrated Programming Environment" on page 60).**

   The order of the routines in the mapfile is determined by the sort order in the Function List display. If you want to use a particular metric to order the routines, you must collect the corresponding performance data.

2. **Load the experiment that you have just generated into the Performance Analyzer (see "Running the Performance Analyzer" on page 81).**

3. **Select the sort metric you want to use to order the routines using the Metrics dialog box.**

4. **Choose Experiment ➤ Create Mapfile. The Create Mapfile dialog box is displayed.**

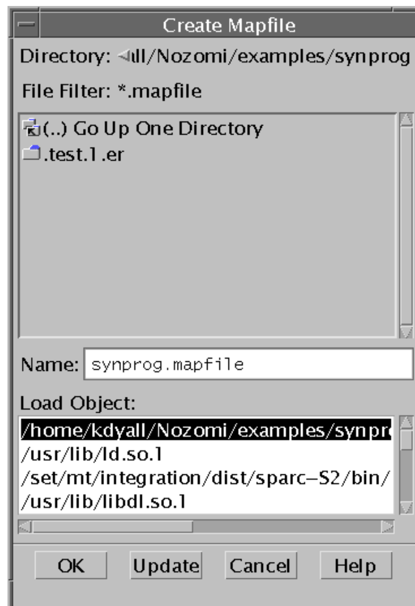


FIGURE 4-9 The Create Mapfile Dialog Box

5. **In the Create Mapfile dialog box, choose the name of the mapfile using the Name text box and the directory list box.**

6. **In the Select Load Object list box, select the load object for which you want to generate the map file (this is usually your program segment).**

7. **Click OK.**

   To use the mapfile to reorder your program:

1. **Ensure that your program is compiled using the `-xF` option, which causes the compiler to generate functions that can be relocated independently.**

2. **Link your program with the `-M` option.**

   For C programs, type the following.

```
% cc -xF -c source-file-list
% cc -Wl -M mapfile-name -o program-name  object-file-list
```

For C++ programs, type the following.

```
% CC -xF -c source-file-list
% CC -M mapfile-name -o program-name  object-file-list
```

For Fortran programs, type the following.

```
% f95 -xF -c source-file-list
% f95 -M mapfile-name -o program-name  object-file-list
```

If you see the following warning message, check any files that are statically linked, such as unshared object and library files, to ensure that these files have been compiled with the `-xF` option.

```
ld: warning: mapfile: text: .text% function-name: object-file-name:
Entrance criteria not met named-file, function-name, has not been
compiled with the -xF option.
```

Examining Information About Samples

You can examine information about samples from the Overview display.

- **To view the Overview display, choose Overview from the Data list box.**

The Overview display contains information about process times during part or all of program execution. It is divided into two panes:

- The left display pane contains a bar showing the percentage of the time spent in various process states averaged over the sample or range of samples selected.
- The right display pane contains a series of bars showing the time spent in various process states for each sample selected for display. Each bar represents the sampling information collected by the Sampling Collector during a single sampling interval. The sample's ID number appears above the sample; the wall time appears below the samples.

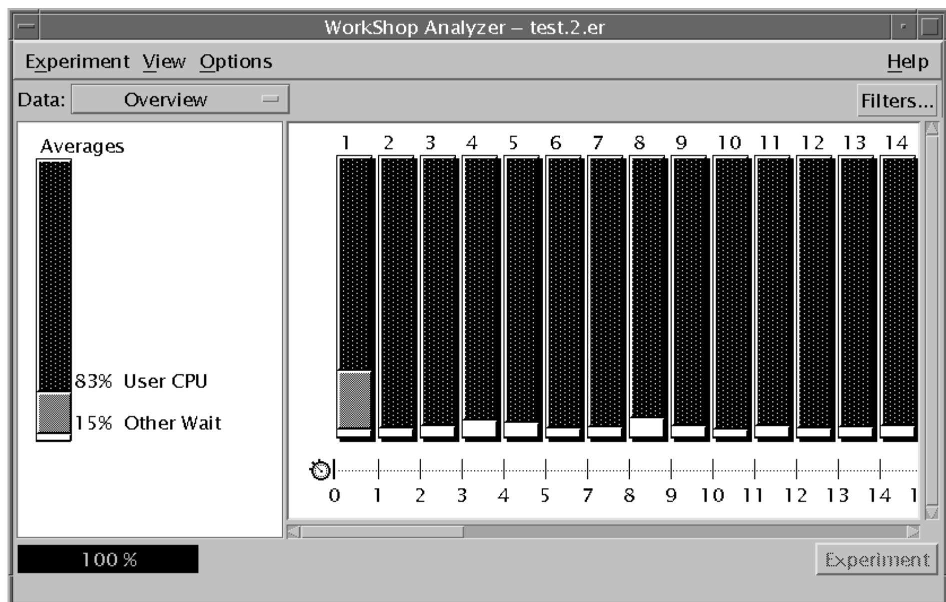


FIGURE 4-10 The Overview Display

The default display for the samples in the Overview display is Fixed Width—that is, the sample bars are all the same width, whether each sampling interval is the same length or not. To change the display to a proportional representation of the samples based on the length of each sample interval use the Options ➤ Set Overview Column Width menu.

Overview information is displayed separately for each experiment. To select the experiment for which you want to see overview information, use the Experiment button in the lower tool bar.

You can see a more detailed analysis of the sample information than is presented in the Overview display, including process state metrics too small to show up in the sampling graphs. To obtain detailed information about specific samples, you must select the samples. To select samples, use the Filters button. See "Selecting Experiments, Samples, Threads, and LWPs" on page 85 for instructions. By default, all samples are selected.

Note – When a sample is selected using the Filters dialog box, a drop shadow appears behind it in the right Overview display pane.

- **To view sample details, choose View ➤ Show Sample Details from the menu bar.**

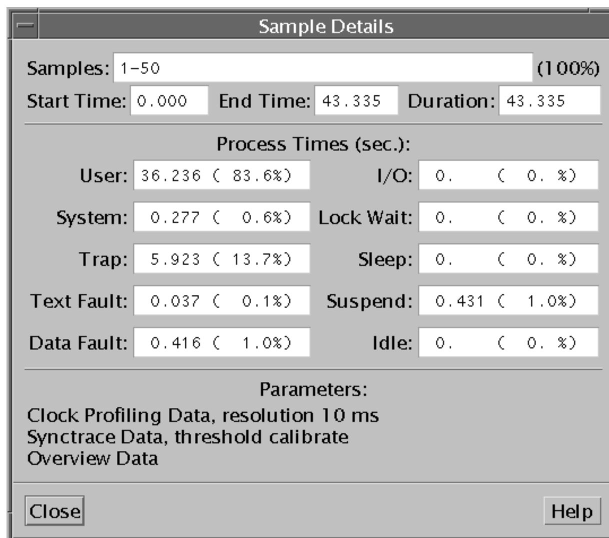


FIGURE 4-11 The Sample Details Window

The Sample Details window is displayed, showing the following metrics:

- The ID of the sample or samples
- The percentage of the total samples selected
- The sampling start time, end time, and duration, in seconds
- A listing of process states and the time spent in each state, represented in seconds and in percentage of the total metric for all the samples selected
- A parameter list showing the types of data that the Sampling Collector recorded in the experiment and the parameters with which the data was recorded

Examining Execution Statistics

The Execution Statistics display lists various system statistics, summed over the selected sample or samples. (For information on how to select and group samples, see “Selecting Experiments, Samples, Threads, and LWPs” on page 85.)

- **To view the Execution Statistics display, choose Execution Statistics from the Data list box.**

Note – All data in the Execution Statistics window can be copied to the clipboard and pasted into any text editor.

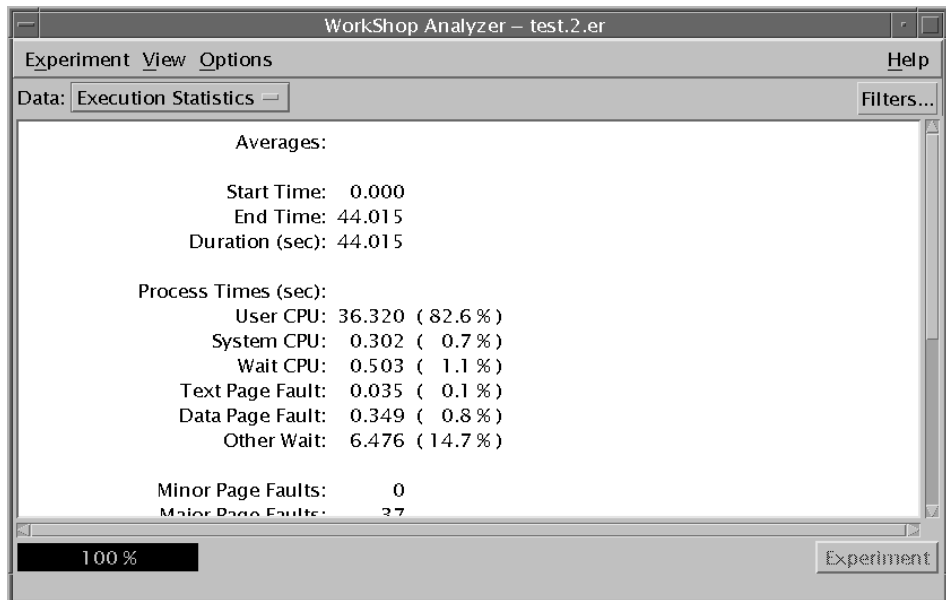


FIGURE 4-12 The Execution Statistics Display

Examining Address-Space Information

Address-space information is available to the Performance Analyzer only if address-space data is selected when the Sampling Collector generates the experiment. Otherwise, the Performance Analyzer reports that address-space data is not available. Address space data is not available on IA hardware.

- **To view the Address Space display, choose Address Space from the Data list box.**

The Address Space display is divided into two display panes:

- The left display pane contains a legend for interpreting the graphical representation on the right.
- The right display pane shows a graphical representation of the program's address space.

The default display is by page (this display also appears if you click the Page radio button). Each square represents a page in the address space, and the fill pattern indicates how your program has affected the page:

- Modified it (written to it)
- Referenced it (read from it)
- Left it unreferenced

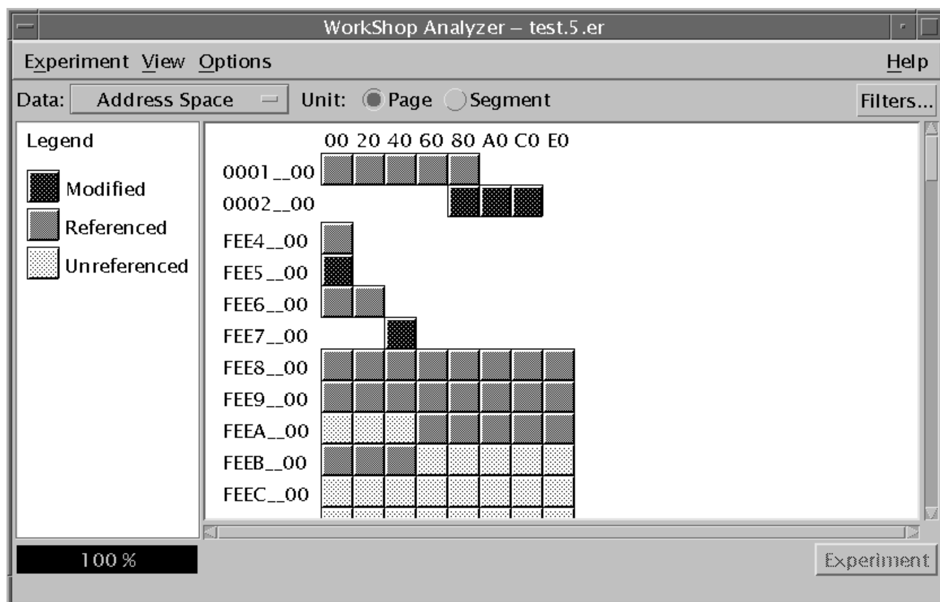


FIGURE 4-13 The Address Space Display

To see a display of the address-space segments, click the Segment radio button in the upper tool bar. The right display pane then shows an undifferentiated representation of the segments of memory used by your program.

To see detailed information about a page or segment:

1. **Use the Unit radio buttons to select page display or segment display.**

2. **Click the page or segment in the right Address Space display pane to select it.**

   When a page or segment is selected, a drop shadow appears behind it in the right Address Space display pane.

3. **Choose View ➤ Show Page Properties or View ➤ Show Segment Properties from the Analyzer window menu bar.**

   The Page Properties window or the Segment Properties window appears, showing the following information:

   - Address of the page or segment
   - Page or segment size in bytes
   - Segment name, if known
   - A list of functions, if any, stored on that page or segment

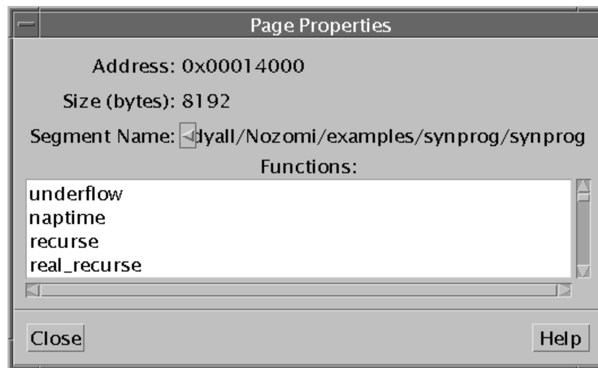


FIGURE 4-14 The Page Properties Window

Printing the Display

To print a text representation of any of the Performance Analyzer displays:

1. **Choose Experiment ➤ Print from the Analyzer window menu bar to open the Print dialog box.**

2. **Use the Print To radio buttons to determine whether you are printing to a printer or a file:**
   - If you are printing to a printer, use the Printer text box to specify a printer name.
   - If you are printing to a file, type the name of the file in the File text box, or use the browse button to open the Print to a File dialog box, in which you can navigate to a directory or file.
3. **Click the Print button.**

Note – If you print the Overview display, you print a listing of statistics for each sample in the experiment, not a text representation of the graphical display.

CHAPTER 5

Analyzing Program Performance Using the er_print Command Line Interface

This chapter explains how to use the er_print utility for performance analysis. The er_print utility prints an ASCII version of the various displays supported by the Performance Analyzer. The information is written to standard output unless you redirect it to a file or printer. You must give the er_print utility the name of one or more experiments or experiment groups generated by the Sampling Collector as arguments. Using the er_print utility you can display metrics of performance for functions, callers and callees; source code and disassembly listings; sampling information; address-space data; and execution statistics.

This chapter covers the following topics.

- er_print Syntax
- Metric Lists
- Function List Commands
- Callers-Callees List Commands
- Source and Disassembly Listing Commands
- Filtering Commands
- Metric List Commands
- Defaults Commands
- Output Commands
- Other Display Commands
- Mapfile Generation Command
- Control Commands
- Information Commands

For a description of the data collected by the Sampling Collector, see Chapter 3.

For instructions on how to use the Performance Analyzer to display information in a graphical format, see Chapter 4.

er_print Syntax

The command-line syntax for er_print is as follows.

```
er_print [ -script script | -command | - ] experiment1 [ experiment2 ... ]
```

The options for er_print are listed in TABLE 5-1.

