Performance Analyzer

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

Part Number 819-0493-11
April 2005, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Contents

Before You Begin 15

How This Book Is Organized 16

Typographic Conventions 17

Supported Platforms 18

Shell Prompts 18

Accessing Sun Studio Software and Man Pages 19

Accessing Sun Studio Documentation 22

Accessing Related Solaris Documentation 25

Resources for Developers 26

Contacting Sun Technical Support 26

Sending Your Comments 27

1. Overview of the Performance Analyzer 29

Starting the Performance Analyzer From the Integrated Development Environment 29

The Tools of Performance Analysis 29

The Collector Tool 30

The Performance Analyzer Tool 30

The `er_print` Utility 31

The `prof`, `gprof`, and `tcov` Tools 31

The Performance Analyzer Window 32

2. Performance Data 33

What Data the Collector Collects 34

Clock Data 35

Hardware Counter Overflow Profiling Data 37

Synchronization Wait Tracing Data 40

Heap Tracing (Memory Allocation) Data 41

MPI Tracing Data 42

Global (Sampling) Data 43

How Metrics Are Assigned to Program Structure 44

Function-Level Metrics: Exclusive, Inclusive, and Attributed 45

Interpreting Attributed Metrics: An Example 46

How Recursion Affects Function-Level Metrics 47

3. Collecting Performance Data 49

Compiling and Linking Your Program 49

Source Code Information 50

Static Linking 50

Optimization 50

Compiling Java Programs 51

Preparing Your Program for Data Collection and Analysis 51

Using Dynamically Allocated Memory 51

Using System Libraries 52

Using Signal Handlers 53

Using `setuid` 54

Program Control of Data Collection 54

The C, C++, Fortran, and Java API Functions 57

Dynamic Functions and Modules 59

Limitations on Data Collection 60

Limitations on Clock-Based Profiling 60

Limitations on Collection of Tracing Data 61

Limitations on Hardware Counter Overflow Profiling 62

Runtime Distortion and Dilation With Hardware Counter Overflow Profiling 62

Limitations on Data Collection for Descendant Processes 63

Limitations on Java Profiling 63

Runtime Performance Distortion and Dilation for Applications Written in the Java Programming Language 64

Where the Data Is Stored 65

Experiment Names 65

Moving Experiments 66

Estimating Storage Requirements 67

Collecting Data 68

Collecting Data Using the `collect` Command 69

Data Collection Options 69

Experiment Control Options 73

Output Options 76

Other Options 77

Collecting Data Using the `dbx collector` Subcommands 78

Data Collection Subcommands 79

Experiment Control Subcommands 82

Output Subcommands 83

Information Subcommands 84

Collecting Data From a Running Process 85

Collecting Data From MPI Programs 87

Storing MPI Experiments 88

Running the `collect` Command Under MPI 90

Collecting Data by Starting `dbx` Under MPI 90

Using `collect` With `ppgsz` 91

4. The Performance Analyzer Tool 93

Starting the Performance Analyzer 93

Analyzer Options 94

Performance Analyzer GUI 96

The Menu Bar 96

Toolbar 96

Analyzer Data Displays 96

Setting Data Presentation Options 104

Finding Text and Data 105

Showing or Hiding Functions 105

Filtering Data 106

Experiment Selection 106

Sample Selection 106

Thread Selection 106

LWP Selection 107

CPU Selection 107

Recording Experiments 107

Generating Mapfiles and Function Reordering 108

Defaults 108

5. Kernel Profiling 109

Kernel Experiments 109

Setting Up Your System for Kernel Profiling 109

Running the `er_kernel` Utility 110

Profiling the Kernel 110

Profiling Under Load 111

Profiling the Kernel and Load Together 112

Profiling a Specific Process or Kernel Thread 113

Analyzing a Kernel Profile 113

6. The `er_print` Command Line Performance Analysis Tool 115

`er_print` Syntax 116

Metric Lists 116

Commands That Control the Function List 119

Commands That Control the Callers-Callees List 122

Commands That Control the Leak and Allocation Lists 124

Commands That Control the Source and Disassembly Listings 124

Commands That Control the Data Space List 128

Commands That List Experiments, Samples, Threads, and LWPs 129

Commands That Control Selections 130

Commands That Control Load Object Selection 132

Commands That List Metrics 133

Commands That Control Output 133

Commands That Print Other Displays 134

Commands That Set Defaults 135

Commands That Set Defaults Only For the Performance Analyzer 137

Miscellaneous Commands 138

Examples 139

7. Understanding the Performance Analyzer and Its Data 141

How Data Collection Works 141

Experiment Format 142

Recording Experiments 144

Interpreting Performance Metrics 145

Clock-Based Profiling 145

Synchronization Wait Tracing 148

Hardware Counter Overflow Profiling 149

Heap Tracing 150

Dataspace Profiling 150

MPI Tracing 151
Call Stacks and Program Execution 151
Single-Threaded Execution and Function Calls 152
Explicit Multithreading 155
Overview of Java Technology-Based Software Execution 156
Java Processing Representations 158
Parallel Execution and Compiler-Generated Body Functions 161
Incomplete Stack Unwinds 165
Mapping Addresses to Program Structure 167
The Process Image 167
Load Objects and Functions 167
Aliased Functions 168
Non-Unique Function Names 168
Static Functions From Stripped Shared Libraries 169
Fortran Alternate Entry Points 169
Cloned Functions 170
Inlined Functions 170
Compiler-Generated Body Functions 171
Outline Functions 172
Dynamically Compiled Functions 172
The <Unknown> Function 173
The <JVM-System> Function 173
The <no Java callstack recorded> Function 173
The <Truncated-stack> Function 174
The <Total> Function 174
Functions Related to Hardware Counter Overflow Profiling 174
Mapping Data Addresses to Program Data Objects 175
Data Object Descriptors 176

8. Understanding Annotated Source and Disassembly Data 179

Annotated Source Code 179

Performance Analyzer Source Tab Layout 180

Annotated Disassembly Code 189

Interpreting Annotated Disassembly 190

Special Lines in the Source, Disassembly and PCs Tabs 194

Outline Functions 194

Compiler-Generated Body Functions 195

Dynamically Compiled Functions 196

Java Native Functions 198

Cloned Functions 199

Static Functions 200

Inclusive Metrics 201

Branch Target 201

Viewing Source/Disassembly Without An Experiment 202

9. Manipulating Experiments 205

Manipulating Experiments 205

Copying Experiments With the `er_cp` Utility 205

Moving Experiments With the `er_mv` Utility 206

Deleting Experiments With the `er_rm` Utility 206

Other Utilities 207

The `er_archive` Utility 207

The `er_export` Utility 208

A. Profiling Programs With `prof`, `gprof`, and `tcov` 209

Using `prof` to Generate a Program Profile 210

Using `gprof` to Generate a Call Graph Profile 212

Using `tcov` for Statement-Level Analysis 215

Creating `tcov` Profiled Shared Libraries 218

Locking Files 219

Errors Reported by `tcov` Runtime Functions 219

Using `tcov` Enhanced for Statement-Level Analysis 221

Creating Profiled Shared Libraries for `tcov` Enhanced 222

Locking Files 222

`tcov` Directories and Environment Variables 223

Index 225

Figures

FIGURE 2-1 Call Tree Illustrating Exclusive, Inclusive, and Attributed Metrics 46

FIGURE 7-1 Schematic Call Tree for a Multithreaded Program That Contains a Parallel Do or Parallel For Construct 163

FIGURE 7-2 Schematic Call Tree for a Parallel Region With a Worksharing Do or Worksharing For Construct 164

Tables

TABLE 2-1 Solaris Timing Metrics 35

TABLE 2-2 Synchronization Wait Tracing Metrics 40

TABLE 2-3 Memory Allocation (Heap Tracing) Metrics 41

TABLE 2-4 MPI Tracing Metrics 42

TABLE 2-5 Classification of MPI Functions Into Send, Receive, Send and Receive, and Other 43

TABLE 3-1 Parameter List for `collector_func_load()` 59

TABLE 3-2 Environment Variable Settings for Preloading the Library `libcollector.so` 87

TABLE 5-1 Field Label Meanings for Kernel Experiments in the Analyzer 113

TABLE 6-1 Metric Type Characters 117

TABLE 6-2 Metric Visibility Characters 117

TABLE 6-3 Metric Name Strings 118

TABLE 6-4 Compiler Commentary Message Classes 126

TABLE 6-5 Additional Options for the `dcc` Command 127

TABLE 6-6 Timeline Display Mode Options 137

TABLE 6-7 Timeline Display Data Types 137

TABLE 7-1 Data Types and Corresponding File Names 142

TABLE 7-2 How Kernel Microstates Contribute to Metrics 146

TABLE 8-1 Annotated Source-Code Metrics 188

TABLE A-1 Performance Profiling Tools 209

Before You Begin

This manual describes the performance analysis tools in the Sun™ Studio 10 software.

- The Collector and Performance Analyzer are a pair of tools that perform statistical profiling of a wide range of performance data and tracing of various system calls, and relate the data 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, C++, or the Java™ programming language; and some understanding of the Solaris™ Operating System (Solaris OS), or the Linux operating system, and UNIX® operating system commands. Some knowledge of performance analysis is helpful but is not required to use the tools.

How This Book Is Organized

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

Chapter 2 describes the data collected by the Collector and how the data is converted into metrics of performance.

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

Chapter 4 describes how to start the Performance Analyzer and how to use the tool to analyze performance data collected by the Collector.

Chapter 6 describes how to use the `er_print` command line interface to analyze the data collected by the Collector.

Chapter 8 describes how to use and understand the information in the source and disassembly windows of the Performance Analyzer.

Chapter 7 describes the process of converting the data collected by the Collector into performance metrics and how the metrics are related to program structure.

Chapter 9 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

TABLE P-1 Typeface 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*.

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*`n`*`]`	`O4, O`
{ }	Braces contain a set of choices for a required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*`dir`*`[:`*`dir`*`]`	`R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`xinline=`*`f1`*`[,…`*`fn`*`]`	`xinline=alpha,dos`

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at `http://www.sun.com/bigadmin/hcl`. These documents cite any implementation differences between the platform types.