TABLE 5-1 Options for the er_print Command

Option	Description
-	Read er_print commands entered at the terminal.
-script *script*	Read commands from the file *script*, which contains a list of er_print commands, one per line. If the -script option is not present, er_print reads commands from the terminal or from the command line.
-*command* [*argument*]	Process the given command.

Multiple options can appear on the er_print command line. They are processed in the order they appear. You can mix scripts, hyphens, and explicit commands in any order. The default action if you do not supply any commands or scripts is to enter interactive mode, in which commands are entered from the keyboard. To exit interactive mode type quit or Ctrl-D.

The commands accepted by er_print are listed in the following sections. You can abbreviate any command with a shorter string as long as the command is unambiguous.

Metric Lists

Many of the er_print commands use a list of metric keywords. The syntax of the list is as follows.

```
metric-keyword-1[:metric-keyword2...]
```

Except for the `size`, `address`, and `name` keywords, a metric keyword consists of three parts: a metric type string, a metric visibility string, and a metric name string. These are joined with no spaces, as follows.

```
<type><visibility><name>
```

The metric type and metric visibility strings are composed of type and visibility characters.

The allowed metric type characters are given in TABLE 5-2. A metric keyword that contains more than one type character is expanded into a list of metric keywords. For example, `ie.user` is expanded into `i.user:e.user`.

TABLE 5-2 Metric Type Characters

Character	Description
`e`	Show exclusive metric value
`i`	Show inclusive metric value
`a`	Show attributed metric value (only for callers-callees metrics)

The allowed metric visibility characters are given in TABLE 5-3. The order of the visibility characters in the visibility string does not matter: it does not affect the order in which the corresponding metrics are displayed. For example, both `i%.user` and `i.%user` are interpreted as `i.user:i%user`.

Metrics that differ only in the visibility are always displayed together in the standard order. If two metric keywords that differ only in the visibility are separated by some other keywords, the metrics appear in the standard order at the position of the first of the two metrics.

TABLE 5-3 Metric Visibility Characters

Character	Description
`.`	Show metric as a time. Applies to timing metrics and hardware counter metrics that measure cycle counts. Interpreted as "+" for other metrics.
`%`	Show metric as a percentage of the total program metric. For attributed metrics in the callers-callees list, show metric as a percentage of the inclusive metric for the selected function.
`+`	Show metric as an absolute value. For hardware counters, this value is the event count. Interpreted as a "`.`" for timing metrics.
`!`	Do not show any metric value. Cannot be used in combination with other visibility characters.

When both type and visibility strings have more than one character, the type is expanded first. Thus `ie.%user` is expanded to `i.%user:e.%user`, which is then interpreted as `i.user:i%user:e.user:e%user`.

The visibility characters "`.`", "`+`" and "`%`" are equivalent for the purposes of defining the sort order. Thus `sort i%user`, `sort i.user`, and `sort i+user` all mean "sort by inclusive user CPU time if it is visible in any form", and `sort i!user` means "sort by inclusive user CPU time, whether or not it is visible".

TABLE 5-4 lists the available `er_print` metric name strings for clock-based metrics, synchronization delay metrics, and the two common hardware counter metrics. For other hardware counter metrics, the metric name string is the same as the counter name. A list of counter names can be obtained by using the `collect` command with no arguments. See "Hardware-Counter Overflow Data" on page 47 for more information on hardware counters.

TABLE 5-4 Metric Name Strings

String	Description
`user`	User CPU time
`wall`	Wall-clock time
`total`	Total LWP time
`system`	System CPU time
`wait`	CPU wait time
`text`	Text-page fault time
`data`	Data-page fault time
`owait`	Other wait time
`sync`	Synchronization wait time
`syncn`	Synchronization wait count
`cycles`	CPU cycles, counted on register 0
`cycles1`	CPU cycles, counted on register 1
`insts`	Instructions issued, counted on register 0
`insts1`	Instructions issued, counted on register 1

Note – The meaning of the `wait` metric string has changed to mean CPU wait time, and there is a new string, `owait`, which includes all other system wait time.

In addition to the name strings listed in TABLE 5-4, there are two name strings that can only be used in default metrics lists. These are `hwc`, which matches any hardware counter name, and `any`, which matches any metric name string.

Function List Commands

The following commands control the display of function information.

`fsummary`

Write a summary metrics panel for each function in the function list. The number of panels written can be limited by using the `limit` command (see "Output Commands" on page 119).

For a description of the summary metrics for a function, see "Viewing Summary Metrics for a Function or a Load Object" on page 90.

`functions`

Write the function list with the currently selected metrics. The number of lines written can be limited by using the `limit` command (see "Output Commands" on page 119).

The default metrics printed are exclusive and inclusive user CPU time, in both seconds and percentage of total program metric. You can change the current metrics displayed with the `metrics` command. This must be done before you issue the `functions` command. You can also change the defaults with the `dmetrics` command.

`metrics` *metric-list*

Specify a selection of function-list metrics. The string *metric-list* can either be the keyword `default`, which restores the default metric selection, or a list of metric keywords, separated by colons. The following example illustrates a metric list.

```
% metrics i.user:i%user:e.user:e%user
```

This command instructs `er_print` to display the following metrics:

- Inclusive user CPU time in seconds
- Inclusive user CPU time percentage
- Exclusive user CPU time in seconds
- Exclusive user CPU time percentage

When the `metrics` command is finished, a message is printed showing the current metric selection. For the preceding example the message is as follows.

```
current: i.user:i%user:e.user:e%user:name
```

For information on the syntax of metric lists, see “Metric Lists” on page 106. To see a listing of the available metrics, use the `metric_list` command.

If a `metrics` command has an error in it, it is ignored with a warning, and the previous settings remain in effect.

`objects`

Write the load-object list with the currently selected metrics. The number of lines written can be limited by using the `limit` command (see “Output Commands” on page 119).

Default is exclusive and inclusive user CPU time, in seconds and percentage of total program metric. You can change the current metrics displayed with the `metrics` command.

`osummary`

Write a summary metrics panel for each load object in the load-object list. The number of panels written can be limited by using the `limit` command (see “Output Commands” on page 119).

For a description of the summary metrics for a load object, see “Viewing Summary Metrics for a Function or a Load Object” on page 90.

`sort` *metric-keyword*

Sort the function list on the specified metric. The string *metric-keyword* is one of the metric keywords described in “Metric Lists” on page 106, as shown in this example.

```
% sort i.user
```

This command tells er_print to sort the function list by inclusive user CPU time. If the metric is not in the experiments that have been loaded, a warning is printed and the command is ignored. When the command is finished, the sort metric is printed.

Callers-Callees List Commands

The following commands control the display of caller and callee information.

callers-callees

Print the callers-callees panel for each of the functions, in the order in which they are sorted. The number of panels written can be limited by using the limit command (see "Output Commands" on page 119). The selected (middle) function is marked with an asterisk, as shown in this example.

```
Excl. User   Incl. User   Attr. User    Name
CPU sec.     CPU sec.     CPU sec.
0.           0.010        0.010         _doprnt
0.           0.           0.            _xflsbuf
0.           0.010        0.           *_realbufend
0.           0.620        0.            _rw_rdlock
0.           0.010        0.010         rw_unlock
```

In this example, _realbufend is the selected function; it is called by _doprnt and _xflsbuf, and it calls _rw_rdlock and rw_unlock.

cmetrics *metric-list*

Specify a selection of callers-callees metrics. *metric-list* is a list of metric keywords, separated by colons, as shown in this example.

```
% cmetrics i.user:i%user:a.user:a%user
```

This command instructs er_print to display the following metrics.

- Inclusive user CPU time in seconds
- Inclusive user CPU time percentage
- Attributed user CPU time in seconds
- Attributed user CPU time percentage

When the `cmetrics` command is finished, a message is printed showing the current metric selection. For the preceding example the message is as follows.

```
current: i.user:i%user:a.user:a%user:name
```

For information on the syntax of metric lists, see “Metric Lists” on page 106. To see a listing of the available metrics, use the `cmetric_list` command.

`csort` *metric-keyword*

Sort the callers-callees display on the specified metric. The string *metric-keyword* is one of the metric keywords described in “Metric Lists” on page 106, as shown in this example.

```
% csort a.user
```

This command tells `er_print` to sort the callers-callees display by attributed user CPU time. When the command finishes, the sort metric is printed.

Source and Disassembly Listing Commands

The following commands control the display of annotated source and disassembly code.

`source | src {` *file* `|` *function* `} [`*N*`]`

Write out annotated source code for either the specified file or the file containing the specified function. The file in either case must be in a directory in your path.

Use the optional parameter *N* (a positive integer) only in those cases where the file or function name is ambiguous; in this case, the *N*th possible choice is used. If you give an ambiguous name without the numeric specifier, `er_print` prints a list of possible object-file names; if the name you gave was a function, the name of the function is appended to the object-file name, and the number that represents the value of *N* for that object file is also printed.

`disasm { file | function } [N]`

Write out annotated disassembly code for either the specified file, or the file containing the specified function. The file in either case must be in a directory in your path.

The optional parameter *N* is used in the same way as for the `source` command.

`scc` *class-list*

Specify the classes of compiler commentary that are shown in the annotated source listing. The class list is a colon-separated list of classes, containing zero or more of the following message classes, and can also contain the highlighting threshold.

- `b[asic]`– Show the basic level messages.
- `v[ersion]` – Show version messages, including source file name and last modified date, versions of the compiler components, compilation date and options.
- `pa[rallel]` – Show messages about parallelization.
- `q[uery]` – Show questions about the code that affect its optimization.
- `l[oop]` – Show messages about loop optimizations and transformations.
- `pi[pe]` – Show messages about pipelining of loops.
- `i[nline]` – Show messages about inlining of functions.
- `m[emops]` – Show messages about memory operations, such as load, store, prefetch.
- `f[e]` – Show front-end messages.
- `all` – Show all messages.
- `none` – Do not show any messages.
- `t[hreshold]=`*nn* – Set the threshold for highlighting a line of source code or disassembly code. If the value of any metric is equal to or greater than *nn*% of the maximum value of that metric for any source line or disassembly instruction within the file, the line is highlighted.

The classes `all` and `none` cannot be used with other classes.

If no `scc` command is given, the default class shown is `basic`. If the `scc` command is given with an empty *class-list*, compiler commentary is turned off. The `scc` command is normally used only in a `.er.rc` file.

`dcc` *class-list*

Specify the classes of compiler commentary that are shown in the annotated disassembly listing. The class list is a colon-separated list of classes. The list of available classes is the same as the list of classes for annotated source code listing, with the addition of the following class.

- `h[ex]` – Show the hexadecimal value of the instructions.

Filtering Commands

This section describes commands that are used to control selection of experiments, samples, threads, and LWPs for display, and to list the current selections.

Selection Lists

The syntax of a selection is shown in the following example. This syntax is used in the command descriptions.

```
[experiment-list:]selection-list[+[experiment-list:]selection-list … ]
```

Each selection list can be preceded by an experiment list, separated from it by a colon and no spaces. You can make multiple selections by joining selection lists with a + sign.

The experiment list and the selection list have the same syntax, which is either the keyword `all` or a list of numbers or ranges of numbers (*n-m*) separated by commas but no spaces, as shown in this example.

```
2,4,9-11,23-32,38,40
```

The experiment numbers can be determined by using the `exp_list` command.

Some examples of selections are as follows.

```
1:1-4+2:5,6
all:1,3-6
```

In the first example, objects 1 through 4 are selected from experiment 1 and objects 5 and 6 are selected from experiment 2. In the second example, objects 1 and 3 through 6 are selected from all experiments. The objects may be LWPs, threads, or samples.

Selection Commands

The commands to select LWPs, samples, and threads are not independent. If the experiment list for a command is different from that for the previous command, the experiment list from the latest command is applied to all three selection targets – LWPs, samples, and threads, in the following way.

- Existing selections for experiments not in the latest experiment list are turned off.
- Existing selections for experiments in the latest experiment list are kept.
- Selections are set to "`all`" for targets for which no selection has been made.

`lwp_select` *lwp-selection*

Select the LWPs about which you want to display information. The list of LWPs you selected is displayed when the command completes.

`sample_select` *sample-selection*

Select the samples about which you want to display information. The list of samples you selected is displayed when the command completes.

`thread_select` *thread-selection*

Select the threads about which you want to display information. The list of threads you selected is displayed when the command completes.

`object_select` *object-list*

Select the load objects about which you want to display information. *object-list* is a list of load objects, separated by commas but no spaces. If an object name itself contains a comma, you must surround the comma with double quotation marks.

The names of the load objects should be either full path names or the basename.

Listing of Selections

The commands for listing what has been selected are given in this section, followed by some examples.

exp_list

Display the full list of experiments loaded with their ID number.

lwp_list

Display the list of LWPs currently selected for analysis.

object_list

Display the list of load objects currently selected for analysis.

sample_list

Display the list of samples currently selected for analysis.

thread_list

Display the list of threads currently selected for analysis.

The following example is an example of an experiment list.

```
(er_print) exp_list
ID Experiment
== ==========
1 test.1.er
2 test.6.er
```

The sample list, thread list and LWP list have the same format. The following example is an example of a sample list.

```
(er_print) sample_list
Exp Sel     Total
=== ======= =====
  1 1-6        31
  2 7-10,15    31
```

The following example is an example of a load object list.

```
(er_print) object_list
Sel Load Object
=== ==================================================
yes /tmp/var/synprog/synprog
yes /opt/SUNWspro/WS6U2/lib/dbxruntime/libcollector.so
yes /usr/lib/libdl.so.1
yes /usr/lib/libc.so.1
```

Metric List Commands

The following commands list the currently selected metrics and all available metric keywords.

metric_list

Display the currently selected metrics in the function list and a list of metric keywords that you can use in other commands (for example, `metrics` and `sort`) to reference various types of metrics in the function list.

cmetric_list

Display the currently selected metrics in the callers-callees list and a list of metric keywords that you can use in other commands (for example, `cmetrics` and `csort`) to reference various types of metrics in the callers-callees list.

Note – Attributed metrics can only be specified for display with the `cmetrics` command, not the `metrics` command, and displayed only with the `callers-callees` command, not the `functions` command.

Defaults Commands

The following commands can be used to set the defaults for `er_print` and for the Performance Analyzer. They can only be used for setting defaults: they cannot be used in input for `er_print`. They can be included in a defaults filed named `.er.rc`.

A defaults file can be included in your home directory, to set defaults for all experiments, or in any other directory, to set defaults locally. When `er_print`, `er_src` or the Performance Analyzer is started, the current directory and your home directory are scanned for defaults files, which are read if they are present, and the system defaults file is also read. Defaults from the `.er.rc` file in your home directory override the system defaults, and defaults from the `.er.rc` file in the current directory override both home and system defaults.

Note – To ensure that you read the defaults file from the directory where your experiment is stored, you must start the Performance Analyzer or the `er_print` utility from that directory.

The defaults file can also include the `scc` and `dcc` commands. Multiple `dmetrics` and `dsort` commands can be given in a defaults file, and the commands within a file are concatenated.