In this document, the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the hardware compatibility lists.

Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell and Korn shell	$
Superuser for Bourne shell and Korn shell	#

Accessing Sun Studio Software and Man Pages

The compilers and tools and their man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the compilers and tools, you must have your `PATH` environment variable set correctly (see "Accessing the Compilers and Tools" on page 19). To access the man pages, you must have the your `MANPATH` environment variable set correctly (see "Accessing the Man Pages" on page 20.).

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` variable and `MANPATH` variables to access this release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Sun Studio compilers and tools are installed in the `/opt` directory on Solaris platforms and in the `/opt/sun` directory on Linux platforms. If your software is not installed in the `/opt` directory on Solaris platforms and in the default directory, ask your system administrator for the equivalent path on your system.

Accessing the Compilers and Tools

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

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

1. **Display the current value of the `PATH` variable by typing the following at a command prompt.**

```
% echo $PATH
```

2. **On Solaris platforms, review the output to find a string of paths that contain** `/opt/SUNWspro/bin/`**. On Linux platforms, review the output to find a string of paths that contain** `/opt/sun/sunstudio10/bin`**.**

   If you find the path, your `PATH` variable is already set to access the compilers and tools. If you do not find the path, set your `PATH` environment variable by following the instructions in the next procedure.

To Set Your `PATH` Environment Variable to Enable Access to the 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. **On Solaris platforms, add the following to your** `PATH` **environment variable. If you have Forte Developer software, Sun ONE Studio software, or another release of Sun Studio software installed, add the following path before the paths to those installations:**

   `/opt/SUNWspro/bin`

   On Linux platforms, add the following to your `PATH` **environment variable:**

   `/opt/sun/sunstudio10/bin`

Accessing the Man Pages

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

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

1. **Request the** `collect` **man page by typing the following at a command prompt.**

```
% man collect
```

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

   If the `collect`(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 procedure for setting your `MANPATH` environment variable.

To Set Your MANPATH Environment Variable to Enable Access to the Man Pages

- **On Solaris platforms, add the following to your `MANPATH` environment variable:**

  `/opt/SUNWspro/man`

- **On Linux platforms, add the following to your `MANPATH` environment variable:**

  `/opt/sun/sunstudio10/man`

Accessing the Integrated Development Environment

The Sun Studio integrated development environment (IDE) provides modules for creating, editing, building, debugging, and analyzing the performance of a C, C++, Java, or Fortran application.

The command to start the IDE is `sunstudio`. For details on this command, see the `sunstudio`(1) man page.

The correct operation of the IDE depends on the IDE being able to find the core platform. The `sunstudio` command looks for the core platform in two locations:

- The command looks first in the default installation directory, `/opt/netbeans/3.5V` on Solaris platforms and `/opt/sun/netbeans/3.5V` on Linux platforms.
- If the command does not find the core platform in the default directory, it assumes that the directory that contains the IDE and the directory that contains the core platform are both installed in or mounted to the same location. For example, on Solaris platforms, if the path to the directory that contains the IDE is `/foo/SUNWspro`, the command looks for the core platform in `/foo/netbeans/3.5V`. On Linux platforms, if the path to the directory that contains the IDE is `/foo/sunstudio10`, the command looks for the core platform in `/foo/netbeans/3.5V`.

If the core platform is not installed or mounted to either of the locations where the `sunstudio` command looks for it, then each user on a client system must set the environment variable `SPRO_NETBEANS_HOME` to the location where the core platform is installed or mounted (*/installation_directory*`/netbeans/3.5V`).

On Solaris platforms, each user of the IDE also must add */installation_directory*`/SUNWspro/bin` to their `$PATH` in front of the path to any other release of Forte Developer software, Sun ONE Studio software or Sun Studio software.

On Linux platforms, each user of the IDE also must add */installation_directory*`/sunstudio10/bin` to their `$PATH` in front of the path to any other release of Sun Studio software.

The path */installation_directory*`/netbeans/3.5V/bin` should not be added to the user's `$PATH`.

Accessing Sun Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms, and at `file:/opt/sun/sunstudio10/docs/index.html` on Linux platforms.

  If your software is not installed in the `/opt` directory on a Solaris platform or the `/opt/sun` directory on a Linux platform, ask your system administrator for the equivalent path on your system.
- Most manuals are available from the `docs.sun.com`[sm] web site. The following titles are available through your installed software only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
- The release notes are available from the `docs.sun.com` web site.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialogs, in the IDE.

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 that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed software on Solaris platforms through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms, and at `file:/opt/sun/sunstudio10/docs/index.html` on Linux platforms.
Online help	HTML available through the Help menu in the IDE or analyzer
Release notes	HTML at `http://docs.sun.com`

Related Compilers and Tools Documentation

For Solaris platforms, the following table describes related documentation that is available at `/opt/SUNWspro/docs/index.html` and at `http://docs.sun.com`. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
C User's Guide	Describes the Sun Studio 10 C programming language compiler along with ANSI C compiler-specific information.
C++ User's Guide	Instructs you in the use of the Sun Studio 10 C++ compiler and provides detailed information on command-line compiler options.
Fortran User's Guide	Describes the compile-time environment and command-line options for the Sun Studio 10 Fortran compiler.
OpenMP API User's Guide	Information on compiler directives used to parallelize programs.
Fortran Programming Guide	Discusses programming techniques, including parallelization, optimization, creation of shared libraries.
Debugging a Program With `dbx`	Reference manual for use of the debugger. Provides information on attaching and detaching to Solaris™ processes, and executing programs in a controlled environment.
Performance Analyzer Readme	Lists new features, and known problems, limitations, and incompatibilities of the Performance Analyzer.
`analyzer`(1), `collect`(1), `collector`(1), `er_print`(1), `er_src`(1), and `libcollector`(3) man pages	Describe Performance Analyzer command-line utilities

For Linux platforms, the following table describes related documentation that is available at `file:/opt/sun/sunstudio10/docs/index.html` at `http://docs.sun.com`. If your software is not installed in the `/opt/sun` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Performance Analyzer Readme	Lists new features, and known problems, limitations, and incompatibilities of the Performance Analyzer.
`analyzer`(1), `collect`(1), `collector`(1), `er_print`(1), `er_src`(1), and `libcollector`(3) man pages	Describe Performance Analyzer command-line utilities
Debugging a Program With `dbx`	Reference manual for use of the debugger. Provides information on attaching and detaching to Solaris processes, and executing programs in a controlled environment.

Accessing Related Solaris Documentation

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

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris Operating System.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.

Document Collection	Document Title	Description
Solaris 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.
Solaris Software Developer Collection	*SPARC Assembly Language Reference Manual*	Describes the assembly language for SPARC® processors.
Solaris 9 Update Collection	*Solaris Tunable Parameters Reference Manual*	Provides reference information on Solaris tunable parameters.

Resources for Developers

Visit http://developers.sun.com/prodtech/cc to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of compilers and tools components, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

You can find additional resources for developers at http://developers.sun.com.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

http://www.sun.com/service/contacting

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL

http://www.sun.com/hwdocs/feedback

Please include the part number (819-0493-11) of your document.

CHAPTER 1

Overview of the Performance Analyzer

Developing high performance applications requires a combination of compiler features, libraries of optimized functions, and tools for performance analysis. The *Performance Analyzer* manual 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.

Starting the Performance Analyzer From the Integrated Development Environment

For information on starting the Performance Analyzer from the Integrated Development Environment (IDE), see the Performance Analyzer Readme, which is available through the documentation index at */installation_directory*`/docs/index.html`. The default installation directory on Solaris platforms is `/opt/SUNWspro`. The default installation directory on Linux platforms is `/opt/sun/sunstudio10`. If the Sun Studio 10 compilers and tools are not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

The Tools of Performance Analysis

This manual deals primarily with the Collector and Performance Analyzer, a pair of tools that you use to collect and analyze performance data for your application. Both tools can be used from the command line or from a graphical user interface.

The 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`.

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 instructions are responsible for resource consumption?
- How did the program arrive at this point in the execution?
- Which resources are being consumed by a function or load object?

The Collector Tool

The Collector tool collects performance data using a statistical method called profiling and by tracing function calls. The data can include call stacks, microstate accounting information, thread synchronization delay data, hardware counter overflow data, Message Passing Interface (MPI) function call data, memory allocation data, and summary information for the operating system and the process. The Collector can collect all kinds of data for C, C++ and Fortran programs, and it can collect profiling data for applications written in the Java™ programming language. It can collect data for dynamically-generated functions and for descendant processes. See Chapter 2 for information about the data collected and Chapter 3 for detailed information about the Collector. The Collector can be run from the Performance Analyzer GUI, from the IDE, from the `dbx` command line tool, and using the `collect` command.

The Performance Analyzer Tool

The Performance Analyzer tool displays the data recorded by the 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 instructions. These metrics are classed into five groups:

- Clock profiling metrics
- Hardware counter metrics
- Synchronization delay metrics
- Memory allocation metrics
- MPI tracing metrics

The Performance Analyzer also displays the raw data in a graphical format as a function of time. The Performance Analyzer can create a mapfile that you can use to change the order of function loading in the program's address space, to improve performance.

See Chapter 4 and the online help in the IDE or the Performance Analyzer GUI for detailed information about the Performance Analyzer, and Chapter 6 for information about the command-line analysis tool, `er_print`.

Chapter 5 describes how you can use the Sun Studio performance tools to profile the kernel while the Solaris™ Operating System (Solaris OS) is running a load.

Chapter 7 discusses topics related to understanding the performance analyzer and its data, including: how data collection works, interpreting performance metrics, call stacks and program execution, and annotated code listings. Annotated source code listings and disassembly code listings that include compiler commentary but do not include performance data can be viewed with the `er_src` utility (see Chapter 9 for more information).

Chapter 8 provides an understanding of the annotated source and disassembly, providing explanations about the different types of index lines and compiler commentary that the Performance Analyzer displays.

Chapter 9 describes how to copy, move, delete, archive, and export experiments.

The `er_print` Utility

The `er_print` utility presents in plain text all the displays that are presented by the Performance Analyzer, with the exception of the Timeline display.