`dmetrics` *metric-list*

Specify the default metrics to be displayed or printed in the function list. The syntax and use of the metric list is described in the section "Metric Lists" on page 106. The order of the metric keywords in the list determines the order in which the metrics are presented and the order in which they appear in the Metric chooser in the Performance Analyzer.

Default metrics for the Callers-Callees list are derived from the function list default metrics by adding the corresponding attributed metric before the first occurrence of each metric name in the list.

`dsort` *metric-list*

Specify the default metric on which the function list is sorted. The sort metric is the first metric in this list that matches a metric in any loaded experiment. The syntax and use of the metric list is described in the section "Metric Lists" on page 106.

The default sort metric for the Callers-Callees list is the attributed metric corresponding to the first metric in the `dsort` metric list that matches a metric in any loaded experiment.

`gdemangle` *library-name*

Set the path to the shared object that supports an API to demangle C++ function names. The shared object must export the C function `cplus_demangle()`, conforming to the GNU standard `libiberty.so` interface.

Output Commands

The following commands control `er_print` display output.

`limit` *n*

Limit output to the first *n* entries of the report; *n* is an unsigned positive integer.

`name { long | short }`

Specify whether to use the long or the short form of function names (C++ only).

`outfile {` *filename* `| - }`

Close any open output file, then open *filename* for subsequent output. If you specify a dash (-) instead of *filename*, output is written to standard output.

Other Display Commands

address_space *experiment-ID*

Display address-space data for the specified experiment. The *experiment-ID* can be obtained from the `exp_list` command. If the *experiment-ID* is not given, the data is displayed for the first experiment loaded.

header *experiment-ID*

Display descriptive information about the specified experiment. The *experiment-ID* can be obtained from the `exp_list` command. If the *experiment-ID* is not given, the information is displayed for the first experiment loaded.

overview *experiment-ID*

Write out the overview data of each of the currently selected samples for the specified experiment. The *experiment-ID* can be obtained from the `exp_list` command. If the *experiment-ID* is not given, the data is displayed for the first experiment loaded.

statistics *experiment-ID*

Write out execution statistics, aggregated over the current sample set for the specified experiment. The *experiment-ID* can be obtained from the `exp_list` command. If the *experiment-ID* is not given, the data is displayed for the first experiment loaded.

Mapfile Generation Command

mapfile *load-object* { *mapfilename* | - }

Write a mapfile for the specified load object to the file *mapfilename*. If you specify a dash (-) instead of *mapfilename*, `er_print` writes the mapfile to standard output.

Control Commands

`quit`

Terminate processing of the current script, or exit interactive mode.

`script` *script*

Process additional commands from the script file *script*.

Information Commands

`help`

Print a list of `er_print` commands.

`{ Version | version }`

Print the current release number of `er_print`.

CHAPTER 6

Understanding the Performance Analyzer and Its Data

The Performance Analyzer reads the event data that is collected by the Sampling Collector and converts it into performance metrics. The metrics are computed for various elements in the structure of the target program, such as instructions, source lines, functions, and load objects. In addition to a header, the data recorded for each event collected has two parts:

- Some event-specific data that is used to compute metrics
- A call stack of the application that is used to associate those metrics with the program structure

The process of associating the metrics with the program structure is not always straightforward, due to the insertions, transformations, and optimizations made by the compiler. This chapter describes the process in some detail and discusses the effect on what you see in the Performance Analyzer displays.

This chapter covers the following topics.

- Interpreting Performance Metrics
- Call Stacks and Program Execution
- Mapping Addresses to Program Structure
- Annotated Code Listings

Interpreting Performance Metrics

The data for each event contains a high-resolution timestamp, a thread ID, and an LWP ID. These three can be used to filter the metrics in the Performance Analyzer by time, thread or LWP. In addition, each event generates specific raw data, which is described in the following sections. Each section also contains a discussion of the accuracy of the metrics derived from the raw data and the effect of data collection on the metrics.

Clock-Based Profiling

The event-specific data for clock-based profiling consists of an array of profiling interval counts for each of the ten microstates maintained by the kernel for each LWP. At the end of the profiling interval, the count for the microstate of each LWP is incremented by 1, and a profiling signal is scheduled. The array is only recorded and reset when the LWP is in user mode in the CPU. If the LWP is in user mode when the profiling signal is scheduled, the array element for the User-CPU state is 1, and the array elements for all the other states are 0. If the LWP is not in user mode, the data is recorded when the LWP next enters user mode, and the array can contain an accumulation of counts for various states.

The call stack is recorded at the same time as the data. If the LWP is not in user mode at the end of the profiling interval, the call stack cannot change until the LWP enters user mode again. Thus the call stack always accurately records the position of the program counter at the end of each profiling interval.

The metrics to which each of the microstates contributes are shown in TABLE 6-1.

TABLE 6-1 How Kernel Microstates Contribute to Metrics

Kernel Microstate	Description	Metric Name
LMS_USER	Running in user mode	User CPU Time
LMS_SYSTEM	Running in system call or page fault	System CPU Time
LMS_TRAP	Running in any other trap	System CPU Time
LMS_TFAULT	Asleep in user text page fault	Text Page Fault Time
LMS_DFAULT	Asleep in user data page fault	Data Page Fault Time
LMS_KFAULT	Asleep in kernel page fault	Other Wait Time
LMS_USER_LOCK	Asleep waiting for user-mode lock	Other Wait Time
LMS_SLEEP	Asleep for any other reason	Other Wait Time
LMS_STOPPED	Stopped (/proc, job control, or lwp_stop)	Other Wait Time
LMS_WAIT_CPU	Waiting for CPU	Wait CPU Time

Accuracy of Timing Metrics

Timing data is collected on a statistical basis, and is therefore subject to all the errors of any statistical sampling method. For very short runs, in which only a small number of profile packets is recorded, the call stacks might not represent the parts of the program which consume the most resources. You should run your program long enough to accumulate hundreds of profile packets for any function or source line you are interested in.

In addition to statistical sampling errors, there are specific errors that arise from the way the data is collected and attributed and the way the program progresses through the system. Some of the circumstances in which inaccuracies or distortions can appear in the timing metrics are described in what follows.

- When an LWP is created, the time it has spent before the first profile packet is recorded is less than the profiling interval, but the entire profiling interval is ascribed to the microstate recorded in the first profile packet. It is likely that the LWP is in the System-CPU microstate when the first profile packet is recorded, and the System CPU Time metric is overaccounted. If there are many LWPs created the error can be many times the profiling interval.
- When an LWP is destroyed, some time is spent after the last profile packet is recorded. This time is often spent in the User-CPU microstate, and therefore the User CPU Time metric is underaccounted. If there are many LWPs destroyed the error can be many times the profiling interval.
- LWP scheduling is done on a smaller time scale than the profiling interval. As a consequence, the recorded state of the LWP might not represent the microstate in which it spent most of the profiling interval. The errors are likely to be larger when there are more LWPs to run than there are processors to run them.
- It is possible for a program to behave in a way which is correlated with the system clock. In this case, the profiling interval always expires when the LWP is in a state which might represent a small fraction of the time spent, and the call stacks recorded for a particular part of the program are overrepresented. On a multiprocessor system, it is possible for the profiling signal to induce a correlation: processors that are interrupted by the profiling signal while they are running LWPs for the program are likely to be in the Trap-CPU microstate when the microstate is recorded.
- The kernel records the microstate value when the profiling interval expires. When the system is under heavy load, that value might not represent the true state of the process. This situation is likely to result in overaccounting of the Trap-CPU or Wait-CPU microstate.
- The threads library sometimes discards profiling signals when it is in a critical section, resulting in an underaccounting of timing metrics.
- Because the time interval between the execution of instructions is much smaller than the profiling interval, there can be an error in the attribution of the time to the instruction.

In addition to the inaccuracies just described, timing metrics are distorted by the process of collecting data. The time spent recording profile packets never appears in the metrics for the program, because the recording is initiated by profiling signal. (This is another instance of correlation.) The user CPU time spent in the recording process is distributed over whatever microstates are recorded. The result is an underaccounting of the User CPU Time metric and an overaccounting of other metrics. The amount of time spent recording data is typically less than one percent of the CPU time for the default profiling interval.

Comparisons of Timing Metrics

If you compare timing metrics obtained from the profiling done in a clock-based experiment with times obtained by other means, you should be aware of the following issues.

For a single-threaded application, the total LWP time recorded for a process is usually accurate to a few tenths of a percent, compared with the values returned by `gethrtime`(3C) for the same process. The CPU time can vary by several percentage points from the values returned by `gethrvtime`(3C) for the same process. Under heavy load, the variation might be even more pronounced. However, the CPU time differences do not represent a systematic distortion, and the relative times reported for different routines, source-lines, and such are not substantially distorted.

For multithreaded applications using unbound threads, differences in values returned by `gethrvtime()` could be meaningless. This is because `gethrvtime()` returns values for an LWP, and a thread can change from one LWP to another.

The LWP times that are reported in the Performance Analyzer can differ substantially from the times that are reported by `vmstat`, because `vmstat` reports times that are summed over CPUs. If the target process has more LWPs than the system on which it is running has CPUs, the Performance Analyzer shows more wait time than `vmstat` reports.

The microstate timings which appear in the Execution Statistics window of the Performance Analyzer are based on process file system usage reports, for which the times spent in the microstates are recorded to high accuracy. See the `proc`(4) man page for more information. You can compare these timings with the metrics for the `<Total>` function, which represents the program as a whole, to gain an indication of the accuracy of the aggregated timing metrics.

Synchronization Wait Tracing

The Sampling Collector collects synchronization delay events by tracing calls to the functions in the threads library, `libthread.so`, or calls to MPI blocking routines. The event-specific data consists of high-resolution timestamps for the request and the grant (beginning and end of the call that is traced), and the address of the synchronization object (the mutex lock being requested, for example). The wait time is the difference between the request time and the grant time. Only events for which the wait time exceeds the specified threshold are recorded. The synchronization wait tracing data is recorded in the experiment at the time of the grant.

If the program uses bound threads, the LWP on which the waiting thread is scheduled cannot perform any other work until the event that caused the delay is completed. The time spent waiting appears both as Synchronization Wait Time and as Other Wait Time.

If the program uses unbound threads, it is possible for the LWP on which the waiting thread is scheduled to have other threads scheduled on it and continue to perform user work. The Other Wait Time is zero if all LWPs are kept busy while some threads are waiting for a synchronization event. However, the Synchronization Wait time is not zero because it is associated with a particular thread, not with the LWP on which the thread is running.

The wait time is distorted by the overhead for data collection. The overhead is proportional to the number of events collected. The fraction of the wait time spent in overhead can be minimized by increasing the threshold for recording events.

Hardware-Counter Overflow Profiling

Hardware-counter overflow profiling data includes a counter ID and the overflow value. The value can be larger than the value at which the counter is set to overflow, because the processor executes some instructions between the overflow and the recording of the event. This is especially true of cycle and instruction counters, which are incremented much more frequently than counters such as floating-point operations or cache misses. The delay in recording the event also means that the program counter address recorded with call stack does not correspond exactly to the overflow event. See "Attribution of Hardware Counter Overflows" on page 146 for more information.

The amount of data collected depends on the overflow value. Choosing a value that is too small can have the following consequences.

- The amount of time spent in the collection process can be a substantial fraction of the execution time of the program. The collection run might spend most of its time handling overflows and writing data instead of running the program.
- A substantial fraction of the counts can come from the collection process. These counts are attributed to the collector function `collector_record_counters`. If you see high counts for this function, the overflow value is too small.
- The collection process can alter the behavior of the program. For example, if you are collecting data on cache misses, the majority of the misses could come from flushing the collector instructions and profiling data from the cache and replacing it with the program instructions and data. The program would appear to have a lot of cache misses, but without data collection there might in fact be very few cache misses.

Choosing a value that is too large can result in too few overflows for good statistics. The counts that are accrued after the last overflow are attributed to the collector function `collector_record_counters`. If you see a substantial fraction of the counts in this function, the overflow value is too large.

Call Stacks and Program Execution

A call stack is a series of program counter addresses (PCs) representing instructions from within the program. The first PC, called the leaf PC, is at the bottom of the stack, and is the address of the next instruction to be executed. The next PC is the address of the call to the function containing the leaf PC; the next PC is the address of the call to that function, and so forth, until the top of the stack is reached. Each such address is known as a return address. The process of recording a call stack involves obtaining the return addresses from the program stack and is referred to as "unwinding the stack".

The leaf PC in a call stack is used to attribute exclusive metrics from the performance data to the function in which that PC is located. Each PC on the stack, including the leaf PC, is used to attribute inclusive metrics to the function in which it is located.

Most of the time, the PCs in the recorded call stack correspond in a natural way to functions as they appear in the source code of the program, and the Performance Analyzer's reported metrics correspond directly to those functions. Sometimes, however, the actual execution of the program does not correspond to a simple intuitive model of how the program would execute, and the Performance Analyzer's reported metrics might be confusing. See "Mapping Addresses to Program Structure" on page 136 for more information about such cases.

Single-Threaded Execution and Function Calls

The simplest case of program execution is that of a single-threaded program calling functions within its own load object.

When a program is loaded into memory to begin execution, a context is established for it that includes the initial address to be executed, an initial register set, and a stack (a region of memory used for scratch data and for keeping track of how functions call each other). The initial address is always at the beginning of the function `_start()`, which is built into every executable.

When the program runs, instructions are executed in sequence until a branch instruction is encountered, which among other things could represent a function call or a conditional statement. At the branch point, control is transferred to the address given by the target of the branch, and execution proceeds from there. (Usually the next instruction after the branch is already committed for execution: this instruction is called the branch delay slot instruction. However, some branch instructions inhibit the execution of the branch delay slot instruction.)

When the instruction sequence that represents a call is executed, the return address is put into a register, and execution proceeds at the first instruction of the function being called.

In most cases, somewhere in the first few instructions of the called function, a new frame (a region of memory used to store information about the function) is pushed onto the stack, and the return address is put into that frame. The register used for the return address can then be used when the called function itself calls another function. When the function is about to return, it pops its frame from the stack, and control returns to the address from which the function was called.

Function Calls Between Shared Objects

When a function in one shared object calls a function in another shared object, the execution is more complicated than in a simple call to a function within the program. Each shared object contains a Program Linkage Table, or PLT, which contains entries for every function external to that shared object that is referenced from it. Initially the address for each external function in the PLT is actually an address within `ld.so`, the dynamic linker. The first time such a function is called, control is transferred to the dynamic linker, which resolves the call to the real external function and patches the PLT address for subsequent calls. PLT addresses can appear in a call stack; see “The `<Unknown>` Function” on page 141.

Signals

When a signal is sent to a process, various register and stack operations occur that make it look as though the leaf PC at the time of the signal is the return address for a call to a system routine, `sigacthandler()`. `sigacthandler()` calls the user-specified signal handler just as any function would call another.

The Performance Analyzer treats the frames resulting from signal delivery as ordinary frames. The user code at the point at which the signal was delivered is shown as calling the system routine `sigacthandler()`, and it in turn is shown as calling the user’s signal handler. Inclusive metrics from both `sigacthandler()` and any user signal handler, and any other functions they call, appear as inclusive metrics for the interrupted routine.