The `prof`, `gprof`, and `tcov` Tools

This manual also includes information about the following performance tools:

- `prof` and `gprof`

  The `prof` utility and the `gprof` utility are UNIX® tools for generating profile data and are included with the Solaris 7 OS, Solaris 8 OS, Solaris 9 OS, and Solaris 10 OS on SPARC® based systems and x86 based systems.

- `tcov`

  The `tcov` utility 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 the `prof` utility, `gprof` utility, and `tcov` utility, see Appendix A.

The Performance Analyzer Window

Note – The following is a brief overview of the Performance Analyzer window. See Chapter 4 and the online help for a complete and detailed discussion of the functionality and features of the tabs discussed below.

The Performance Analyzer window consists of a multi-tabbed display, with a menu bar and a toolbar. The tab that is displayed when the Performance Analyzer is started shows 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 lightweight process (LWP), by CPU, and by time slice.

For a selected function, another tab displays the callers and callees of the function. This tab can be used to navigate the call tree—in search of high metric values, for example.

Two other tabs 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 performance data is displayed as a function of time in another tab.

Other tabs show details of the experiments and load objects, summary information for a function, memory leaks, and statistics for the process.

You can navigate the Performance Analyzer from the keyboard as well as with a mouse.

CHAPTER 2

Performance Data

The performance tools work by recording data about specific events while a program is running, and converting the data into measurements of program performance called metrics.

This chapter describes the data collected by the performance tools, how it is processed and displayed, and how it can be used for performance analysis. Because there is more than one tool that collects performance data, the term Collector is used to refer to any of these tools. Likewise, because there is more than one tool that analyzes performance data, the term analysis tools is use to refer to any of these tools.

This chapter covers the following topics.

- What Data the Collector Collects
- How Metrics Are Assigned to Program Structure

See Chapter 3 for information on collecting and storing performance data.

See Chapter 4 for information on analyzing performance data with the performance analyzer.

See Chapter 5 for information on profiling the kernel while the Solaris OS is running a load.

See Chapter 6 for information on analyzing performance data with the `er_print` utility

What Data the Collector Collects

The Collector collects three different kinds of data: profiling data, tracing data and global data.

- Profiling data is collected by recording profile events at regular intervals. The interval is either a time interval obtained by using the system clock or a number of hardware events of a specific type. When the interval expires, a signal is delivered to the system and the data is recorded at the next opportunity.
- Tracing data is collected by interposing a wrapper function on various system functions so that calls to the system functions can be intercepted and data recorded about the calls.
- Global data is collected by calling various system routines to obtain information. The global data packet is called a sample.

Both profiling data and tracing data contain information about specific events, and both types of data are converted into performance metrics. Global data is not converted into metrics, but is used to provide markers that can be used to divide the program execution into time segments. The global data gives an overview of the program execution during that time segment.

The data packets collected at each profiling event or tracing event include the following information:

- A header identifying the data
- A high-resolution timestamp
- A thread ID
- A lightweight process (LWP) ID
- A processor (CPU) ID, when available from the operating system
- A copy of the call stack. For Java programs, two callstacks are recorded: the machine call stack and the Java call stack.

For more information on threads and lightweight processes, see Chapter 7.

In addition to the common data, each event-specific data packet contains information specific to the data type. The five types of data that the Collector can record are:

- Clock profile data
- Hardware counter overflow profiling data (Solaris OS only)
- Synchronization wait tracing data (Solaris OS only)
- Heap tracing (memory allocation) data
- MPI tracing data (Solaris OS only)

These five data types, the metrics that are derived from them, and how you might use them, are described in the following subsections. A sixth type of data, global sampling data, cannot be converted to metrics because it does not include call stack information.

Clock Data

When you are doing clock-based profiling, the data collected depends on the metrics provided by the operating system.

Clock-based Profiling Under the Solaris OS

In clock-based profiling under the Solaris OS, 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 default profiling interval is approximately 10 milliseconds (10 ms). The Collector provides a high-resolution profiling interval of approximately 1 ms and a low-resolution profiling interval of approximately 100 ms., and, where the OS permits, allows arbitrary intervals. Running the `collect` command with no arguments prints the range and resolution allowable on the system on which it is run.

The metrics that are computed from clock-based data are defined in the following table.

TABLE 2-1 Solaris Timing Metrics

Metric	Definition
User CPU time	LWP time spent running in user mode on the CPU.
Wall time	LWP time spent in LWP 1. This is usually the “wall clock time”
Total LWP time	Sum of all LWP times.
System CPU time	LWP time spent running in kernel mode on the CPU or in a trap state.
Wait CPU time	LWP time spent waiting for a CPU.
User lock time	LWP time spent waiting for a lock.

TABLE 2-1 Solaris Timing Metrics *(Continued)*

Metric	Definition
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 kernel page, or time 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 user lock time tells you that threads are unable to obtain the lock that they request.
- 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" in the Performance Analyzer online help) 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.

Clock-based Profiling Under the Linux OS

Under the Linux OS, the only metric available is User CPU time. Although the total CPU utilization time reported is accurate, it may not be possible for the Analyzer to determine the proportion of the time that is actually System CPU time as accurately as for the Solaris OS. Although the Analyzer displays the information as if the data were for a lightweight process (LWP), in reality there are no LWP's on a Linux OS; the displayed LWP ID is actually the thread ID.

Hardware Counter Overflow Profiling Data

Hardware counters keep track of events like cache misses, cache stall cycles, floating-point operations, branch mispredictions, CPU cycles, and instructions executed. In hardware counter overflow profiling, the Collector records a profile packet when a designated hardware counter of the CPU on which an LWP is running overflows. The counter is reset and continues counting. The profile packet includes the overflow value and the counter type.

Various CPU families support from two to eighteen simultaneous hardware counter registers. The Collector can collect data on one or more registers. For each register the 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 any 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 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. On machines with multiple CPUs, the clock frequency used to convert the metrics is the harmonic mean of the clock frequencies of the individual CPUs. Because each type of processor has its own set of hardware counters, and because the number of hardware counters is large, the hardware counter metrics are not listed here. The next subsection tells you how to find out what hardware counters are available.

One use of hardware counters is to diagnose problems with the flow of information into and out of the CPU. 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.

Some of the hardware counters provide similar or related information. For example, branch mispredictions and instruction cache misses are often related because a branch misprediction causes the wrong instructions to be loaded into the instruction cache, and these must be replaced by the correct instructions. The replacement can cause an instruction cache miss, or an instruction translation lookaside buffer (ITLB) miss, or even a page fault.

Hardware counter overflows are often delivered one or more instructions after the instruction which caused the event and the corresponding event counter to overflow: this is referred to as “skid” and it can make counter overflow profiles difficult to interpret. In the absence of hardware support for precise identification of the causal instruction, an apropos backtracking search for a candidate causal instruction may be attempted.

When such backtracking is supported and specified during collection, hardware counter profile packets additionally include the PC (program counter) and EA (effective address) of a candidate memory-referencing instruction appropriate for the

hardware counter event. (Subsequent processing during analysis is required to validate the candidate event PC and EA.) This additional information about memory-referencing events facilitates various data-oriented analyses.

Hardware Counter Lists

Hardware counters are processor-specific, so the choice of counters available to you depends on the processor that you are using. 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 on any particular system from the Collector by typing `collect` with no arguments in a terminal window on that system. If the processor and system support hardware counter profiling, the `collect` command prints two lists containing information about hardware counters. The first list contains "well-known" (aliased) hardware counters; the second list contains raw hardware counters.

Here is an example that shows the entries in the counter list. The counters that are considered well-known are displayed first in the list, followed by a list of the raw hardware counters. Each line of output in this example is formatted for print.

```
Well known HW counters available for profiling:
cycles[/{0|1}],9999991 ('CPU Cycles', alias for Cycle_cnt; CPU-cycles)
insts[/{0|1}],9999991 ('Instructions Executed', alias for Instr_cnt; events)
dcrm[/1],100003 ('D$ Read Misses', alias for DC_rd_miss; load events)
...
Raw HW counters available for profiling:
Cycle_cnt[/{0|1}],1000003 (CPU-cycles)
Instr_cnt[/{0|1}],1000003 (events)
DC_rd[/0],1000003 (load events)
```

Format of the Well-Known Hardware Counter List

In the well-known hardware counter list, the first field (for example, `cycles`) gives the alias name that can be used in the `-h counter...` argument of the `collect` command. This alias name is also the identifier to use in the `er_print` command.

The second field lists the available registers for the counter; for example, `[/{0|1}]`. For well-known counters, the default value has been chosen to provide a reasonable sample rate. Because actual rates vary considerably, you might need to specify a non-default value.

The third field, for example, `9999991`, is the default overflow value for the counter.

The fourth field, in parentheses, contains type information. It provides a short description (for example, `CPU Cycles`), the raw hardware counter name (for example, `Cycle_cnt`), and the type of units being counted (for example, `CPU-cycles`), which can include up to two words.

If the first word of type information is:

- `load`, `store`, or `load-store`, the counter is memory-related. You can prepend a + sign to the counter name (for example, `+dcrm`) in the `collect -h` command, to request a search for the precise instruction and virtual address that caused the event. The + sign also enables dataspace profiling; see "The DataObjects Tab" on page 101 and "The DataLayout Tab" on page 101 for details.
- `not-program-related`, the counter captures events initiated by some other program, such as CPU-to-CPU cache snoops. Using the counter for profiling generates a warning and profiling does not record a call stack.

If the second or only word of the type information is:

- `CPU-cycles`, the counter can be used to provide a time-based metric. The metrics reported for such counters are converted by default to inclusive and exclusive times, but can optionally be shown as event counts.
- `events`, the metric is inclusive and exclusive event counts, and cannot be converted to a time.

In the well-known hardware counter list in the example, the type information contains one word, `CPU-cycles` for the first counter and `events` for the second counter. For the third counter, the type information contains two words, `load events`.

Format of the Raw Hardware Counter List

The information included in the raw hardware counter list is a subset of the information in the well-known hardware counter list. Each line includes the internal counter name as used by `cpu-track`(1), the register number(s) on which that counter can be used, the default overflow value, and the counter units, which can be either `CPU-cycles` or `Events`.

If the counter measures events unrelated to the program running, the first word of type information is `not-program-related`. For such a counter, profiling does not record a call stack, but instead shows the time being spent in an artificial function, `collector_not_program_related`. Thread and LWP ID's are recorded, but are meaningless.

The default overflow value for raw counters is 1000003. This value is not ideal for most raw counters, so you should specify timeout values when specifying raw counters.

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 are collected by tracing calls to the Solaris or pthread thread functions. 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. Currently, synchronization wait tracing is only available for systems running the Solaris OS.

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.

Synchronization tracing for Java programs is based on events generated when a thread attempts to acquire a Java Monitor. Both machine and Java call stacks are collected for these events, but no synchronization tracing data is collected for internal locks used within the JVM™. In the machine representation, thread synchronization devolves into calls to `_lwp_mutex_lock`, and no synchronization data is shown, since these calls are not traced.

Synchronization wait tracing data is converted into the following metrics:

TABLE 2-2 Synchronization Wait Tracing Metrics

Metric	Definition
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 blocked, 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.

Heap Tracing (Memory Allocation) Data

Calls to memory allocation and deallocation functions that are not properly managed can be a source of inefficient data usage and can result in poor program performance. In heap tracing, the Collector traces memory allocation and deallocation requests by interposing on the C standard library memory allocation functions `malloc`, `realloc`, `valloc`, and `memalign` and the deallocation function `free`. Calls to `mmap` are treated as memory allocations, which allows heap tracing events for Java memory allocations to be recorded. The Fortran functions `allocate` and `deallocate` call the C standard library functions, so these routines are also traced indirectly.

For Java programs, heap tracing data records all object allocation events (generated by the user code), and object deallocation events (generated by the garbage collector). In addition, any use of C/C++ memory-management functions, such as `malloc` and `free`, also generates events that are recorded. Those events may come from native code, or from the JVM machine itself. Heap profiling for Java programs does not work with version 1.5.0 or later of the JVM machine, and will not be supported in future releases.

In the machine representation, memory is allocated and deallocated by the JVM machine, typically in very large chunks. Memory allocation from the Java code is handled entirely by the JVM and its garbage collector using the C/C++ memory mapping function `mmap`.

Heap tracing data is converted into the following metrics:

TABLE 2-3 Memory Allocation (Heap Tracing) Metrics

Metric	Definition
Allocations	The number of calls to the memory allocation functions.
Bytes allocated	The sum of the number of bytes allocated in each call to the memory allocation functions.
Leaks	The number of calls to the memory allocation functions that did not have a corresponding call to a deallocation function.
Bytes leaked	The number of bytes that were allocated but not deallocated.

Collecting heap tracing data can help you identify memory leaks in your program or locate places where there is inefficient allocation of memory.

Another definition of memory leaks that is commonly used, such as in the `dbx` debugging tool, says a memory leak is a dynamically-allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. The definition of leaks used here includes this alternative definition, but also includes memory for which pointers do exist.

MPI Tracing Data

The Collector can collect data on calls to the Message Passing Interface (MPI) library. Currently, MPI tracing is only available for system running the Solaris OS. The functions for which data is collected are listed below.

`MPI_Allgather`	`MPI_Allgatherv`	`MPI_Allreduce`
`MPI_Alltoall`	`MPI_Alltoallv`	`MPI_Barrier`
`MPI_Bcast`	`MPI_Bsend`	`MPI_Gather`
`MPI_Gatherv`	`MPI_Irecv`	`MPI_Isend`
`MPI_Recv`	`MPI_Reduce`	`MPI_Reduce_scatter`
`MPI_Rsend`	`MPI_Scan`	`MPI_Scatter`
`MPI_Scatterv`	`MPI_Send`	`MPI_Sendrecv`
`MPI_Sendrecv_replace`	`MPI_Ssend`	`MPI_Wait`
`MPI_Waitall`	`MPI_Waitany`	`MPI_Waitsome`
`MPI_Win_fence`	`MPI_Win_lock`	

MPI tracing data is converted into the following metrics:

TABLE 2-4 MPI Tracing Metrics

Metric	Definition
MPI Receives	Number of receive operations in MPI functions that receive data
MPI Bytes Received	Number of bytes received in MPI functions
MPI Sends	Number of send operations in MPI functions that send data
MPI Bytes Sent	Number of bytes sent in MPI functions
MPI Time	Time spent in all calls to MPI functions
Other MPI Calls	Number of calls to other MPI functions

The number of bytes recorded as received or sent is the buffer size given in the call. This number might be larger than the actual number of bytes received or sent. In the global communication functions and collective communication functions, the number of bytes sent or received is the maximum number, assuming direct interprocessor communication and no optimization of the data transfer or re-transmission of the data.

The functions from the MPI library that are traced are listed in TABLE 2-5, categorized as MPI send functions, MPI receive functions, MPI send and receive functions, and other MPI functions.

TABLE 2-5 Classification of MPI Functions Into Send, Receive, Send and Receive, and Other

Category	Functions
MPI send functions	`MPI_Bsend`, `MPI_Isend`, `MPI_Rsend`, `MPI_Send`, `MPI_Ssend`
MPI receive functions	`MPI_Irecv`, `MPI_Recv`
MPI send and receive functions	`MPI_Allgather`, `MPI_Allgatherv`, `MPI_Allreduce`, `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Bcast`, `MPI_Gather`, `MPI_Gatherv`, `MPI_Reduce`, `MPI_Reduce_scatter`, `MPI_Scan`, `MPI_Scatter`, `MPI_Scatterv`, `MPI_Sendrecv`, `MPI_Sendrecv_replace`
Other MPI functions	`MPI_Barrier`, `MPI_Wait`, `MPI_Waitall`, `MPI_Waitany`, `MPI_Waitsome`, `MPI_Win_fence`, `MPI_Win_lock`

Collecting MPI tracing data can help you identify places where you have a performance problem in an MPI program that could be due to MPI calls. Examples of possible performance problems are load balancing, synchronization delays, and communications bottlenecks.

Global (Sampling) Data

Global data is recorded by the Collector in packets called sample packets. Each packet contains a header, a timestamp, execution statistics from the kernel such as page fault and I/O data, context switches, and a variety of page residency (working-set and paging) statistics. The data recorded in sample packets is global to the program and is not converted into performance metrics. The process of recording sample packets is called sampling.

Sample packets are recorded in the following circumstances:

- When the program stops for any reason in the Debugging window of the IDE or in `dbx`, such as at a breakpoint, if the option to do this is set
- At the end of a sampling interval, if you have selected periodic sampling. The sampling interval is specified as an integer in units of seconds. The default value is 1 second
- When you choose Debug → Performance Toolkit → Enable Collector and select the Periodic Samples checkbox in the Collector window

- When you use the `dbx collector sample record` command to manually record a sample
- At a call to `collector_sample`, if you have put calls to this routine in your code (see "Program Control of Data Collection" on page 54)
- When a specified signal is delivered, if you have used the `-l` option with the `collect` command (see the `collect`(1) man page)
- When collection is initiated and terminated
- When you pause collection with the `dbx collector pause` command (just berfore the pause) and when you resume collection with the `dbx collector resume` command(just after the resume)
- Before and after a descendant process is created

The performance tools use the data recorded in the sample packets to group the data into time periods, which are called samples. You can filter the event-specific data by selecting a set of samples, so that you see only information for these particular time periods. You can also view the global data for each sample.

The performance tools make no distinction between the different kinds of sample points. To make use of sample points for analysis you should choose only one kind of point to be recorded. In particular, if you want to record sample points that are related to the program structure or execution sequence, you should turn off periodic sampling, and use samples recorded when `dbx` stops the process, or when a signal is delivered to the process that is recording data using the `collect` command, or when a call is made to the Collector API functions.

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. If the information is available, 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 7 for a more detailed explanation of how this is done.

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 an inclusive metric came from calls from or to another function: they attribute metrics to another function.

For a function that only appears at the bottom of call stacks (a leaf function), the exclusive and inclusive metrics are the same.

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 recorded 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 sum 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 Attributed Metrics: An Example

Exclusive, inclusive and attributed metrics are illustrated in FIGURE 2-1, 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.

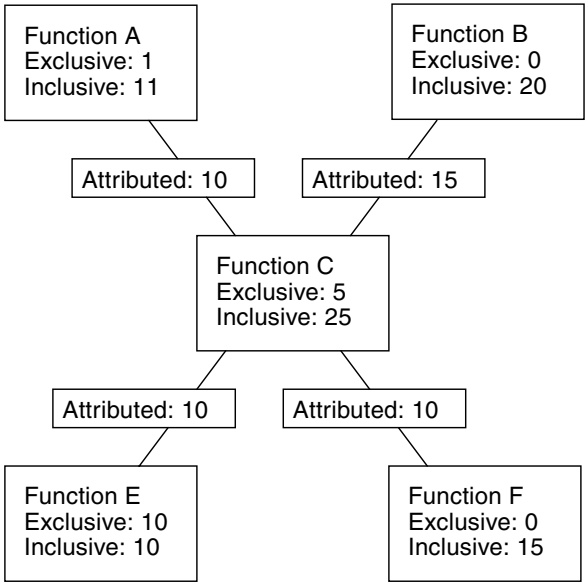

FIGURE 2-1 Call Tree Illustrating Exclusive, Inclusive, and Attributed Metrics

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).

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 metric for the event 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 metric for the event 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 is attributed to itself and not to the initiating function. This attribution occurs because the inclusive metric for all the higher invocations of the recursive function is 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.

CHAPTER 3

Collecting Performance Data

The first stage of performance analysis is data collection. This chapter describes what is required for data collection, where the data is stored, how to collect data, and how to manage the data collection. For more information about the data itself, see Chapter 2.

This chapter covers the following topics.

- Compiling and Linking Your Program
- Preparing Your Program for Data Collection and Analysis
- Limitations on Data Collection
- Where the Data Is Stored
- Estimating Storage Requirements
- Collecting Data
- Collecting Data Using the `collect` Command
- Collecting Data Using the `dbx collector` Subcommands
- Collecting Data From a Running Process
- Collecting Data From MPI Programs
- Using `collect` With `ppgsz`

Compiling and Linking Your Program

You can collect and analyze data for a program compiled with almost any option, but some choices affect what you can collect or what you can see in the Performance Analyzer. The issues that you should take into account when you compile and link your program are described in the following subsections.