Traps

Traps can be issued by an instruction or by the hardware, and are caught by a trap handler. System traps are traps which are initiated from an instruction and trap into the kernel. All system calls are implemented using trap instructions, for example. Some examples of hardware traps are those issued from the floating point unit when

it is unable to complete an instruction (such as the `fitos` instruction on the UltraSPARC III platform), or when the instruction is not implemented in the hardware.

When a trap is issued, the LWP enters system mode. The microstate is usually switched from User state to Trap state then to System state. The time spent handling the trap can show as a combination of System CPU time and User CPU time, depending on the point at which the microstate is switched. The time is attributed to the instruction in the user's code from which the trap was initiated (or to the system call).

For some system calls, it is considered critical to provide as efficient handling of the call as possible. The traps generated by these calls are known as *fast traps*. Among the system routines which generate fast traps are `gethrtime` and `gethrvtime`. In these routines, the microstate is not switched because of the overhead involved.

In other circumstances it is also considered critical to provide as efficient handling of the trap as possible. Some examples of these are TLB (translation lookaside buffer) misses and register window spills and fills, for which the microstate is also not switched.

In both cases, the time spent is recorded as User CPU time. However, the hardware counters are turned off because the mode has been switched to system mode. The time spent handling these traps can therefore be estimated by taking the difference between User CPU time in a clock-based experiment and Cycles time in a hardware-counter experiment.

There is one case in which the trap handler switches back to user mode, and that is the misaligned memory reference trap for an 8-byte integer which is aligned on a 4-byte boundary in Fortran. A frame for the trap handler appears on the stack, and a call to the handler can appear in the Performance Analyzer, attributed to the integer load or store instruction.

When an instruction traps into the kernel, the instruction following the trapping instruction appears to take a long time, because it cannot start until the kernel has finished executing the trapping instruction.

Tail-Call Optimization

The compiler can do one particular optimization whenever the last thing a particular routine does is to call another routine. Rather than generating a new frame, the callee re-uses the frame from the caller, and the return address for the callee is copied from the caller. The motivation for this optimization is to reduce the size of the stack, and, on SPARC platforms, to reduce the use of register windows.

Suppose that the call sequence in your program source looks like this:

```
A -> B -> C -> D
```

When B and C are tail-call optimized, the call stack looks as if routine A calls routines B, C, and D directly.

```
A -> B
A -> C
A -> D
```

That is, the call tree is flattened. When code is compiled with the -g option, tail-call optimization takes place only at a compiler optimization level of 4 or higher. When code is compiled without the -g option, tail-call optimization takes place at a compiler optimization level of 2 or higher.

Explicit Multithreading

A simple program executes in a single thread, on a single LWP (light-weight process). Multithreaded executables make calls to a thread creation routine, to which the target function for execution is passed. When the target exits, the thread is destroyed by the threads library. Newly-created threads begin execution at a routine called `_thread_start()`, which calls the function passed in the thread creation call. For any call stack involving the target as executed by this thread, the top of the stack is `_thread_start()`, and there is no connection to the caller of the thread creation routine. Inclusive metrics associated with the created thread will therefore only propagate up as far as `_thread_start()` and the `<Total>` function.

In addition to creating the threads, the threads library also creates LWPs to execute the threads. Threading can be done either with bound threads, where each thread is bound to a specific LWP, or with unbound threads, where each thread can be scheduled on a different LWP at different times.

- If bound threads are used, the threads library creates one LWP per thread.
- If unbound threads are used, the threads library decides how many LWPs to create to run efficiently, and which LWPs to schedule the threads on. It can create more LWPs at a later time if they are needed.

The operating system controls the assignment of LWPs to CPUs for execution, while the threads library controls the scheduling of threads onto LWPs. For example, when a thread is at a synchronization barrier such as a `mutex_lock`, the threads library can schedule a different thread on the LWP on which the first thread was executing. The time spent waiting for the lock by the thread that is at the barrier appears in the Synchronization Wait Time metric, but since the LWP is not idle, the time will not be accrued into the Other Wait Time metric.

Parallel Execution and Compiler-Generated Body Functions

If your code contains Sun, Cray, or OpenMP parallelization directives, it can be compiled for parallel execution. (OpenMP is a feature available with the Sun WorkShop 6 update 2 Fortran 95 and C compilers. Refer to the chapters on parallelization and OpenMP in the *Fortran Programming Guide* and *C User's Guide* for background on parallelization strategies and OpenMP directives, or visit the web site defining the OpenMP standard, `http://www.openmp.org`)

When a loop or other parallel construct is compiled for parallel execution, the compiler-generated code is executed by multiple threads, coordinated by the microtasking library. Parallelization by the Forte Developer compilers follows the procedure outlined below.

Generation of Body Functions

When the compiler encounters a parallel construct, it sets up the code for parallel execution by placing the body of the construct in a separate *body function* and replacing the construct with a call to a microtasking library routine. The microtasking library routine is responsible for dispatching threads to execute the body function. The address of the body function is passed to the microtasking library routine as an argument.

- If the parallel construct is delimited with one of the following:
  - The Sun Fortran directive `c$par doall`
  - The Cray Fortran directive `c$mic doall`
  - A Fortran OpenMP `c$omp PARALLEL`, `c$omp PARALLEL DO`, or `c$omp PARALLEL SECTIONS` directive
  - A C OpenMP `#pragma omp parallel`, `#pragma omp parallel for`, or `#pragma omp parallel sections` directive

  then the construct is replaced with a call to the microtasking library routine `__mt_MasterFunction_()`. A loop that is parallelized automatically by the compiler is also replaced by a call to `__mt_MasterFunction_()`.
- If a `c$omp PARALLEL` construct contains one or more worksharing `c$omp DO` or `c$omp SECTIONS` directives, each worksharing construct is replaced by a call to the microtasking library routine `__mt_Worksharing_()` and a new body function is created for each.

The compiler assigns names to body functions in the form shown here.

```
_$1$mf_string1_$namelength$functionname$linenumber$string2
```

- *string1* denotes the type of parallel construct.
- *namelength* is the number of characters in *functionname.*
- *functionname* is the name of the function from which the construct was extracted, ending in an underscore for Fortran functions.
- *linenumber* is the line number of the beginning of the construct in the original source.
- *string2* is related to the name of the source file.

To make the data easier to analyze, the Performance Analyzer provides these functions with a more readable name, in addition to the compiler-generated name.

Parallel Execution Sequence

The program begins execution with only one thread, the main thread. The first time the program calls `__mt_MasterFunction_()`, this function calls the Solaris threads library routine, `thr_create()` to create worker threads. Each worker thread executes the microtasking library routine `__mt_SlaveFunction_()`, which was passed as an argument to `thr_create()`.

Once the threads have been created, `__mt_MasterFunction_()` manages the distribution of available work among the main thread and the worker threads. If work is not available, `__mt_SlaveFunction_()` calls `__mt_WaitForWork_()`, in which the worker thread waits for available work. As soon as work becomes available, the thread returns to `__mt_SlaveFunction_()`.

When work is available, each thread executes a call to `__mt_run_my_job_()`, to which information about the body function is passed. The sequence of execution from this point depends on whether the body function was generated from a parallel sections directive, a parallel do (or parallel for) directive, or a parallel directive.

- In the parallel sections case, `__mt_run_my_job_()` calls the body function directly.
- In the parallel do case, `__mt_run_my_job_()` calls other routines, which depend on the nature of the loop, and the other routines call the body function.
- In the parallel case, `__mt_run_my_job_()` calls the body function, and all threads execute the code in the body function until they encounter a call to `__mt_WorkSharing_()`. In this routine there is another call to `__mt_run_my_job_()`, which calls the worksharing body function, either directly in the case of a worksharing section, or via the loop-dependent routines in the case of a worksharing do or for. If `nowait` was specified in the worksharing directive, each thread returns to the parallel body function and continues executing. Otherwise, the threads return to `__mt_WorkSharing_()`, which calls `__mt_EndOfTaskBarrier_()` to synchronize the threads before continuing.

When the work is finished, the threads return to `__mt_MasterFunction_()` or `__mt_SlaveFunction_()` and call `__mt_EndOfTaskBarrier_()` to perform any synchronization work involved in the termination of the parallel construct. The worker threads then call `__mt_WaitForWork_()` again, while the main thread continues to execute in the serial region.

The call sequence described here applies not only to a program running in parallel, but also to a program compiled for parallelization but running on a single-CPU machine, or on a multiprocessor machine using only one LWP.

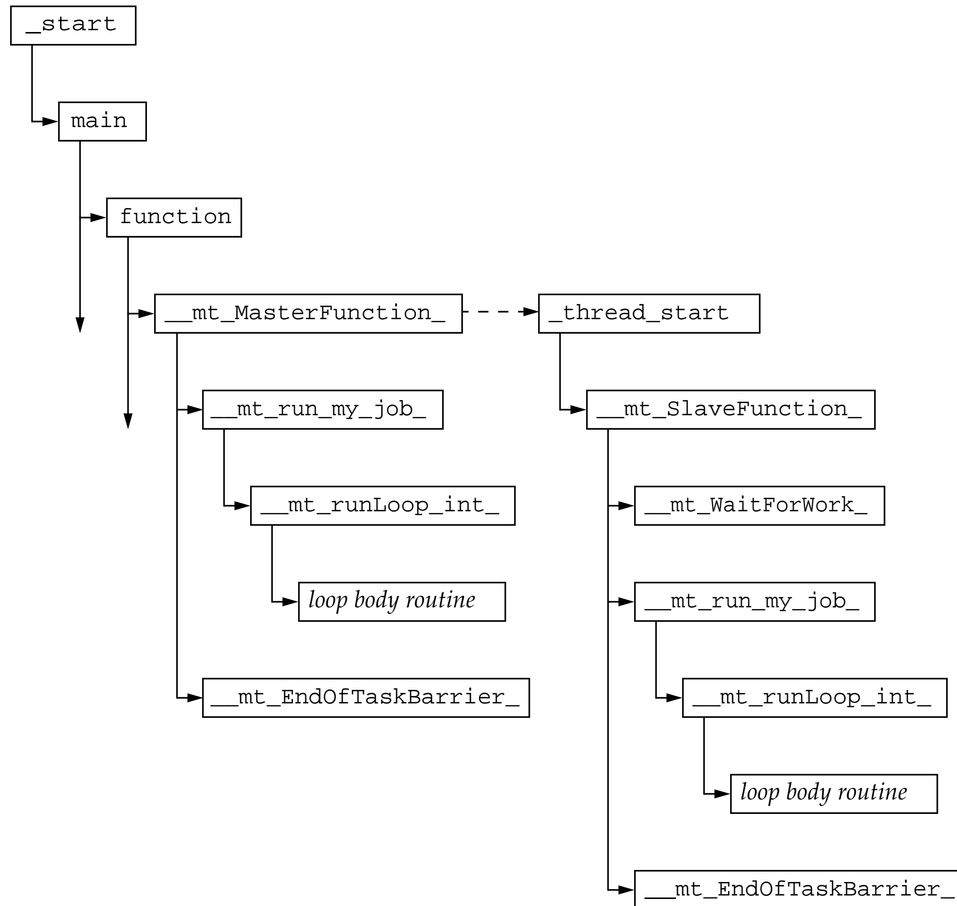


FIGURE 6-1 Schematic Call Tree for a Multithreaded Program That Contains a Parallel Do Construct

The call sequence for a simple parallel do construct is illustrated in FIGURE 6-1. The call stack for a worker thread begins with the threads library routine `_thread_start()`, the routine which actually calls `__mt_SlaveFunction_()`.

The dotted arrow indicates the initiation of the thread as a consequence of a call from `__mt_MasterFunction_()` to `thr_create()`. The continuing arrows indicate that there might be other function calls which are not represented here.

The call sequence for a parallel region in which there is a worksharing do construct is illustrated in FIGURE 6-2. The caller of `__mt_run_my_job_()` is either `__mt_MasterFunction_()` or `__mt_SlaveFunction_()`. The entire diagram can replace the call to `__mt_run_my_job_()` in FIGURE 6-1.

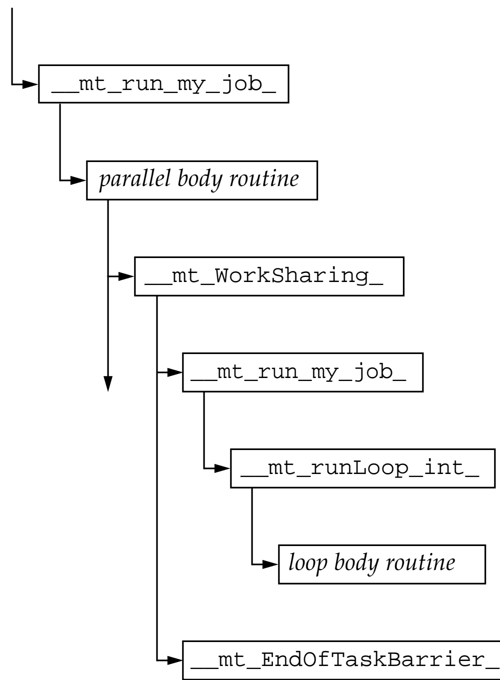


FIGURE 6-2 Schematic Call Tree for a Parallel Region With a Worksharing Do Construct

In these call sequences, all the compiler-generated body functions are called from the same routine (or routines) in the microtasking library, which makes it difficult to associate the metrics from the body function with the original function. The Performance Analyzer inserts an imputed call to the original function between the body function and the microtasking library function that calls it, and inserts an imputed call to the original function and the body function between the barrier function, `__mt_EndOfTaskBarrier_()`, and `__mt_MasterFunction_()`, `__mt_SlaveFunction_()` or `__mt_WorkSharing_()`. The metrics due to the synchronization are therefore attributed to the body routine, and the metrics for the body routine are attributed to the original function. With these insertions, inclusive metrics from the body function propagate directly to the original function rather than through the microtasking library routines. The side effect of these imputed calls is that the original function appears as a callee of the microtasking routines: these

calls should be ignored. Double-counting of inclusive metrics is avoided by the mechanism used for recursive function calls (see "How Recursion Affects Function-Level Metrics" on page 80).

Worker threads typically use CPU time while they are in `__mt_WaitForWork_()` in order to reduce latency when new work arrives, that is, when the main thread reaches a new parallel construct. This is known as a busy-wait. However, you can set an environment variable to specify a sleep wait, which shows up in the Performance Analyzer as Other Wait time instead of User CPU time. There are generally two situations where the worker threads spend time waiting for work, where you might want to redesign your program to reduce the waiting:

- When the main thread is executing in a serial region and there is nothing for the worker threads to do
- When the work load is unbalanced, and some threads have finished and are waiting while others are still executing

By default, the microtasking library uses threads that are bound to LWPs. You can override this default by setting the environment variable `MT_BIND_LWP` to `FALSE`.

Note – The multiprocessing dispatch process is implementation-dependent and might change from release to release.

Mapping Addresses to Program Structure

Once a call stack is processed into PC values, the Performance Analyzer maps those PCs to shared objects, functions, source lines, and disassembly lines (instructions) in the program. This section describes those mappings.