Source Code Information

To see source code in annotated Source and Disassembly analyses, and source lines in the Lines analyses, you must compile the source files of interest with the `-g` compiler option (`-g0` for C++ to enable front-end inlining) to generate debug symbol information. The format of the debug symbol information can be either stabs or DWARF2, as specified by `-xdebugformat=(stabs|dwarf)`.

To prepare compilation objects with debug information that allows dataspace hardware counter profiles, currently only for the C compiler for SPARC® processors, compile by specifying `-xhwcprof -xdebugformat=dwarf` and any level of optimization. (Currently, this functionality is not available without optimization.) To see program data objects in Data Objects analyses, also add `-g` (or `-g0` for C++) to obtain full symbolic information.

Executables and libraries built with DWARF format debugging symbols automatically include a copy of each constituent object file's debugging symbols. Executables and libraries built with stabs format debugging symbols also include a copy of each constituent object file's debugging symbols if they are linked with the `-xs` option, which leaves stabs symbols in the various object files as well as the executable. The inclusion of this information is particularly useful if you need to move or remove the object files. With all of the debugging symbols in the executables and libraries themselves, it is easier to move the experiment and the program-related files to a new location.

Static Linking

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 Collector prints an error message and does not collect data. The error occurs because the collector library, among others, is dynamically loaded when you run the Collector.

You should not statically link any of the system libraries. If you do, you might not be able to collect any kind of tracing data. Nor should you link to the Collector library, `libcollector.so`.

Optimization

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. In addition, the call sequence can appear to be different from what you expect if the compiler performs tail-call optimizations. Optimization may cause unwind failures. See "Tail-Call Optimization" on page 155 for more information.

Compiling Java Programs

No special action is required for compiling Java programs with the `javac` command.

Preparing Your Program for Data Collection and Analysis

You do not need to do anything special to prepare most programs for data collection and analysis. You should read one or more of the subsections below if your program does any of the following:

- Installs a signal handler
- Explicitly dynamically loads a system library
- Dynamically compiles functions
- Creates descendant processes
- Uses the asynchronous I/O library
- Uses the profiling timer or hardware counter API directly
- Calls `setuid`(2) or executes a `setuid` file.

Also, if you want to control data collection from your program, you should read the relevant subsection.

Using Dynamically Allocated Memory

Many programs rely on dynamically-allocated memory, using features such as:

- `malloc`, `valloc`, `alloca` (C/C++)
- `new` (C++)
- stack local variables (Fortran)
- `MALLOC`, `MALLOC64` (Fortran)

You must take care to ensure that a program does not rely on the initial contents of dynamically allocated memory, unless the memory allocation method is explicitly documented as setting an initial value: for example, compare the descriptions of `calloc` and `malloc` in the man page for `malloc`(3C).

Occasionally, a program that uses dynamically-allocated memory might appear to work correctly when run alone, but might fail when run with performance data collection enabled. Symptoms might include unexpected floating point behavior, segmentation faults, or application-specific error messages.

Such behavior might occur if the uninitialized memory is, by chance, set to a benign value when the application is run alone, but is set to a different value when the application is run in conjunction with the performance data collection tools. In such cases, the performance tools are not at fault. Any application that relies on the contents of dynamically allocated memory has a latent bug: an operating system is at liberty to provide any content whatsoever in dynamically allocated memory, unless explicitly documented otherwise. Even if an operating system happens to always set dynamically allocated memory to a certain value today, such latent bugs might cause unexpected behavior with a later revision of the operating system, or if the program is ported to a different operating system in the future.

here are some tools that may help in finding such latent bugs:

- `f95 -xcheck=init_local`

  For more information, see the *Fortran User's Guide* or the `f95`(1) man page
- lint

  For more information, see the *C User's Guide* or the `lint`(1) man page
- Runtime checking under `dbx`

  For more information, see the *Debugging a Program With* `dbx` manual or the `dbx`(1) man page.
- Purify

Using System Libraries

The Collector interposes on functions from various system libraries, to collect tracing data and to ensure the integrity of data collection. The following list describes situations in which the Collector interposes on calls to library functions.

- Collecting synchronization wait tracing data. The Collector interposes on functions from the Solaris threads library, `libthread.so`[1] on the Solaris 8 OS and the Solaris 9 OS, and from the Solaris C library, `libc.so`, on the Solaris 10 OS. Synchronization wait tracing is not available on the Linux OS.

- Collecting heap tracing data. The Collector interposes on the functions `malloc`, `realloc`, `memalign` and `free`. Versions of these functions are found in the C standard library, `libc.so` and also in other libraries such as `libmalloc.so` and `libmtmalloc.so`.
- Collecting MPI tracing data. The Collector interposes on functions from the Solaris MPI library, `libmpi.so`. MPI tracing is not available under Linux.
- Ensuring the integrity of clock data. The Collector interposes on `setitimer` and prevents the program from using the profiling timer.
- Ensuring the integrity of hardware counter data. The Collector interposes on functions from the hardware counter library, `libcpc.so` and prevents the program from using the counters. Calls from the program to functions from this library return with a return value of `-1`.
- Enabling data collection on descendant processes. The Collector interposes on the functions `fork`(2), `fork1`(2), `vfork`(2), `fork`(3F), `system`(3C), `system`(3F), `sh`(3F), `popen`(3C), and `exec`(2) and its variants. Calls to `vfork` are replaced internally by calls to `fork1`. These interpositions are only done for the `collect` command.
- Guaranteeing the handling of the `SIGPROF` and `SIGEMT` signals by the Collector. The Collector interposes on `sigaction` to ensure that its signal handler is the primary signal handler for these signals.

Under some circumstances the interposition does not succeed:

- Statically linking a program with any of the libraries that contain functions that are interposed.
- Attaching `dbx` to a running application that does not have the collector library preloaded.
- Dynamically loading one of these libraries and resolving the symbols by searching only within the library.

The failure of interposition by the Collector can cause loss or invalidation of performance data.

Using Signal Handlers

The Collector uses two signals to collect profiling data: `SIGPROF` for all experiments and `SIGEMT` for hardware counter experiments only. The Collector installs a signal handler for each of these signals. The signal handler intercepts and processes its own

1. The default threads library, `/usr/lib/libthread.so`, on the Solaris OS8 (known as T1) has several problems when profiling. It may discard profiling interrupts when no thread is scheduled onto an LWP; in such cases, the Total LWP Time reported may seriously underestimate the true LWP time. Under some circumstances, it may also get a segmentation violation accessing an internal library mutex, causing the application to crash. The workaround is to use the alternate threads library (`/usr/lib/lwp/libthread.so`, known as T2), by prepending `/usr/lib/lwp` to your `LD_LIBRARY_PATH` setting. On Solaris 9, the default library is T2., and is incorporated into the libc library

signal, but passes other signals on to any other signal handlers that are installed. If a program installs its own signal handler for these signals, the Collector re-installs its signal handler as the primary handler to guarantee the integrity of the performance data.

The `collect` command can also use user-specified signals for pausing and resuming data collection and for recording samples. These signals are not protected by the Collector although a warning is written to the experiment if a user handler is installed. It is your responsibility to ensure that there is no conflict between use of the specified signals by the Collector and any use made by the application of the same signals.

The signal handlers installed by the Collector set a flag that ensures that system calls are not interrupted for signal delivery. This flag setting could change the behavior of the program if the program's signal handler sets the flag to permit interruption of system calls. One important example of a change in behavior occurs for the asynchronous I/O library, `libaio.so`, which uses `SIGPROF` for asynchronous cancel operations, and which does interrupt system calls. If the collector library, `libcollector.so`, is installed, the cancel signal invariably arrives too late to cancel the asynchronous I/O operation.

If you attach `dbx` to a process without preloading the collector library and enable performance data collection, and the program subsequently installs its own signal handler, the Collector does not re-install its own signal handler. In this case, the program's signal handler must ensure that the `SIGPROF` and `SIGEMT` signals are passed on so that performance data is not lost. If the program's signal handler interrupts system calls, both the program behavior and the profiling behavior are different from when the collector library is preloaded.

Using `setuid`

Restrictions enforced by the dynamic loader make it difficult to use `setuid`(2) and collect performance data. If your program calls `setuid` or executes a `setuid` file, it is likely that the Collector cannot write an experiment file because it lacks the necessary permissions for the new user ID.

Program Control of Data Collection

If you want to control data collection from your program, the Collector shared library, `libcollector.so` contains some API functions that you can use. The functions are written in C. A Fortran interface is also provided. Both C and Fortran interfaces are defined in header files that are provided with the library.

The API functions are defined as follows.

```
void collector_sample(char *name);
void collector_pause(void);
void collector_resume(void);
void collector_thread_pause(unsigned int t);
void collector_thread_resume(unsigned int t);
void collector_terminate_expt(void);
```

Similar functionality is provided for Java™ programs by the `CollectorAPI` class, which is described in "The Java Interface" on page 56.

The C and C++ Interface

There are two ways to access the C and C++ interface:

- One way is to include `collectorAPI.h` and link with `-lcollectorAPI` (which contains real functions to check for the existence of the underlying `libcollector.so` API functions).

  This way requires that you link with an API library, and works under all circumstances. If no experiment is active, the API calls are ignored.

- The second way is to include `libcollector.h` (which contains macros that check for the existence of the underlying `libcollector.so` API functions).

  This way works when used in the main executable, and when data collection is started at the same time the program starts. This way does not always work when `dbx` is used to attach to the process, nor when used from within a shared library that is `dlopen`'d by the process. This second way is provided for backward compatibility.

Caution – Do not link a program in any language with `-lcollector`. If you do, the Collector can exhibit unpredictable behavior.

The Fortran Interface

The Fortran API `libfcollector.h` file defines the Fortran interface to the library. The application must be linked with `-lcollectorAPI` to use this library. (An alternate name for the library, `-lfcollector`, is provided for backward compatibility.) The Fortran API provides the same features as the C and C++ API, excluding the dynamic function and thread pause and resume calls.