The Process Image

When a program is run, a process is instantiated from the executable for that program. The process has a number of regions in its address space, some of which are text and represent executable instructions, and some of which are data which is not normally executed. PCs as recorded in the call stack normally correspond to addresses within one of the text segments of the program.

The first text section in a process derives from the executable itself. Others correspond to shared objects that are loaded with the executable, either at the time the process is started, or dynamically loaded by the process. The PCs in a call stack are resolved based on the executable and shared objects loaded at the time the call stack was recorded. Executables and shared objects are very similar, and are collectively referred to as load objects.

Because shared objects can be loaded and unloaded in the course of program execution, any given PC might correspond to different functions at different times during the run. In addition, different PCs might correspond to the same function, when a shared object is unloaded and then reloaded at a different address.

Load Objects and Functions

Each load object, whether an executable or a shared object, contains a text section with the instructions generated by the compiler, a data section for data, and various symbol tables. All load objects must contain an ELF symbol table, which gives the names and addresses of all the globally-known functions in that object. Load objects compiled with the `-g` option contain additional symbolic information, which can augment the ELF symbol table and provide information about functions that are not global, additional information about object modules from which the functions came, and line number information relating addresses to source lines.

The term *function* is used to describe a set of instructions that represent a high-level operation described in the source code. The term covers subroutines as used in Fortran, methods as used in C++, and the like. Functions are described cleanly in the source code, and normally their names appear in the symbol table representing a set of addresses; if the program counter is within that set, the program is executing within that function.

In principle, any address within the text segment of a load object can be mapped to a function. Exactly the same mapping is used for the leaf PC and all the other PCs on the call stack. Most of the functions correspond directly to the source model of the program. Some do not; these functions are described in the following sections.

Aliased Functions

Typically, functions are defined as global, meaning that their names are known everywhere in the program. The name of a global function must be unique within the executable. If there is more than one global function of a given name within the address space, the runtime linker resolves all references to one of them. The others are never executed, and so do not appear in the function list. From the Summary Metrics window, you can see the shared object and object module that contain the selected function.

Under various circumstances, a function can be known by several different names. A very common example of this is the use of so-called weak and strong symbols for the same piece of code. A strong name is usually the same as the corresponding weak name, except that it has a leading underscore. Many of the functions in the threads library also have alternate names for pthreads and Solaris threads, as well as strong and weak names and alternate internal symbols. In all such cases, only one name is used in the function list of the Performance Analyzer. The name chosen is the last symbol at the given address in alphabetic order. This choice most often corresponds to the name that the user would use. In the Summary Metrics window, all the aliases for the selected function are shown.

Non-Unique Function Names

While aliased functions reflect multiple names for the same piece of code, there are circumstances under which multiple pieces of code have the same name:

- Sometimes, for reasons of modularity, functions are defined as static, meaning that their names are known only in some parts of the program (usually a single compiled object module). In such cases, several functions of the same name referring to quite different parts of the program appear in the Performance Analyzer. In the Summary Metrics window, the object module name for each of these functions is given to distinguish them from one another. In addition, any selection of one of these functions can be used to show the source, disassembly, and the callers and callees of that specific function.
- Sometimes a program uses wrapper or interposition functions that have the weak name of a function in a library and supersede calls to that library function. Some wrapper functions call the original function in the library, in which case both instances of the name appear in the Performance Analyzer function list. Such functions come from different shared objects and different object modules, and can be distinguished from each other in that way. The Sampling Collector wraps some library functions, and both the wrapper function and the real function can appear in the Performance Analyzer.

Static Functions From Stripped Shared Libraries

Static functions are often used within libraries, so that the name used internally in a library does not conflict with a name that the user might use. When libraries are stripped, the names of static functions are deleted from the symbol table. In such cases, the Performance Analyzer generates an artificial name for each text region in the library containing stripped static functions. The name is of the form `<static>@0x12345`, where the string following the `@` sign is the offset of the text

region within the library. The Performance Analyzer cannot distinguish between contiguous stripped static functions and a single such function, so two or more such functions can appear with their metrics coalesced.

Stripped static functions are shown as called from the correct caller, except when the PC from the static function is a leaf PC that appears after the save instruction in the static function. Without the symbolic information, the Performance Analyzer does not know the save address, and cannot tell whether to use the return register as the caller. It always ignores the return register. Since several functions can be coalesced into a single `<static>@0x12345` function, the real caller or callee might not be distinguished from the adjacent routines.

Fortran Alternate Entry Points

Fortran provides a way of having multiple entry points to a single piece of code, allowing a caller to call into the middle of a function. When such code is compiled, it consists of a prologue for the main entry point, a prologue to the alternate entry point, and the main body of code for the function. Each prologue sets up the stack for the function's eventual return and then branches or falls through to the main body of code.

The prologue code for each entry point always corresponds to a region of text that has the name of that entry point, but the code for the main body of the routine receives only one of the possible entry point names. The name received varies from one compiler to another.

The prologues rarely account for any significant amount of time, and the "functions" corresponding to entry points other than the one that is associated with the main body of the subroutine rarely appear in the Performance Analyzer. Call stacks representing time in Fortran subroutines with alternate entry points usually have PCs in the main body of the subroutine, rather than the prologue, and only the name associated with the main body will appear as a callee. Likewise, all calls from the subroutine are shown as being made from the name associated with the main body of the subroutine.

Inlined Functions

An inlined function is a function for which the instructions generated by the compiler are inserted at the call site of the function instead of an actual call. There are two kinds of inlining, both of which are done to improve performance, and both of which affect the Performance Analyzer.

- C++ inline function definitions. The rationale for inlining in this case is that the cost of calling a function is much greater than the work done by the inlined function, so it is better to simply insert the code for the function at the call site, instead of setting up a call. Typically, access functions are defined to be inlined, because they often only require one instruction. When you compile with the `-g` option, inlining of functions is disabled; compilation with `-g0` permits inlining of functions.
- Explicit or automatic inlining performed by the compiler at high optimization levels (4 and 5). Explicit and automatic inlining is performed even when `-g` is turned on. The rationale for this type of inlining can be to save the cost of a function call, but more often it is to provide more instructions for which register usage and instruction scheduling can be optimized.

Both kinds of inlining have the same effect on the display of metrics. Functions that appear in the source code but have been inlined do not show up in the function list, nor do they appear as callees of the routines into which they have been inlined. Metrics that would otherwise appear as inclusive metrics at the call site of the inlined function, representing time spent in the called function, are actually shown as exclusive metrics attributed to the call site, representing the instructions of the inlined function.

Note – Inlining can make data difficult to interpret, so you might want to disable inlining when you compile your program for performance analysis.

In some cases, even when a function is inlined, a so-called out-of-line function is left. Some call sites call the out-of-line function, but others have the instructions inlined. In such cases, the function appears in the function list but the metrics attributed to it represent only the out-of-line calls.

Compiler-Generated Body Functions

When a compiler parallelizes a loop in a function, or a region that has parallelization directives, it creates new body functions that are not in the original source code. These functions are described in "Parallel Execution and Compiler-Generated Body Functions" on page 132.

The Performance Analyzer shows these functions as normal functions, and assigns a name to them based on the function from which they were extracted, in addition to the compiler-generated name. Their exclusive and inclusive metrics represent the time spent in the body function. In addition, the function from which the construct was extracted shows inclusive metrics from each of the body functions. The means by which this is achieved is described in "Parallel Execution Sequence" on page 133

When a function containing parallel loops is inlined, the names of its compiler-generated body functions reflect the function into which it was inlined, not the original function.

Outline Functions

Outline functions can be created during feedback optimization. They represent code that is not normally expected to be executed. Specifically, it is code that is not executed during the "training run" used to generate the feedback. To improve paging and instruction-cache behavior, such code is moved elsewhere in the address space, and is made into a function with a name in the form shown here.

```
_$1$outlinestring1$namelength$functionname$linenumber$string2
```

- *string1* is related to the specific section of outlined code.
- *namelength* is the number of characters in *functionname.*
- *functionname* is the name of the function from which the section was extracted.
- *linenumber* is the line number of the beginning of the section in the original source.
- *string2* is related to a compiler internal name.

Outline functions are not really called, but rather are jumped to; similarly they do not return, they jump back. In order to make the behavior more closely match the user's source code model, the Performance Analyzer imputes an artificial call from the main routine to its outline portion.

Outline functions are shown as normal functions, with the appropriate inclusive and exclusive metrics. In addition, the metrics for the outline function are added as inclusive metrics in the function from which the code was outlined.

The `<Unknown>` Function

Under some circumstances, a PC does not map to a known function. In such cases, the PC is mapped to the special function named `<Unknown>`.

The following circumstances will show PCs mapping to `<Unknown>`:

- When the PC corresponds to the PLT (Program Linkage Table) in a load object. This happens whenever a function in one load object calls a function in a different shared object. The actual call transfers first to a three-instruction sequence in the PLT, and then to the real destination.

- When the PC corresponds to an address in the data section of the executable or a shared object. Normally data is not executable, so a data address never appears in a call stack. Programs that are self-modifying or for which routines are dynamically compiled write instructions into the program data space before executing them. The SPARC V7 version of `libc.so` has several functions (`.mul` and `.div`, for example) in its data section. The code is in the data section so that it can be dynamically rewritten to use machine instructions when the library detects that it is executing on a SPARC V8 or V9 platform.
- When the PC is not within any known load object. The most likely cause of this is an unwind failure, where the value recorded as a PC is not a PC at all, but rather some other word. If the PC is the return register, and it does not seem to be within any known load object, it is ignored, rather than attributed to the `<Unknown>` function.

Callers and callees of the `<Unknown>` function represent the previous and next PCs in the call stack, and are treated normally.

The `<Total>` Function

The `<Total>` function is an artificial construct used to represent the program as a whole. All performance metrics, in addition to being attributed to the functions on the call stack, are attributed to the special function `<Total>`. It appears at the top of the function list and its data can be used to give perspective on the data for other functions. In the Callers-Callees list, it is shown as the nominal caller of `_start()` in the main thread of execution of any program, and also as the nominal caller of `_thread_start()` for created threads.

Annotated Code Listings

The annotated source code and annotated disassembly code features of the Performance Analyzer are useful for determining which source lines or instructions within a function are responsible for poor performance. This section describes the annotation process and some of the issues involved in interpreting the annotated code.

Annotated Source Code

Annotated source shows the resource consumption of an application at the source-line level. It is produced by taking the PCs that are recorded in the application's call stack, and mapping each PC to a source line. To produce an annotated source file, the Performance Analyzer first determines all of the functions that are generated in a particular object module (`.o` file) or load object, then scans the data for all PCs from each function. In order to produce annotated source, the Performance Analyzer must be able to find and read the object module or load object to determine the mapping from PCs to source lines, and it must be able to read the source file to produce an annotated copy, which is displayed.

The compilation process goes through many stages, depending on the level of optimization requested, and transformations take place which can confuse the mapping of instructions to source lines. For some optimizations, source line information might be completely lost, while for others, it might be confusing. The compiler relies on various heuristics to track the source line for an instruction, and these heuristics are not infallible.

The four possible formats for the metrics that can appear on a line of annotated source code are explained in TABLE 6-2.

TABLE 6-2 Annotated Source-Code Metrics

Metric	Significance
(Blank)	No PC in the program corresponds to this line of code. This case should always apply to comment lines, and applies to apparent code lines in the following circumstances: • All the instructions from the apparent piece of code have been eliminated during optimization. • The code is repeated elsewhere, and the compiler performed common subexpression recognition and tagged all the instructions with the lines for the other copy. • The compiler tagged an instruction with an incorrect line number.
`0.`	Some PCs in the program were tagged as derived from this line, but there was no data that referred to those PCs: they were never in a call stack that was sampled statistically or traced for thread-synchronization data. The `0.` metric does not indicate that the line was not executed, only that it did not show up statistically in a profile and that a thread-synchronization call from that line never had a delay that exceeded the threshold.
`0.000`	At least one PC from this line appeared in the data, but the computed metric value rounded to zero.
`1.234`	The metrics for all PCs attributed to this line added up to the non-zero numerical value shown.

Compiler Commentary

Various parts of the compiler can incorporate commentary into the executable. Each comment is associated with a specific line of source code. When the annotated source is written, the compiler commentary for any source line appears immediately preceding the source line.

The compiler commentary describes many of the transformations which have been made to the source code to optimize it. These transformations include loop optimizations, parallelization, inlining and pipelining.

The `<Unknown>` Line

Whenever the source line for a PC can not be determined, the metrics for that PC are attributed to a special source line that is inserted at the top of the annotated source file. High metrics on that line indicates that part of the code from the given object module does not have line-mappings. Annotated disassembly can help you determine the instructions that do not have mappings.

Common Subexpression Elimination

One very common optimization recognizes that the same expression appears in more than one place, and that performance can be improved by generating the code for that expression in one place. For example, if the same operation appears in both the `if` and the `else` branches of a block of code, the compiler can move that operation to just before the `if` statement. When it does so, it assigns line numbers to the instructions based on one of the previous occurrences of the expression. If the line numbers assigned to the common code correspond to one branch of an `if` structure, and the code actually always takes the other branch, the annotated source shows metrics on lines within the branch that is not taken.

Annotated Disassembly Code

Annotated disassembly provides an assembly-code listing of the instructions of a function or object module, with the performance metrics associated with each instruction. Annotated disassembly can be displayed in several ways, determined by whether line-number mappings and the source file are available, and whether the object module for the function whose annotated disassembly is being requested is known:

- If the object module is not known, the Performance Analyzer disassembles the instructions for just the specified function, and does not show any source lines in the disassembly.

- If the object module is known, the disassembly covers all functions within the object module.
- If the source file is available, and line number data is recorded, the Performance Analyzer interleaves the source with the disassembly.
- If the compiler has inserted any commentary into the object code, it too, is interleaved in the disassembly.

Each instruction in the disassembly code is annotated with the following information.

- A source line number, as reported by the compiler
- Its relative address
- The hexadecimal representation of the instruction, if requested
- The assembler ASCII representation of the instruction

Where possible, call addresses are resolved to symbols (such as function names). Metrics are shown on the lines for instructions, but not on any interleaved source or commentary. Possible metric values are as described for source-code annotations, in TABLE 6-2.

When code is not optimized, line numbers are simple, and the interleaving of source and disassembled instructions appears natural. When optimization takes place, instructions from later lines sometimes appear before those from earlier lines. The Performance Analyzer's algorithm for interleaving is that whenever an instruction is shown as coming from line *N*, all source lines up to and including line *N* are written before the instruction. Compiler commentary associated with line *N* of the source are written immediately before that line.

Interpreting annotated disassembly is not straightforward. The leaf PC is the address of the next instruction to execute, so metrics attributed to an instruction should be considered as time spent waiting for the instruction to execute. However, the execution of instructions does not always happen in sequence, and there might be delays in the recording of the call stack. To make use of annotated disassembly, you should become familiar with the hardware on which you record your experiments and the way in which it loads and executes instructions.

The next few subsections discuss some of the issues of interpreting annotated disassembly.

Instruction Issue Grouping

Instructions are loaded and issued in groups: the instruction issue group. Which instructions are in the group depends on the hardware, the instruction type, the instructions already being executed, and any dependencies on other instructions or registers. This means that some instructions might be underrepresented because they are always issued in the same clock cycle as the previous instruction, so they never

represent the next instruction to be executed. It also means that when the call stack is recorded, there might be several instructions which could be considered the "next" instruction to execute.

Instruction Issue Delay

Sometimes, specific leaf PCs appear more frequently because the instruction that they represent is delayed before issue. This can occur for a number of reasons, some of which are listed below:

- The previous instruction takes a long time to execute and is not interruptible, for example when an instruction traps into the kernel.
- An arithmetic instruction needs a register that is not available because the register contents were set by an earlier instruction that has not yet completed. An example of this sort of delay is a load instruction that has a data cache miss.
- A floating-point arithmetic instructions is waiting for another floating-point instruction to complete. This situation occurs for instructions that cannot be pipelined, such as square root and floating-point divide.
- The instruction cache does not include the memory word that contains the instruction (I-cache miss).

Attribution of Hardware Counter Overflows

The call stack for a hardware counter overflow event is recorded at some point further on in the sequence of instructions than the point at which the overflow occurred, for various reasons including the time taken to handle the signal generated by the overflow. For some counters, such as cycles or instructions issued, this does not matter. For other counters, such as those counting cache misses or floating point operations, the metric is attributed to a different instruction from that which is responsible for the overflow. Often the true PC is only a few instructions before the recorded PC, and the instruction can be correctly located in the disassembly listing. However, if there is a branch target within this instruction range, it might be difficult or impossible to tell which instruction corresponds to the true PC.

CHAPTER 7

Manipulating Experiments and Viewing Annotated Code Listings

This chapter describes the utilities which are available for use with the Sampling Collector and Performance Analyzer.

This chapter covers the following topics:

- Manipulating Experiments
- Viewing Annotated Code Listings With `er_src`
- Other Utilities

Manipulating Experiments

Experiments are stored in a hidden directory, which is created by the Sampling Collector. To manipulate experiments, you cannot use the usual Unix commands `cp`, `mv` and `rm`. Three utilities which behave like the Unix commands have been provided to copy, move and delete experiments. These are `er_cp`(1), `er_mv`(1) and `er_rm`(1), and are described below.

The visible experiment file contains an absolute path to the experiment when the experiment is created. If you change the path without using one of these utilities to move the experiment, the path in the experiment no longer matches the location of the experiment. Running the Analyzer or `er_print` on the experiment in the new location either does not find the experiment, because the path is not valid, or finds the wrong experiment, if a new experiment has been created in the old location. The utilities remove the path from the experiment name when they copy or move an experiment.

The data in the experiment includes archive files for each of the load objects used by your program. These archive files contain the absolute path of the load object and the date on which it was last modified. This information is not changed when you move or copy an experiment.

`er_cp [-V]` *experiment1 experiment2*

`er_cp [-V]` *experiment-list directory*

The first form of the er_cp command copies *experiment1* to *experiment2*. If *experiment2* exists, er_cp exits with an error message. The second form copies a blank-separated list of experiments to a directory. If the directory already contains an experiment with the same name as one of the experiments being copied, er_mv exits with an error message. The -V option prints the version of er_cp.

`er_mv [-V]` *experiment1 experiment2*

`er_mv [-V]` *experiment-list directory*

The first form of the er_mv command moves *experiment1* to *experiment2*. If *experiment2* exists, er_mv exits with an error message. The second form moves a blank-separated list of experiments to a directory. If the directory already contains an experiment with the same name as one of the experiments being moved, er_mv exits with an error message. The -V option prints the version of er_mv.

`er_rm [-f] [-V]` *experiment-list*

Removes a list of experiments or experiment groups. When experiment groups are removed, each experiment in the group is removed then the group file is removed. The -f option suppresses error messages and ensures successful completion, whether or not the experiments are found. The -V option prints the version of er_rm.

Viewing Annotated Code Listings With er_src

Annotated source code and annotated disassembly code can be viewed using the er_src utility, without running an experiment. The display is generated in the same way as in the Performance Analyzer, except that it does not display any metrics. See “Examining Annotated Source Code and Disassembly Code” on page 95 for information on the Analyzer display. The syntax of the er_src command is

```
er_src [ options ] object item tag
```

object is the name of an executable, a shared object, or an object file (`.o` file).

item is the name of a function or of a source or object file used to build the executable or shared object; it can be omitted when an object file is specified.

tag is an index used to determine which *item* is being referred to when multiple functions have the same name. If it is not needed, it can be omitted. If it is needed and is omitted, a message listing the possible choices is printed.

The following sections describe the options accepted by the `er_src` utility.

`-c` *commentary-classes*

Define the compiler commentary classes to be shown. *commentary-classes* is a list of classes separated by colons. See "Source and Disassembly Listing Commands" on page 112 for a description of these classes.

The commentary classes can be specified in a defaults file. The system wide `er.rc` defaults file is read first, then a `.er.rc` file in the user's home directory, if present, then a `.er.rc` file in the current directory. Defaults from the `.er.rc` file in your home directory override the system defaults, and defaults from the `.er.rc` file in the current directory override both home and system defaults. These files are also used by the Performance Analyzer and `er_print`, but only the settings for source and disassembly compiler commentary are used by `er_src`.

See "Defaults Commands" on page 118 for a description of the defaults files. Commands in a defaults file other than `scc` and `dcc` are ignored by `er_src`.

`-d`

Include the disassembly in the listing. The default listing does not include the disassembly. If there is no source available, a listing of the disassembly without compiler commentary is produced.

`-h`

Show the hexadecimal representation of the instructions in the disassembly. The hexadecimal representation is not shown by default. This option is ignored if disassembly is not requested.

-o *filename*

Open the file *filename* for output of the listing. If *filename* is a dash (-), output is written to stdout. By default, the listing is displayed in your default text editor.

-V

Print the current release version.

Other Utilities

There are some other utilities that should not need to be used in normal circumstances. They are documented here for completeness, with a description of the circumstances in which it might be necessary to use them.

The er_archive Utility

The syntax of the er_archive command is as follows.

```
er_archive [-q] [-F] [-V] experiment
```

The er_archive utility is automatically run when an experiment completes normally, or upon initial invocation of the Analyzer or er_print command on an experiment. It reads the list of shared objects referenced in the experiment, and constructs an archive file for each. Each output file is named with a suffix of .archive, and contains function and module mappings for the shared object.

If the target program terminates abnormally, er_archive might not be run by the Sampling Collector. If the user wishes to examine the experiment from an abnormally-terminated run on a different machine from the one on which it was recorded, er_archive must be manually run on the experiment, on the machine on which the data was recorded.

An archive file is generated for all shared objects referred to in the experiment. These archives contain the addresses, sizes and names of each object file and each function in the load object, as well as the absolute path of the load object and a time stamp for its last modification.

If the shared object can not be found when er_archive is run, or if it has a time stamp differing from that recorded in the experiment, or if er_archive is run on a different machine from that on which the experiment was recorded, the archive file contains a warning. Warnings are also written to stderr whenever er_archive is run manually (without the -q flag).

The following sections describe the options accepted by the er_archive utility.

-q

Do not write any warnings to stderr. Warnings are incorporated into the archive file, and shown in the Analyzer or er_print output.

-F

Force writing or rewriting of archive files. This argument can be used to run er_archive by hand, to rewrite files that had warnings.

-V

Write version number information.

The er_export Utility

The syntax of the er_export command is as follows.

```
er_export [-V] experiment
```

The er_export utility converts the raw data in an experiment into ASCII text. The format and the content of the file are subject to change, and should not be relied on for any use. This utility is intended to be used only when the Performance Analyzer cannot read an experiment; the output allows the tool developers to understand the raw data and analyze the failure. The -V option prints version number information.

APPENDIX **A**

Profiling Programs With `prof`, `gprof`, and `tcov`

The tools discussed in this appendix are standard utilities for timing programs and obtaining performance data to analyze, and are called “traditional profiling tools”. The profiling tools `prof` and `gprof` are provided with Solaris versions 2.6, 7, and 8 of the Solaris operating environment *SPARC Platform Edition* and Solaris *Intel Platform Edition*. `tcov` is a code coverage tool provided with the Forte™ Developer product.

Note – If you want to track how many times a function is called or how often a line of source code is executed, use the traditional profiling tools. If you want a detailed analysis of where your program is spending time, you can get more accurate information using the Sampling Collector and Performance Analyzer. See Chapter 3 and Chapter 4 for information on using these tools.

TABLE A-1 describes the information that is generated by these standard performance profiling tools.

TABLE A-1 Performance Profiling Tools

Command	Output
`prof`	Generates a statistical profile of the CPU time used by a program and an exact count of the number of times each function is entered.
`gprof`	Generates a statistical profile of the CPU time used by a program, along with an exact count of the number of times each function is entered and the number of times each arc (caller-callee pair) in the program’s call graph is traversed.
`tcov`	Generates exact counts of the number of times each statement in a program is executed.

Not all the traditional profiling tools work on modules written in programming languages other than C. See the sections on each tool for more information about languages.

This appendix covers the following topics:

- Using prof to Generate a Program Profile
- Using gprof to Generate a Call Graph Profile
- Using tcov for Statement-Level Analysis
- Using tcov Enhanced for Statement-Level Analysis
- Creating Profiled Shared Libraries for tcov Enhanced

Using prof to Generate a Program Profile

prof generates a statistical profile of the CPU time used by a program and counts the number of times each function in a program is entered. Different or more detailed data is provided by the gprof call-graph profile and the tcov code coverage tools.

To generate a profile report using prof:

1. **Compile your program with the -p compiler option.**
2. **Run your program.**

   Profiling data is sent to a profile file called mon.out. This file is overwritten each time you run the program.
3. **Run prof to generate a profile report.**

   The syntax of the prof command is as follows.

```
% prof program-name
```

Here, *program-name* is the name of the executable. The profile report is written to stdout. It is presented as a series of rows for each routine under these column headings:

- %Time—The percentage of the total CPU time consumed by this routine.
- Seconds—The total CPU time accounted for by this routine.
- Cumsecs—A running sum of the number of seconds accounted for by this routine and those listed before it.
- #Calls—The number of times this routine is called.
- msecs/call—The average number of milliseconds this routine consumes each time it is called.
- Name—The name of the routine.

The use of `prof` is illustrated in the following example.

```
% cc -p -o index.assist index.assist.c
% index.assist
% prof index.assist
```

The profile report from `prof` is shown in the table below:

%Time	Seconds	Cumsecs	#Calls	msecs/ call	Name
19.4	3.28	3.28	11962	0.27	compare_strings
15.6	2.64	5.92	32731	0.08	_strlen
12.6	2.14	8.06	4579	0.47	__doprnt
10.5	1.78	9.84			mcount
9.9	1.68	11.52	6849	0.25	_get_field
5.3	0.90	12.42	762	1.18	_fgets
4.7	0.80	13.22	19715	0.04	_strcmp
4.0	0.67	13.89	5329	0.13	_malloc
3.4	0.57	14.46	11152	0.05	_insert_index_entry
3.1	0.53	14.99	11152	0.05	_compare_entry
2.5	0.42	15.41	1289	0.33	lmodt
0.9	0.16	15.57	761	0.21	_get_index_terms
0.9	0.16	15.73	3805	0.04	_strcpy
0.8	0.14	15.87	6849	0.02	_skip_space
0.7	0.12	15.99	13	9.23	_read
0.7	0.12	16.11	1289	0.09	ldivt
0.6	0.10	16.21	1405	0.07	_print_index

```
.
.
.  (The rest of the output is insignificant)
```

The profile report shows that most of the program execution time is spent in the `compare_strings()` routine; after that, most of the CPU time is spent in the `_strlen()` library routine. To make this program more efficient, the user would concentrate on the `compare_strings()` function, which consumes nearly 20% of the total CPU time, and improve the algorithm or reduce the number of calls.

It is not obvious from the `prof` profile report that `compare_strings()` is heavily recursive, but you can deduce this by using the call graph profile described in "Using `gprof` to Generate a Call Graph Profile" on page 156. In this particular case, improving the algorithm also reduces the number of calls.

Note – For Solaris 7 and 8 platforms, the profile of CPU time is accurate for programs that use multiple CPUs, but the fact that the counts are not locked may affect the accuracy of the counts for functions.

Using `gprof` to Generate a Call Graph Profile

While the flat profile from `prof` can provide valuable data for performance improvements, a more detailed analysis can be obtained by using a call graph profile to display a list identifying which modules are called by other modules, and which modules call other modules. Sometimes removing calls altogether can result in performance improvements.

Note – `gprof` attributes the time spent within a function to the callers in proportion to the number of times that each arc is traversed. Because all calls are not equivalent in performance, this behavior might lead to incorrect assumptions. See "The `gprof` Fallacy" on page 17 for an example.

Like `prof`, `gprof` generates a statistical profile of the CPU time that is used by a program and it counts the number of times that each function is entered. `gprof` also counts the number of times that each arc in the program's call graph is traversed. An *arc* is a caller-callee pair.

Note – For Solaris 7 and 8 platforms, the profile of CPU time is accurate for programs that use multiple CPUs, but the fact that the counts are not locked may affect the accuracy of the counts for functions.

To generate a profile report using gprof:

1. **Compile your program with the appropriate compiler option.**
   - For C programs, use the -xpg option.
   - For Fortran programs, use the -pg option.
2. **Run your program.**

   Profiling data is sent to a profile file called gmon.out. This file is overwritten each time you run the program.
3. **Run gprof to generate a profile report.**

   The syntax of the prof command is as follows.

```
% gprof program-name
```

Here, *program-name* is the name of the executable. The profile report is written to stdout, and can be large. The report consists of two major items:

- The full call graph profile, which shows information about the callers and callees of each routine in the program. The format is illustrated in the example given below.
- The “flat” profile, which is similar to the summary the prof command supplies.

The profile report from gprof contains an explanation of what the various parts of the summary mean and identifies the granularity of the sampling, as shown in the following example.

```
granularity: each sample hit covers 4 byte(s) for 0.07% of 14.74
seconds
```

The “4 bytes” means resolution to a single instruction. The “0.07% of 14.74 seconds“ means that each sample, representing ten milliseconds of CPU time, accounts for 0.07% of the run.