Insert the following statement to use the API functions for Fortran:.

```
include "libfcollector.h"
```

Caution – Do not link a program in any language with `-lcollector`. If you do, the Collector can exhibit unpredictable behavior.

The Java Interface

Use the following statement to import the CollectorAPI class and access the Java™ API. Note however that your application must be invoked with a classpath pointing to */installation_directory*`/lib/collector.jar` where *installation-directory* is the directory in which the Sun Studio software is installed.

```
import com.sun.forte.st.collector.CollectorAPI;
```

The Java™ CollectorAPI methods are defined as follows:

```
CollectorAPI.sample(String name)
CollectorAPI.pause()
CollectorAPI.resume()
CollectorAPI.threadPause(Thread thread)
CollectorAPI.threadResume(Thread thread)
CollectorAPI.terminate()
```

The Java API includes the same functions as the C and C++ API, excluding the dynamic function API.

The C include file `libcollector.h` contains macros that bypass the calls to the real API functions if data is not being collected. In this case the functions are not dynamically loaded. However, using these macros is risky because the macros do not work well under some circumstances. It is safer to use `collectorAPI.h` because it does not use macros. Rather, it refers directly to the functions.

The Fortran API subroutines call the C API functions if performance data is being collected, otherwise they return. The overhead for the checking is very small and should not significantly affect program performance.

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

If you intend to use the API functions in a multithreaded program, you should ensure that they are only called by one thread. With the exception of `collector_thread_pause()` and `collector_thread_resume()`, the API functions perform actions that apply to the process and not to individual threads. If each thread calls the API functions, the data that is recorded might not be what you expect. For example, if `collector_pause()` or `collector_terminate_expt()` is called by one thread before the other threads have reached the same point in the program, collection is paused or terminated for all threads, and data can be lost from the threads that were executing code before the API call. To control data collection at the level of the individual threads, use the `collector_thread_pause()` and `collector_thread_resume()` functions. There are two viable ways of using these functions: by having one master thread make all the calls for all threads, including itself; or by having each thread make calls only for itself. Any other usage can lead to unpredictable results.

The C, C++, Fortran, and Java API Functions

The descriptions of the API functions follow.

- **C and C++**: `collector_sample(char *name)`

  Fortran: `collector_sample(string name)`

  Java: `CollectorAPI.sample(String name)`

  Record a sample packet and label the sample with the given name or string. The label is displayed by the Performance Analyzer in the Event tab. The Fortran argument `string` is of type `character`.

  Sample points contain data for the process and not for individual threads. In a multithreaded application, the `collector_sample()` API function ensures that only one sample is written if another call is made while it is recording a sample. The number of samples recorded can be less than the number of threads making the call.

  The Performance Analyzer does not distinguish between samples recorded by different mechanisms. If you want to see only the samples recorded by API calls, you should turn off all other sampling modes when you record performance data.

- **C, C++, Fortran**: `collector_pause()`

  Java: `CollectorAPI.pause()`

  Stop writing event-specific data to the experiment. The experiment remains open, and global data continues to be written. The call is ignored if no experiment is active or if data recording is already stopped. This function stops the writing of all event-specific data even if it is enabled for specific threads by the `collector_thread_resume()` function.

- **C, C++, Fortran**: `collector_resume()`

  Java: `CollectorAPI.resume()`

  Resume writing event-specific data to the experiment after a call to `collector_pause()`. The call is ignored if no experiment is active or if data recording is active.

- **C and C++ only**: `collector_thread_pause(unsigned int t)`

  Java: `CollectorAPI.threadPause(Thread t)`

  Stop writing event-specific data from the thread specified in the argument list to the experiment. The argument t is the POSIX thread identifier for C/C++ programs and a Java thread for Java programs. If the experiment is already terminated, or no experiment is active, or writing of data for that thread is already turned off, the call is ignored. This function stops the writing of data from the specified thread even if the writing of data is globally enabled. By default, recording of data for individual threads is turned on.

- **C and C++ only**: `collector_thread_resume(unsigned int t)`

  Java: `CollectorAPI.threadResume(Thread t)`

  Resume writing event-specific data from the thread specified in the argument list to the experiment. The argument `t` is the POSIX thread identifier for C/C++ programs and a Java thread for Java programs. If the experiment is already terminated, or no experiment is active, or writing of data for that thread is already turned on, the call is ignored. Data is written to the experiment only if the writing of data is globally enabled as well as enabled for the thread.

- **C, C++, Fortran**: `collector_terminate_expt()`

  Java: `CollectorAPI.terminate`

  Terminate the experiment whose data is being collected. No further data is collected, but the program continues to run normally. The call is ignored if no experiment is active.

Dynamic Functions and Modules

If your C program or C++ program dynamically compiles functions into the data space of the program, you must supply information to the Collector if you want to see data for the dynamic function or module in the Performance Analyzer. The information is passed by calls to collector API functions. The definitions of the API functions are as follows.

```
void collector_func_load(char *name, char *alias,
    char *sourcename, void *vaddr, int size, int lntsize,
    Lineno *lntable);
void collector_func_unload(void *vaddr);
```

You do not need to use these API functions for Java methods that are compiled by the Java HotSpot™ virtual machine, for which a different interface is used. The Java interface provides the name of the method that was compiled to the Collector. You can see function data and annotated disassembly listings for Java compiled methods, but not annotated source listings.

The descriptions of the API functions follow.

- `collector_func_load()`

Pass information about dynamically compiled functions to the Collector for recording in the experiment. The parameter list is described in the following table.

TABLE 3-1 Parameter List for `collector_func_load()`

Parameter	Definition
`name`	The name of the dynamically compiled function that is used by the performance tools. The name does not have to be the actual name of the function. The name need not follow any of the normal naming conventions of functions, although it should not contain embedded blanks or embedded quote characters.
`alias`	An arbitrary string used to describe the function. It can be `NULL`. It is not interpreted in any way, and can contain embedded blanks. It is displayed in the Summary tab of the Analyzer. It can be used to indicate what the function is, or why the function was dynamically constructed.
`sourcename`	The path to the source file from which the function was constructed. It can be `NULL`. The source file is used for annotated source listings.
`vaddr`	The address at which the function was loaded.

TABLE 3-1 Parameter List for `collector_func_load()` *(Continued)*

Parameter	Definition
`size`	The size of the function in bytes.
`lntsize`	A count of the number of entries in the line number table. It should be zero if line number information is not provided.
`lntable`	A table containing `lntsize` entries, each of which is a pair of integers. The first integer is an offset, and the second entry is a line number. All instructions between an offset in one entry and the offset given in the next entry are attributed to the line number given in the first entry. Offsets must be in increasing numeric order, but the order of line numbers is arbitrary. If `lntable` is `NULL`, no source listings of the function are possible, although disassembly listings are available.

- `collector_func_unload()`

Inform the collector that the dynamic function at the address `vaddr` has been unloaded.

Limitations on Data Collection

This section describes the limitations on data collection that are imposed by the hardware, the operating system, the way you run your program, or by the Collector itself.

There are no limitations on simultaneous collection of different data types: you can collect any data type with any other data type.

Limitations on Clock-Based Profiling

The minimum value of the profiling interval and the resolution of the clock used for profiling depend on the particular operating environment. The maximum value is set to 1 second. The value of the profiling interval is rounded down to the nearest multiple of the clock resolution. The minimum and maximum value and the clock resolution can be found by typing the `collect` command with no arguments.