The use of gprof is illustrated in the following example.

```
% cc -xpg -o index.assist index.assist.c
% index.assist
% gprof index.assist > g.output
```

The following table is part of the call graph profile.

index	%time	self	descendants	called/total parents called+self called/total children	name	index
		0.00	14.47	1/1	start	[1]
[2]	98.2	0.00	14.47	1	_main	[2]
		0.59	5.70	760/760	_insert_index_entry	[3]
		0.02	3.16	1/1	_print_index	[6]
		0.20	1.91	761/761	_get_index_terms	[11]
		0.94	0.06	762/762	_fgets	[13]
		0.06	0.62	761/761	_get_page_number	[18]
		0.10	0.46	761/761	_get_page_type	[22]
		0.09	0.23	761/761	_skip_start	[24]
		0.04	0.23	761/761	_get_index_type	[26]
		0.07	0.00	761/820	_insert_page_entry	[34]
				10392	_insert_index_entry	[3]
		0.59	5.70	760/760	_main	[2]
[3]	42.6	0.59	5.70	760+10392	_insert_index_entry	[3]
		0.53	5.13	11152/11152	_compare_entry	[4]
		0.02	0.01	59/112	_free	[38]
		0.00	0.00	59/820	_insert_page_entry	[34]
				10392	_insert_index_entry	[3]

In this example there are 761 lines of data in the input file to the `index.assist` program. The following conclusions can be made:

- `fgets()` is called 762 times. The last call to `fgets()` returns an end-of-file.
- The `insert_index_entry()` function is called 760 times from `main()`.
- In addition to the 760 times that `insert_index_entry()` is called from `main()`, `insert_index_entry()` also calls itself 10,392 times. `insert_index_entry()` is heavily recursive.
- `compare_entry()`, which is called from `insert_index_entry()`, is called 11,152 times, which is equal to 760+10,392 times. There is one call to `compare_entry()` for every time that `insert_index_entry()` is called. This is correct. If there were a discrepancy in the number of calls, you would suspect some problem in the program logic.
- `insert_page_entry()` is called 820 times in total: 761 times from `main()` while the program is building index nodes, and 59 times from `insert_index_entry()`. This frequency indicates that there are 59 duplicated index entries, so their page number entries are linked into a chain with the index nodes. The duplicate index entries are then freed; hence the 59 calls to `free()`.

Using `tcov` for Statement-Level Analysis

The `tcov` utility gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also summarizes information about basic blocks. `tcov` does not produce any time-based data.

Note – Although `tcov` works with both C and C++ programs, it does not support files that contain `#line` or `#file` directives. `tcov` does not enable test coverage analysis of the code in the `#include` header files.

To generate annotated source code using `tcov`:

1. **Compile your program with the appropriate compiler option.**
   - For C programs, use the `-xa` option.
   - For Fortran and C++ programs, use the `-a` option.

If you compile with the `-a` or `-xa` option you must also link with it. The compiler creates a coverage data file with the suffix `.d` for each object file. The coverage data file is created in the directory specified by the environment variable `TCOVDIR`. If `TCOVDIR` is not set, the coverage data file is created in the current directory.

Note – Programs compiled with `-xa` (C) or `-a` (other compilers) run more slowly than they normally would, because updating the `.d` file for each execution takes considerable time.

---

2. **Run your program.**

   When your program completes, the coverage data files are updated.

3. **Run `tcov` to generate annotated source code.**

   The syntax of the `tcov` command is as follows.

```
% tcov options source-file-list
```

Here, *source-file-list* is a list of the source code filenames. For a list of options, see the `tcov`(1) man page. The default output of `tcov` is a set of files, each with the suffix `.tcov`, which can be changed with the `-o` *filename* option.

A program compiled for code coverage analysis can be run multiple times (with potentially varying input); `tcov` can be used on the program after each run to compare behavior.

The following example illustrates the use of `tcov`.

```
% cc -xa -o index.assist index.assist.c
% index.assist
% tcov index.assist.c
```

This small fragment of the C code from one of the modules of `index.assist` shows the `insert_index_entry()` function, which is called recursively. The numbers to the left of the C code show how many times each statement was executed. The `insert_index_entry()` function is called 11,152 times.

```
        struct index_entry *
11152-> insert_index_entry(node, entry)
        struct index_entry *node;
        struct index_entry *entry;
        {
            int result;
            int level;

            result = compare_entry(node, entry);
            if (result == 0) { /* exact match */
                               /* Place the page entry for the duplicate */
                               /* into the list of pages for this node */
59  ->          insert_page_entry(node, entry->page_entry);
                free(entry);
                return(node);
            }

11093->     if (result > 0)/* node greater than new entry -- */
                           /* move to lesser nodes */
3956->          if (node->lesser != NULL)
3626->              insert_index_entry(node->lesser, entry);
                else {
330 ->              node->lesser = entry;
                    return (node->lesser);
                }
            else        /* node less than new entry -- */
                        /* move to greater nodes */
7137->          if (node->greater != NULL)
6766->              insert_index_entry(node->greater, entry);
                else {
371 ->              node->greater = entry;
                    return (node->greater);
                }
        }
```

`tcov` places a summary like the following at the end of the annotated program listing for `index.assist.tcov`:

```
             Top 10 Blocks


           Line       Count

           240        21563

           241        21563

           245        21563

           251        21563

           250        21400

           244        21299

           255        20612

           257        16805

           123        12021

           124        11962


 77        Basic blocks in this file

 55        Basic blocks executed

 71.43     Percent of the file executed


           439144    Total basic block executions

           5703.17   Average executions per basic block
```

Creating tcov Profiled Shared Libraries

It is possible to create a tcov profiled shareable library and use it in place of the corresponding library in binaries which have already been linked. Include the -xa (C) or -a (other compilers) option when creating the shareable libraries, as shown in this example.

```
% cc -G -xa -o foo.so.1 foo.o
```

This command includes a copy of the tcov profiling subroutines in the shareable libraries, so that clients of the library do not need to relink. If a client of the library is already linked for profiling, then the version of the tcov subroutines used by the client is used to profile the shareable library.

Locking Files

tcov uses a simple file-locking mechanism for updating the block coverage database in the .d files. It employs a single file, tcov.lock, for this purpose. Consequently, only one executable compiled with -xa (C) or -a (other compilers) should be running on the system at a time. If the execution of the program compiled with the -xa (or -a) option is manually terminated, then the tcov.lock file must be deleted manually.

Files compiled with the -xa or -a option call the profiling tool subroutines automatically when a program is linked for tcov profiling. At program exit, these subroutines combine the information collected at runtime for file xyz.f (for example) with the existing profiling information stored in file xyz.d. To ensure this information is not corrupted by several people simultaneously running a profiled binary, a xyz.d.lock lock file is created for xyz.d for the duration of the update. If there are any errors in opening or reading xyz.d or its lock file, or if there are inconsistencies between the runtime information and the stored information, the information stored in xyz.d is not changed.

If you edit and recompile xyz.f the number of counters in xyz.d can change. This is detected if an old profiled binary is run.

If too many people are running a profiled binary, some of them cannot obtain a lock. An error message is displayed after a delay of several seconds. The stored information is not updated. This locking is safe across a network. Since locking is performed on a file-by-file basis, other files may be correctly updated.

The profiling subroutines attempt to deal with automounted file systems that have become inaccessible. They still fail if the file system containing a coverage data file is mounted with different names on different machines, or if the user running the

profiled binary does not have permission to write to either the coverage data file or the directory containing it. Be sure all the directories are uniformly named and writable by anyone expected to run the binary.

Errors Reported by `tcov` Runtime Routines

The following error messages may be reported by the `tcov` runtime routines:

- The user running the binary lacks permission to read or write to the coverage data file. The problem also occurs if the coverage data file has been deleted.

```
tcov_exit: Could not open coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- The user running the binary lacks permission to write to the directory containing the coverage data file. The problem also occurs if the directory containing the coverage data file is not mounted on the machine where the binary is being run.

```
tcov_exit: Could not write coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- Too many users are trying to update a coverage data file at the same time. The problem also occurs if a machine has crashed while a coverage data file is being updated, leaving behind a lock file. In the event of a crash, the longer of the two files should be used as the post-crash coverage data file. Manually remove the lock file.

```
tcov_exit: Failed to create lock file 'lock-file-name' for coverage
data file 'coverage-data-file-name' after 5 tries. Is someone else
running this executable?
```

- No memory is available, and the standard I/O package will not work. You cannot update the coverage data file at this point.

```
tcov_exit: Stdio failure, probably no memory left.
```

- The lock file name is longer by six characters than the coverage data file name. Therefore, the derived lock file name may not be legal.

```
tcov_exit: Coverage data file path name too long (length
characters) 'coverage-data-file-name'.
```

- A library or binary that has tcov profiling enabled is simultaneously being run, edited, and recompiled. The old binary expects a coverage data file of a certain size, but the editing often changes that size. If the compiler creates a new coverage data file at the same time that the old binary is trying to update the old coverage data file, the binary may see an apparently empty or corrupt coverage file.

```
tcov_exit: Coverage data file 'coverage-data-file-name' is too short.
Is it out of date?
```

Using tcov Enhanced for Statement-Level Analysis

Like the original tcov, tcov Enhanced gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also gives a summary of information about basic blocks. tcov Enhanced works with both C and C++ source files.

tcov Enhanced overcomes some of the shortcomings of the original tcov. The improved features of tcov Enhanced are:

- It provides more complete support for C++.
- It supports code found in #include header files and corrects a flaw that obscured coverage numbers for template classes and functions.
- Its runtime is more efficient than the original tcov runtime.
- It is supported for all the platforms that the compilers support.

To generate annotated source code using tcov Enhanced:

1. **Compile your program with the `-xprofile=tcov` compiler option.**

   Unlike tcov, tcov Enhanced does not generate any files at compile time.

2. **Run your program.**

   A directory is created to store the profile data, and a single coverage data file called `tcovd` is created in that directory. By default, the directory is created in the location where you run the program *program-name*, and it is called *program-name*`.profile`. The directory is also known as the *profile bucket*. The defaults can be changed using environment variables (see "`tcov` Directories and Environment Variables" on page 167).

3. **Run `tcov` to generate annotated source code.**

   The syntax of the `tcov` command is as follows.

```
% tcov option-list source-file-list
```

Here, *source-file-list* is a list of the source code filenames, and *option-list* is a list of options, which can be obtained from the `tcov`(1) man page. You must include the `-x` option to enable `tcov` Enhanced processing.

The default output of `tcov` Enhanced is a set of annotated source files whose names are derived by appending `.tcov` to the corresponding source file name.

The following example illustrates the syntax of `tcov` Enhanced.

```
% cc -xprofile=tcov -o index.assist index.assist.c
% index.assist
% tcov -x index.assist.profile index.assist.c
```

The output of `tcov` Enhanced is identical to the output from the original `tcov`.

Creating Profiled Shared Libraries for `tcov` Enhanced

You can create profiled shared libraries for use with `tcov` Enhanced by including the `-xprofile=tcov` compiler option, as shown in the following example.

```
% cc -G -xprofile=tcov -o foo.so.1 foo.o
```

Locking Files

tcov Enhanced uses a simple file-locking mechanism for updating the block coverage data file. It employs a single file created in the same directory as the tcovd file. The file name is tcovd.temp.lock. If execution of the program compiled for coverage analysis is manually terminated, then the lock file must be deleted manually.

The locking scheme does an exponential back-off if there is a contention for the lock. If, after five tries, the tcov runtime cannot acquire the lock, it exits, and the data is lost for that run. In this case, the following message is displayed.

```
tcov_exit: temp file exists, is someone else running this
executable?
```

tcov Directories and Environment Variables

When you compile a program for tcov and run the program, the running program generates a profile bucket. If a previous profile bucket exists, the program uses that profile bucket. If a profile bucket does not exist, it creates the profile bucket.

The profile bucket specifies the directory where the profile output is generated. The name and location of the profile output are controlled by defaults that you can modify with environment variables.

Note – tcov uses the same defaults and environment variables that are used by the compiler options that you use to gather profile feedback: -xprofile=collect and -xprofile=use. For more information about these compiler options, see the documentation for the relevant compiler.

The default profile bucket is named after the executable with a .profile extension and is created in the directory where the executable is run. Therefore, if you run a program called /usr/bin/xyz from /home/userdir, the default behavior is to create a profile bucket called xyz.profile in /home/userdir.

A UNIX process can change its current working directory during the execution of a program. The current working directory used to generate the profile bucket is the current working directory of the program at exit. In the rare case where a program actually does change its current working directory during execution, you can use the environment variables to control where the profile bucket is generated.

You can set the following environment variables to modify the defaults:

- SUN_PROFDATA

  Can be used to specify the name of the profile bucket at runtime. The value of this variable is always appended to the value of SUN_PROFDATA_DIR if both variables are set. Doing this may be useful if the name of the executable is not the same as the value in argv[0] (for example, the invocation of the executable was through a symbolic link with a different name).

- SUN_PROFDATA_DIR

  Can be used to specify the name of the directory that contains the profile bucket. It is used at runtime and by the tcov command.

- TCOVDIR

  TCOVDIR is supported as a synonym for SUN_PROFDATA_DIR to maintain backward compatibility. Any setting of SUN_PROFDATA_DIR causes TCOVDIR to be ignored. If both SUN_PROFDATA_DIR and TCOVDIR are set, a warning is displayed when the profile bucket is generated.

  TCOVDIR is used at runtime and by the tcov command.

Index

A

adding experiments to the Performance Analyzer, 84
address spaces, text and data regions, 136
address-space data
  collecting from dbx, 64
  collecting from the Sampling Collector window, 63
  collecting with collect, 56
  displaying in er_print, 120
  displaying in the Performance Analyzer, 102
  information recorded, 50
  pages and segments, detailed information about, 103
aliased functions, 137
alternate entry points in Fortran functions, 139
analyzer command, 82
Analyzer, *See* Performance Analyzer
annotated disassembly code
  description, 144
  displaying in the Performance Analyzer, 95
  hardware counter metric attribution, 146
  instruction issue dependencies, 145
  interpreting, 145
  metric formats, 143
  printing in er_print, 113
  viewing with er_src, 148
annotated source code
  compiler commentary, 144
  description, 143
  displaying in the Performance Analyzer, 95
  from tcov, 161
  metric formats, 143
  printing in er_print, 112
  required compiler options, 54
  viewing with er_src, 148
arc, call graph, defined, 156
attaching the Sampling Collector to a running process, 69
attributed metrics
  defined, 78
  illustrated, 79
  use of, 79

B

body functions, compiler-generated
  defined, 132
  displayed by the Performance Analyzer, 140
  names, 132
  propagation of inclusive metrics, 135

C

C++ name demangling, setting default library in .er.rc file, 119
call stacks
  defined, 128
  effect of tail-call optimization on, 131
  mapping addresses to program structure, 136
  unwinding, 128

caller-callee metrics
attributed, defined, 79
default, 93
displaying in the Performance Analyzer, 92
displaying list of in er_print, 117
printing in er_print, 111
selecting in er_print, 111
selecting in the Performance Analyzer, 94
sort order in er_print, 112
sort order in the Performance Analyzer, 94
clock-based profiling
collecting data from the Sampling Collector window, 61
collecting data in dbx, 65
collecting data with collect, 57
comparison with gethrtime and gethrvtime, 126
data in profile packet, 124
defined, 46
distortion due to overheads, 125
high-resolution, 46
interval, *See* profiling interval
metrics, defined, 76, 124
collect command
address-space data (-a) option, 56
clock-based profiling data (-p) option, 57
collecting data with, 55
disable data collection (-n) option, 57
experiment directory (-d) option, 59
experiment group (-g) option, 59
experiment name (-o) option, 59
hardware-counter data (-h) option, 56
listing the options of, 56
record sample point (-l) option, 58
stop target after exec (-x) option, 58
synchronization wait tracing data (-s) option, 58
syntax, 55
toggle data recording (-y) option, 59
verbose (-v) option, 60
version (-V) option, 60
Collector, *see* Sampling Collector
commentary, compiler, in annotated source code, 144
common subexpression elimination, 144
comparing experiments, 83
compiler commentary
classes in annotated disassembly code, er_print, 114
classes in annotated source code, er_print, 113
description, in annotated source code, 144
compiler-generated body functions
defined, 132
displayed by the Performance Analyzer, 140
names, 132
propagation of inclusive metrics, 135
compilers, accessing, 3
compiling
for data collection and analysis, 54
for gprof, 157
for prof, 154
for tcov, 159
for tcov Enhanced, 165
copying an experiment, 148
correlation, effect on metrics, 125

D

data collection
controlling from your program, 52
disabling in dbx, 67
enabling in dbx, 67
from MPI programs, 71
linking for, 54
MPI program, using collect, 73
MPI program, using dbx, 73
pausing in dbx, 67
pausing in your program, 53
rate of, 51
resuming in dbx, 67
resuming in your program, 53
using dbx, 64
using the collect command, 55
using the Sampling Collector window, 60
dbx
collecting data under MPI, 73
running the Sampling Collector in, 64
dbx collector subcommands
address_space, 64
close, 68
disable, 67
enable, 67
enable_once (obsolete), 68

hwprofile, 65
pause, 67
profile, 65
quit, 68
resume, 67
sample, 66
show, 68
status, 68
store, 67
synctrace, 66
dialog boxes
Add Experiment, 84
Callers-Callees Metrics, 94
Create Mapfile, 97
Drop Experiment, 85
Find, 91
Function List Metrics, 88
Load Experiment, 84
Print, 103
Select Load Objects Included, 87
directives, parallelization, 132
disassembly code, annotated
description, 144
displaying in the Performance Analyzer, 95
metric formats, 143
printing in er_print, 113
viewing with er_src, 148
disk space, estimating for experiments, 51
displays
Address Space, 102
Execution Statistics, 101
Function List, 87
Overview, 99
documentation index, 5
documentation, accessing, 5
dropping experiments from the Performance Analyzer, 85

E

editor window, sharing in the Performance Analyzer, 83
entry points, alternate, in Fortran functions, 139
environment variables
LD_LIBRARY_PATH, 70
LD_PRELOAD, 70
MANPATH, 5
PATH, 4
SUN_PROFDATA, 168
SUN_PROFDATA_DIR, 168
TCOVDIR, 160, 168
er_archive utility, 150
er_cp utility, 148
er_export utility, 151
er_mv utility, 148
er_print commands
address_space, 120
callers-callees, 111
cmetric_list, 117
cmetrics, 111
csort, 112
dcc, 114
disasm, 113
dmetrics, 118
dsort, 119
exp_list, 116
fsummary, 109
functions, 109
gdemangle, 119
header, 120
help, 121
limit, 119
lwp_list, 116
lwp_select, 115
mapfile, 120
metric_list, 117
metrics, 109
name, 119
object_list, 116
object_select, 115
objects, 110
osummary, 110
outfile, 119
overview, 120
quit, 121
sample_list, 116
sample_select, 115
scc, 113
script, 121
sort, 110
source, 112
src, 112
statistics, 120
thread_list, 116
thread_select, 115
Version, 121
version, 121

er_print utility
command-line options, 106
commands, *See* er_print commands
metric keywords, 108
metric lists, 106
purpose, 105
syntax, 106
er_rm utility, 148
er_src utility, 148
errors reported by tcov, 164
exclusive metrics
defined, 78
illustrated, 79
use of, 79
execution statistics
copying and pasting, 101
displaying in the Performance Analyzer, 101
information recorded, 50
printing in er_print, 120
exiting the Performance Analyzer, 83
experiment directory
navigating to in the Sampling Collector window, 61
specifying in dbx, 67
specifying with collect, 59
experiment group, 148
default name, 50
defined, 50
name restrictions, 50
specifying name in dbx, 68
specifying name with collect, 59
experiment name
default, 50
MPI default, 72
MPI, using MPI_comm_rank and a script, 74
restrictions, 50
specifying in dbx, 68
specifying in the Sampling Collector window, 61
specifying with collect, 59
experiments
adding to the Performance Analyzer, 84
comparing, 83
copying, 148
default name, 50
defined, 50
dropping from the Performance Analyzer, 85
listing in er_print, 116
loading into the Performance Analyzer, 84
moving, 148
moving MPI, 72
MPI storage issues, 72
naming, 50
removing, 148
showing header information in er_print, 120
storage requirements, estimating, 51
terminating from your program, 54
explicit multithreading, 131

F

fast traps, 130
filtering information, 85
fixed-width and proportional sample graphs, switching between, 99
frame
defined, 129
from trap handler, 130
reuse of in tail-call optimization, 130
function calls
between shared objects, 129
imputed, in OpenMP programs, 135
in single-threaded programs, 128
recursive, 80, 93
function list
printing in er_print, 109
searching for functions and load objects, 91
sort order, specifying in er_print, 110
sort order, specifying in the Performance Analyzer, 88
function metrics
selecting display of, 88
summary, viewing, 90
function names, C++
choosing long or short form in er_print, 119
setting default demangling library in .er.rc file, 119
function-list metrics
default, 88
displaying list of in er_print, 117
selecting default in .er.rc file, 118
selecting in er_print, 109
selecting in the Performance Analyzer, 88
setting default sort order in .er.rc file, 119

functions
aliased, 137
alternate entry points (Fortran), 139
body, compiler-generated, *See* body functions, compiler-generated
defined, 137
inlined, 139
load-object, addresses of, 137
mapping to source code, 137
non-unique, names of, 138
outline, 141
searching for in the Function List display, 91
static, in stripped shared libraries, 138
static, with duplicate names, 138
`<Total>`, 142
`<Unknown>`, 141
wrapper, 138

G

`gprof`
limitations, 156
output from, interpreting, 157
summary, 153
using, 157

H

hardware counter list
description of fields, 48
obtaining from the Sampling Collector window, 62
obtaining with `collect`, 56
obtaining with `dbx collector`, 65
hardware counters
choosing in the Sampling Collector window, 62
choosing with `collect`, 56
choosing with `dbx collector`, 65
list described, 48
names, 48
obtaining a list of, 56, 62, 65
overflow value, 47
hardware-counter overflow profiling
collecting data from the Sampling Collector window, 62
collecting data with `collect`, 56
collecting data with `dbx`, 65
data in profile packet, 127
defined, 47
limitations, 49
hardware-counter overflow value
defined, 47
experiment size, effect on, 51
setting in the Sampling Collector window, 62
setting with `collect`, 57
setting with `dbx collector`, 65
high-resolution profiling, 46

I

inclusive metrics
defined, 78
illustrated, 79
use of, 79
inlined functions, 139
input file
terminating in `er_print`, 121
to `er_print`, 121
instruction issue, effect on annotated disassembly display, 145

K

keywords, metric, `er_print` utility, 108

L

`LD_LIBRARY_PATH` environment variable, 70
`LD_PRELOAD` environment variable, 70
`libcollector.so` shared library
preloading, 70
using in your program, 52
limitations
experiment group name, 50
experiment name, 50
hardware-counter overflow profiling, 49
profiling interval value, 46
`tcov`, 159
limiting output in `er_print`, 119
load objects
contents of, 137
defined, 137

functions, addresses of, 137
listing selected, in er_print, 116
printing list in er_print, 110
searching for in the Function List display, 91
selecting in er_print, 115
selecting in the Performance Analyzer, 87
symbol tables, 137
loading an experiment into the Performance Analyzer, 84
load-object metrics
selecting display of, 88
summary, viewing, 90
lock file management
tcov, 163
tcov Enhanced, 167
LWPs
listing selected, in er_print, 116
selecting in er_print, 115
selecting in the Performance Analyzer, 85

M

man pages, accessing, 3
MANPATH environment variable, setting, 5
mapfiles
generating with er_print, 120
generating with the Performance Analyzer, 97
reordering a program with, 98
metrics
assigning to instructions, 145
attributed, defined, 78
attributed, use of, 79
caller-callee, *See* caller-callee metrics
clock-based profiling, defined, 76, 124
defined, 76
effect of correlation, 125
exclusive, defined, 78
exclusive, use of, 79
filtering by experiment, LWP, thread and sample, 85
function-list, default, 88
hardware counter, attributing to instructions, 146
inclusive, defined, 78
inclusive, use of, 79
summary for a function or load object, 90
switching between function and load-object display, 88
synchronization delay events, defined, 77
synchronization wait time, defined, 77
synchronization wait tracing, defined, 77
microtasking library routines, 132
moving an experiment, 148
MPI (Message Passing Interface), programs written with, 71
MPI experiments
default name, 51
loading into the Performance Analyzer, 84
moving, 72
storage issues, 72
MPI programs
attaching to, 70
collecting data from, 71
collecting data with collect, 73
collecting data with dbx, 73
experiment names, 51, 72
experiment storage issues, 72
tracing blocking calls, 47
multithreaded applications, attaching the Sampling Collector to, 69
multithreading
explicit, 131
parallelization directives, 132

N

naming an experiment, 50
navigating through program structure, 93
non-unique function names, 138

O

OpenMP parallelization
with Forte Developer compilers, 132
optimizations
common subexpression elimination, 144
tail-call, 130
options, command-line, er_print utility, 106
outline functions, 141
output file, in er_print, 119

overflow value, hardware-counter
defined, 47
setting in the Sampling Collector window, 62
setting with `collect`, 57
setting with `dbx collector`, 65
overview data
displaying in the Performance Analyzer, 99
printing in `er_print`, 120

P

page and segment address-space displays, switching between, 103
pages, address-space, detailed information about, 103
parallel execution
call sequence, 133
compiling for, 132
directives, 132
`PATH` environment variable, setting, 4
Performance Analyzer
adding experiments to, 84
caller-callee metrics, default, 93
defined, 7, 75
dropping experiments from, 85
exiting, 83
function-list metrics, default, 88
loading an experiment, 84
main window, 82
mapfiles, generating, 97
printing the display, 103
searching for functions and load objects, 91
sharing of editor window between sessions, 83
starting, 81
performance data, conversion into metrics, 76
performance metrics, *See* metrics
PLT (Program Linkage Table), 129
preloading `libcollector.so`, 70
Print dialog box, 103
printing the Performance Analyzer displays, 103
process address-space text and data regions, 136
process times
detailed analysis of in samples, 100
sample, displaying, 99
`prof`
limitations, 156
output from, 155
summary, 153
using, 154
profile bucket, `tcov` Enhanced, 166, 167
profile packet
clock-based data, 124
defined, 45
hardware-counter overflow data, 127
header information, 45
size of, 51
synchronization wait tracing data, 126
profiled shared libraries, creating
for `tcov`, 163
for `tcov` Enhanced, 166
profiling interval
defined, 46
experiment size, effect on, 51
limitations on value, 46
setting in the Sampling Collector window, 61
setting with `dbx collector`, 65
setting with the `collect` command, 57
profiling, defined, 45
program counter (PC), defined, 128
program execution
call stacks described, 128
explicit multithreading, 131
OpenMP parallel, 133
shared objects and function calls, 129
signal handling, 129
single-threaded, 128
tail-call optimization, 130
traps, 129
Program Linkage Table (PLT), 129
program structure
mapping call-stack addresses to, 136
navigating through, 93
program, reordering with a mapfile, 98
proportional and fixed-width sample graphs, switching between, 99

R

recursive function calls, 80, 93

regular expressions, used by the Performance Analyzer search facility, 91
removing an experiment or experiment group, 148
reordering a program with a mapfile, 98

S

samples
circumstances of recording, 49
defined, 49
detailed analysis of process times in, 100
displaying overview information, 99
information contained in packet, 49
interval, *See* sampling interval
listing selected, in er_print, 116
packet defined, 45
process times, displaying, 99
recording from your program, 53
recording with collect, 58
selecting in er_print, 115
selecting in the Performance Analyzer, 85
selecting sampling mode in dbx, 66
Sampling Collector
API, using in your program, 52
attaching to a running process, 69
defined, 7, 45
disabling in dbx, 67
disabling in the Sampling Collector window, 60
enabling in dbx, 67
enabling in the Sampling Collector window, 60
running from the Sampling Collector window, 60
running in dbx, 64
running with collect, 55
sampling interval
defined, 50
setting in dbx, 66
setting in the Sampling Collector window, 63
segment and page address-space displays, switching between, 103
segments, address-space, detailed information about, 103
shared objects, function calls between, 129
shell prompts, 3
signals, 129
single-threaded program execution, and function calls, 128
Solaris versions supported, 3
sort order
caller-callee metrics, in er_print, 112
caller-callee metrics, in the Performance Analyzer, 94
function list, specifying in er_print, 110
function list, specifying in the Performance Analyzer, 88
source code
annotated, *See* annotated source code
mapping functions to, 137
stack frame
defined, 129
reuse of in tail-call optimization, 130
starting the Performance Analyzer, 81
static functions
duplicate names, 138
in stripped shared libraries, 138
storage requirements, estimating for experiments, 51
summary metrics
copying and pasting, 91
function or load object, displaying in the Performance Analyzer, 90
function, printing in er_print, 109
load objects, printing in er_print, 110
SUN_PROFDATA environment variable, 168
SUN_PROFDATA_DIR environment variable, 168
symbol tables, load-object, 137
synchronization delay events
data in profile packet, 126
defined, 46
metric defined, 77
synchronization wait time
defined, 46, 126
metric, defined, 77
synchronization wait tracing
collecting data from the Sampling Collector window, 62
collecting data in dbx, 66
collecting data with collect, 58
data in profile packet, 126
defined, 46
metrics, defined, 77
MPI blocking calls, 47

preloading libcollector.so, 70
threshold, *See* threshold, synchronization wait tracing
wait time, 46, 126
syntax
er_archive utility, 150
er_export utility, 151
er_print utility, 106
er_src utility, 148

T

tail-call optimization, 130
tcov
annotated source code, 161
compiling a program for, 159
errors reported by, 164
limitations, 159
lock file management, 163
output, interpreting, 161
profiled shared libraries, creating, 163
summary, 153
using, 159
tcov Enhanced
advantages of, 165
compiling a program for, 165
lock file management, 167
profile bucket, 166, 167
profiled shared libraries, creating, 166
using, 165
TCOVDIR environment variable, 160, 168
text editors, choosing, 95
threads
bound and unbound, 131, 136
creation of, 131
listing selected, in er_print, 116
main, 133
scheduling of, 131, 132
selecting in er_print, 115
selecting in the Performance Analyzer, 85
wait mode, 136
worker, 131, 133
threshold, synchronization wait tracing
calibration, 47
defined, 47
increasing to minimize overhead, 127
setting in the Sampling Collector window, 62
setting with dbx collector, 66
setting with the collect command, 58
<Total> function, 142
traps, 129
typographic conventions, 2

U

<Unknown> function
callers and callees, 142
mapping of PC to, 141
<Unknown> line, in annotated source code, 144
unwinding the stack, 128

V

version information
for collect, 60
for er_cp, 148
for er_mv, 148
for er_print, 121
for er_rm, 148
for er_src, 150
for the Performance Analyzer, 82

W

wait time, *See* synchronization wait time
windows
Analyzer, 82
Callers-Callees, 93
Page Properties, 103
Sample Details, 100
Sampling Collector, 61
Segment Properties, 103
Summary Metrics, 90
wrapper functions, 138