The system clock is used for profiling in early versions of the Solaris 8 OS. It has a resolution of 10 milliseconds, unless you choose to enable the high-resolution system clock. 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
```

In the Solaris 9 OS, the Solaris10 OS, and later versions of the Solaris 8 OS, it is not necessary to enable the high-resolution system clock for high-resolution profiling.

Runtime Distortion and Dilation with Clock-profiling

Clock-based profiling records data when a SIGPROF signal is delivered to the target. It causes dilation to process that signal, and unwind the call stack. The deeper the call stack, and the more frequent the signals, the greater the dilation. To a limited extent, clock-mased profiling shows some distortion, deriving from greater dilation for those parts of the program executing with the deepest stacks.

Limitations on Collection of Tracing Data

You cannot collect any kind of tracing data from a program that is already running unless the Collector library, `libcollector.so`, had been preloaded. See "Collecting Data From a Running Process" on page 85 for more information.

Runtime Distortion and Dilation with Tracing

Tracing data dilates the run in proportion to the number of events that are traced. If done with clock-based profiling, the clock data is distorted by the dilation induced by tracing events.

Limitations on Hardware Counter Overflow Profiling

Hardware counter overflow profiling has several limitations:

- You can only collect hardware counter overflow data on processors that have hardware counters and that support overflow profiling. On other systems, hardware counter overflow profiling is disabled. UltraSPARC® processors prior to the UltraSPARC® III processor family do not support hardware counter overflow profiling.
- You cannot collect hardware counter overflow data with versions of the Solaris OS that precede the Solaris 8 release.
- 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. If `cpustat` is started during data collection, the hardware counter overflow profiling is terminated and an error is recorded in the experiment.
- You cannot use the hardware counters in your own code with the `libcpc`(3) API if you are doing hardware counter overflow profiling. The Collector interposes on the `libcpc` library functions and returns with a return value of `-1` if the call did not come from the Collector.
- If you try to collect hardware counter data on a running program that is using the hardware counter library by attaching `dbx` to the process, the experiment may be corrupted.

Note – To view a list of all available counters, run the `collect` command with no arguments.

Runtime Distortion and Dilation With Hardware Counter Overflow Profiling

Hardware counter overflow profiling records data when a SIGEMT is delivered to the target. It causes dilation to process that signal, and unwind the call stack. Unlike clock-based profiling, for some hardware counters, different parts of the program might generate events more rapidly than other parts, and show dilation in that part of the code. Any part of the program that generates such events very rapidly might be significantly distorted. Similarly, some events might be generated in one thread disproportionately to the other threads.

Limitations on Data Collection for Descendant Processes

You can collect data on descendant processes subject to the following limitations:

- If you want to collect data for all descendant processes that are followed by the Collector, you must use the `collect` command with the `-F on` or `-F all` option. With the `-F` option set to `on`, you can collect data automatically for calls to `fork` and its variants and `exec` and its variants. With the `-F` option set to `all`, the Collector follows all descendant processes, including those due to calls to `system`, `popen`, and `sh`.
- If you want to collect data for individual descendant processes, you must attach `dbx` to the process. See Appendix "Collecting Data From a Running Process" on page 85 for more information.
- If you want to collect data for individual descendant processes, you must use a separate `dbx` to attach to each process and enable the collector. Note, however, that data collection for Java programs is not currently supported under `dbx`.

Limitations on Java Profiling

You can collect data on Java programs subject to the following limitations:

- You should use a version of the Java™ 2 Software Development Kit (J2SDK) no earlier than 1.4.2_02. Do not use versions 1.4.2 or 1.4.2_01, which may crash. By default, the `collect` command uses the path where the J2SDK was installed by the Sun Studio installer, if any. You can override this default path by setting either the `JDK_HOME` environment variable or the `JAVA_PATH` environment variable.The Collector verifies that the version of `java` it finds in these environment variables is an ELF executable, and if it is not, an error message is printed, indicating which environment variable was used, and the full path name that was tried.
- You must use the `collect` command to collect data. You cannot use the `dbx collector` subcommands or the data collection capabilities of the IDE.
- Applications that create descendant processes that run JVM software cannot be profiled.
- If you want to use the 64-bit JVM software, you must use the `-j on` flag and specify the 64-bit JVM as the target. Do not use `java -d64` to collect data using the 64-bit JVM software. If you do so, no data is collected.
- A non-Java application can dynamically open `libjvm.so` and pass it a Java class. A workaround to support profiling for such an application, is to not set `-j` to `on` for the initial collect invocation, and to ensure that the `-Xruncollector` option is passed to the invoked JVM software.

Using JVM versions earlier than 1.4.2_02 compromises the data as follows:

- **JVM 1.4.2_01**: This version of the JVM software might crash during data collection.
- **JVM 1.4.2**: This version of the JVM software might crash during data collection.
- **JVM 1.4.1**: The Java representation is correctly recorded and shown, but all JVM housekeeping is shown as the JVM functions themselves. Some of the time spent executing JVM code in data space is shown with names for the code regions as supplied by the JVM software. A significant amount of time is shown in the `<Unknown>` function, since some of the code regions created by the JVM software are not named. In addition, various bugs in JVM 1.4.1 might cause the program being profiled to crash.
- **JVM 1.4.0**: No Java representation is possible, and a significant amount of time is shown in `<Unknown>`.
- **JVMs earlier than 1.4.0**: Profiling Java applications with JVM software earlier than version1.4.0 is not supported.

Runtime Performance Distortion and Dilation for Applications Written in the Java Programming Language

Java profiling uses the Java™ Virtual Machine Profiling Interface (JVMPI) if you are running J2SDK 1.4.2, or the Java™ Virtual Machine Tools Interface (JVMTI) if you are running J2SDK 1.5.0, which can cause some distortion and dilation of the run.

For clock-based profiling and hardware counter overflow profiling, the data collection process makes various calls into the JVM, and handles profiling events in signal handlers. The overhead of these routines, and the cost of writing the experiments to disk will dilate the runtime of the Java program. Such dilation is typically less than 10%.

Although the default garbage collector supports JVMPI, there are other garbage collectors that do not. Any data-collection run specifying such a garbage collector will get a fatal error.

For heap profiling, the data collection process uses JVMPI events describing memory allocation and garbage collection, which can cause significant dilation in runtime. Most Java applications generate many of these events, which leads to large experiments, and scalability problems processing the data. Furthermore, if these events are requested, the garbage collector disables some inlined allocations, costing additional CPU time for the longer allocation path.

For synchronization tracing, data collection uses other JVMTI events, which causes dilation in proportion to the amount of monitor contention in the application.

Where the Data Is Stored

The data collected during one execution of your application is called an experiment. The experiment consists of a set of files that are stored in a directory. The name of the experiment is the name of the directory.

In addition to recording the experiment data, the Collector creates its own 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. You can specify the storage location when you run the Collector.

Experiments for descendant processes are stored inside the experiment for the founder process.

Experiment Names

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 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 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 Collector prints an error message.

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 a special header line and an experiment name on each subsequent 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.

You can create an experiment group file by creating a plain text file whose first line is

```
#analyzer experiment group
```

and adding the names of the experiments on subsequent lines. The name of the file must end in `.erg`.

The default experiment 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 default experiment name is *group*`.`*m*`.er`. If you specify an experiment name, it overrides these defaults. See "Collecting Data From MPI Programs" on page 87 for more information.

Experiments for descendant processes are named with their lineage as follows. To form the experiment name for a descendant process, an underscore, a code letter and a number are added to the stem of its creator's experiment name. The code letter is `f` for a fork, x for an exec, and c for combination. The number is the index of the fork or exec (whether successful or not). For example, if the experiment name for the founder process is `test.1.er`, the experiment for the child process created by the third call to `fork` is `test.1.er/_f3.er`. If that child process calls `exec` successfully, the experiment name for the new descendant process is `test.1.er/_f3_x1.er`.

Moving Experiments

If you want to move an experiment to another computer to analyze it, you should be aware of the dependencies of the analysis on the operating environment in which the experiment was recorded.

The archive files contain all the information necessary to compute metrics at the function level and to display the timeline. However, if you want to see annotated source code or annotated disassembly code, you must have access to versions of the load objects or source files that are identical to the ones used when the experiment was recorded.

The Performance Analyzer searches for the source, object and executable files in the following locations in turn, and stops when it finds a file of the correct basename:

- The archive directories of experiments.
- The current working directory.
- The absolute pathname as recorded in the executables or compilation objects.

You can change the search order or add other search directories from the Analyzer GUI or by using the `setpath` and `addpath` directives.

To ensure that you see the correct annotated source code and annotated disassembly code for your program, you can copy the source code, the object files and the executable into the experiment before you move or copy the experiment. If you don't want to copy the object files, you can link your program with `-xs` to ensure that the information on source lines and file locations are inserted into the executable. You can automatically copy the load objects into the experiment using the `-A` option of the `collect` command or the `dbx collector archive` command.

Estimating Storage Requirements

This section gives some guidelines for estimating the amount of disk space needed to record an experiment. The size of the experiment depends directly on the size of the data packets and the rate at which they are recorded, the number of LWPs used by the program, and the execution time of the program.

The data packets contain event-specific data and data that depends on the program structure (the call stack). The amount of data that depends on the data 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. Data packets are recorded for each LWP in the experiment. Note that for Java programs, there are two call stacks of interest: the Java call stack and the machine call stack, which therefore result in more data being written to disk.

The rate at which profiling data packets are recorded is controlled by the profiling interval for clock data and by the overflow value for hardware counter data. However, the choice of these parameters also affects the data quality and the distortion of program performance due to the data collection overhead. Smaller values of these parameters give better statistics but also increase the overhead. The default values of the profiling interval and the overflow value have been carefully chosen as a compromise between obtaining good statistics and minimizing the overhead. Smaller values also mean more data.

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 is 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 is 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.

Your estimate of the size of the experiment should also take into account the disk space used by the archive files, which is usually a small fraction of the total disk space requirement (see the previous section). If you are not sure how much space you need, try running your experiment for a short time. From this test 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 Collector allocates buffers in memory to store the profile data before writing it to disk. Currently no way exists to specify the size of these buffers. If the Collector runs out of memory, try to reduce the amount of data collected.

If your estimate of the space required to store the experiment is larger than the space you have available, consider collecting data for part of the run rather than the whole run. You can collect data on part of the run with the `collect` command, with the `dbx collector` subcommands, or by inserting calls in your program to the collector API. You can also limit the total amount of profiling and tracing data collected with the `collect` command or with the `dbx collector` subcommands.

Note – The Performance Analyzer cannot read more than 2 GB of performance data.

Collecting Data

You can collect performance data in either the standalone Performance Analyzer or the Analyzer window in the IDE in several ways:

- Using the `collect` command from the command line (see "Collecting Data Using the `collect` Command" on page 69 and the `collect`(1) man page). The `collect` command-line tool has smaller data collection overheads than `dbx` or the Collector dialog in the Debugger in the IDE so this method can be superior to the others.
- Using the Performance Tools Collect dialog in the Performance Analyzer (see "Collecting Performance Data Using the Performance Tools Collector" in the Performance Analyzer online help)
- Using the Collector dialog in the Debugger (see "Collecting Performance Data Using the Debugger" in the Performance Analyzer online help)
- Using the `collector` command from the `dbx` command line (see "Collecting Data Using the `dbx collector` Subcommands" on page 78 and "collector Command" in the Debugging online help in the IDE)

The following data collection capabilities are available only with the Performance Tools Collect dialog and the `collect` command:

- Collecting data on Java programs. If you try to collect data on a Java program with Collector Dialog in the Debugger in the IDE or with the `collector` command in `dbx`, the information that is collected is for the JVM software, not the Java program.
- Collecting data automatically on descendant processes.

Collecting Data Using the `collect` Command

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

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

Here, *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.

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 Profiling Data" on page 37. See also "Limitations on Hardware Counter Overflow Profiling" on page 62.

Data Collection Options

These options control the types of data that are collected. See "What Data the Collector Collects" on page 34 for a description of the data types.

If you do not specify data collection options , the default is `-p on`, which enables clock-based profiling with the default profiling interval of approximately 10 milliseconds. The default is turned off by the `-h` option but not by any of the other data collection options.

If you explicitly disable clock-based profiling is, and neither any kind of tracing nor hardware counter overflow profiling is enabled, the `collect` command prints a warning message, and collects global data only.

`-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 approximately 10 milliseconds.
- `lo[w]` – Turn on clock-based profiling with the low-resolution profiling interval of approximately 100 milliseconds.
- `hi[gh]` – Turn on clock-based profiling with the high-resolution profiling interval of approximately 1 millisecond. In the Solaris 8 OS, high-resolution profiling must be explicitly enabled. See "Limitations on Clock-Based Profiling" on page 60 for information on enabling high-resolution profiling.
- *value* – Turn on clock-based profiling and set the profiling interval to *value*. The default units for *value* are milliseconds. You can specify *value* as an integer or a floating-point number. The numeric value can optionally be followed by the suffix `m` to select millisecond units or `u` to select microsecond units. The value should be a multiple of the 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 clock resolution.

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

`-h` *counter_definition_1* `...[,`*counter_definition_n*`]`

Collect hardware counter overflow profiling data. The number of counter definitions is processor-dependent. A counter definition can take one of the following forms, depending on whether the processor supports attributes for hardware counters.

`[+]`*counter_name*`[/`*register_number*`][,`*interval*`]`

`[+]`*counter_name*`[~`*attribute_1*`=`*value_1*`]...[~`*attribute_n*`=`*value_n*`][/`*register_number*`][,`*interval*`]`

The processor-specific *counter_name* can be one of the following:

- Awell-known (aliased) counter name
- A raw (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 you specify more than one counter , they must use different registers. If they do not use different registers, the `collect` command prints an error message and exits. Some counters can count on either register.

To obtain a list of available counters, type `collect` with no arguments in a terminal window. A description of the counter list is given in the section "Hardware Counter Lists" on page 38.

If the hardware counter counts events that relate to memory access, youcan perfix the counter name with a + sign to turn on searching for the true PC of the instruction that caused the counter overflow. If the search is successful, the PC and effective address that was referenced are stored in the event data packet.

On some processors, attribute options can be associated with a hardware counter. If a processor supports attribute options, then running the `collect` command with no arguments lists the counter definitions including the attribute names. You can specify attribute values in decimal or hexadecimal format.

The interval (overflow value) is the number of events counted at which the hardware counter overflows and the overflow event is recorded. The interval can be set to one of the following:

- `on`, or a null string – The default overflow value, which you can determine by typing `collect` with no arguments.
- `hi[gh]` – The high-resolution value for the chosen counter, which is approximately ten times shorter than the default overflow value. The abbreviation `h` is also supported for compatibility with previous software releases.
- `lo[w]` – The low-resolution value for the chosen counter, which is approximately ten times longer than the default overflow value.
- *interval* – A specific overflow value, which must be a positive integer and can be in decimal or hexadecimal format.

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 62.

If you use the `-h` option without explicitly specifying a `-p` option, clock-based profiling is turned off. To collect both hardware counter data and clock-based data, you must specify both a `-h` option and a `-p` option.

`-s` *option*

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

- `all` – Enable synchronization wait tracing with a zero threshold. This option forces all synchronization events to be recorded.

- `calibrate` – Enable synchronization wait tracing and set the threshold value by calibration at runtime. (Equivalent to `on`.)
- `off` – Disable synchronization wait tracing.
- `on` – Enable 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.

Synchronization wait tracing data is not recorded for Java monitors.

`-H` *option*

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

- `on` – Turn on tracing of heap allocation and deallocation requests.
- `off` – Turn off heap tracing.

Heap tracing is turned off by default.

`-m` *option*

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

- `on` – Turn on tracing of MPI calls.
- `off` – Turn off tracing of MPI calls.

MPI tracing is turned off by default.

See "MPI Tracing Data" on page 42 for more information about the MPI functions whose calls are traced and the metrics that are computed from the tracing data.

`-S` *option*

Record sample packets periodically. The allowed values of *option* are:

- `off` – Turn off periodic sampling.
- `on` – Turn on periodic sampling with the default sampling interval of 1 second.
- *value* – Turn on periodic sampling and set the sampling interval to *value*. The interval value must be positive, and is given in seconds.

By default, periodic sampling at 1 second intervals is enabled.

Experiment Control Options

-F *option*

Control whether or not descendant processes should have their data recorded. The allowed values of *option* are:

- `on` – Record experiments only on descendant processes that are created by functions `fork`, `exec`, and their variants.
- `all` – Record experiments on all descendant processes.
- `off` – Do not record experiments on descendant processes.

If you specify the `-F on` option, the Collector follows processes created by calls to the functions `fork`(2), `fork1`(2), `fork`(3F), `vfork`(2), and `exec(2)` and its variants. The call to `vfork` is replaced internally by a call to `fork1`.

If you specify the `-F all` option, the Collector follows all descendant processes including those created by calls to `system`(3C), `system`(3F), `sh`(3F), and `popen`(3C), and similar functions, and their associated descendant processes.

If you specify the `-F on` or `-F all` argument, the Collector opens a new experiment for each descendant process inside the founder experiment. These new experiments are named by adding an underscore, a letter, and a number to the experiment suffix, as follows:

- The letter is either an "f" to indicate a fork, an "x" to indicate an exec, or "c" to indicate any other descendant process.
- The number is the index of the fork or exec (whether successful or not) or other call.

For example, if the experiment name for the initial process is `test.1.er`, the experiment for the child process created by its third fork is `test.1.er/_f3.er`. If that child process execs a new image, the corresponding experiment name is `test.1.er/_f3_x1.er`. If that child creates another process using a popen call, the experiment name is `test.1.er/_f3_x1_c1.er`.

The Analyzer andthe `er_print`utility automatically read experiments for descendant processes when the founder experiment is read, but the experiments for the descendant processes are not selected for data display.

To select the data for display from the command line, specify the path name explicitly to either `er_print` or `analyzer`. The specified path must include the founder experiment name, and descendant experiment name inside the founder directory.

For example, here's what you specify to see the data for the third fork of the `test.1.er` experiment:

```
er_print test.1.er/_f3.er

analyzertest.1.er/_f3.er
```

Alternatively, you can prepare an experiment group file with the explicit names of the descendant experiments in which you are interested.

To examine descendant processes in the Analyzer, load `the` founder experiment and select `Filter Data` from the View menu. A list of experiments is displayed with only the founder experiment checked. Uncheck it and check the descendant experiment of interest.

`-j` *option*

Enable Java profiling when the target is a JVM machine. The allowed values of *option* are:

- `on` — Recognize methods compiled by the Java HotSpot virtual machine, and attempt to record Java call stacks.
- `off`— Do not attempt to recognize methods compiled by the Java HotSpot virtual machine.
- *path* — Record profiling data for the JVM machine installed in the specified *path*.

The `-j` option is not needed if you want to collect data on a `.class` file or a `.jar` file, provided that the path to the `java` executable is in either the `JDK_HOME` environment variable or the `JAVA_PATH` environment variable. You can then specify *program* as the `.class` file or the `.jar` file, with or without the extension.

If you cannot define the path to `java` in any of these variables, or if you want to disable the recognition of methods compiled by the Java HotSpot virtual machine you can use this option. If you use this option, *program* must be a Java virtual machine whose version is not earlier than 1.4.2_02. The `collect` command verifies that *program* is a JVM machine, and is an ELF executable; if it is not, the `collect` command prints an error message.

If you want to collect data using the 64-bit JVM machine, you must not use the `-d64` option to `java` for a 32-bit JVM machine. If you do so, no data is collected. Instead you must specify the path to the 64-bit JVM machine either in *program* or in one of the environment variables given in this section.

-J *java_arguments*

Specify arguments to be passed to the JVM machine used for profiling. If you specify the -J option, but do not specify Java profiling, an error is generated, and no experiment is run.

-l *signal*

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

You can specify the signal 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.

If you use this option and your program has its own signal handler, you should make sure that the signal that you specify with -l is passed on to the Collector's signal handler, and is not intercepted or ignored.

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, use the dbx commands ignore PROF and ignore EMT to 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 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 Collector is started in the recording state.

If you use this option and your program has its own signal handler, make sure that the signal that you specify with `-y` is passed on to the Collector's signal handler, and is not intercepted or ignored.

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

Output Options

`-o` *experiment_name*

Use *experiment_name* as the name of the experiment to be recorded. The *experiment_name* string must end in the string ".er"; if not, the `collect` utility prints an error message and exits.

`-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` utility 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` utility prints an error message and exits. If the group exists, the experiment is added to it. If *group-name* is not an absolute path, the experiment group is placed in the directory *directory-name* if a directory has been specified with `-d`, otherwise it is placed in the current directory.

`-A` *option*

Control whether or not load objects used by the target process should be archived or copied into the recorded experiment. The allowed values of option are:

- `off` – do not archive load objects into the experiment.
- `on` – archive load objects into the experiment.
- `copy` – copy and archive load objects into the experiment.

If you expect to copy experiments to a different machine from which they were recorded, or to read the experiments from a different machine, specify `-A copy`. Using this option does not copy any source files or object files into the experiment. You should ensure that those files are accessible on the machine to which you are copying the experiment.

`-L` *size*

Limit the amount of profiling data recorded to *size* megabytes. The limit applies to the sum of the amounts of clock-based profiling data, hardware counter overflow profiling data, and synchronization wait tracing data, but not to sample points. The limit is only approximate, and can be exceeded.

When the limit is reached, no more profiling data is recorded but the experiment remains open until the target process terminates. If periodic sampling is enabled, sample points continue to be written.

The default limit on the amount of data recorded is 2000 Mbytes. This limit was chosen because the Performance Analyzer cannot process experiments that contain more than 2 Gbytes of data. To remove the limit, set *size* to `unlimited` or `none`.

`-O` *file*

Append all output from `collect` itself to the name *file*, but do not redirect the output from the spawned target. If file is set to `/dev/null`, suppress all output from `collect`, including any error messages.

Other Options

`-C` *comment*

Put the *comment* into the `notes` file for the experiment. You can supply up to ten -C options. The contents of the notes file are prepended to the experiment header.

`-n`

Do not run the target but print the details of the experiment that would be generated if the target were run. This option is a dry run option.

`-R`

Display the text version of the Performance Analyzer Readme in the terminal window. If the readme is not found, a warning is printed. No further arguments are examined, and no further processing is done.

`-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 Using the `dbx collector` Subcommands

To run the 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 Collector. They are ignored with a warning if an experiment is active.

`profile` *option*

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

- `on` – Enables clock-based profiling with the default profiling interval of 10 ms.
- `off` – Disables clock-based profiling.
- `timer` *interval* – Sets the profiling interval. The allowed values of *interval* are
  - `on` – Use the default profiling interval of approximately 10 milliseconds.
  - `lo[w]` – Use the low-resolution profiling interval of approximately 100 milliseconds.
  - `hi[gh]` – Use the high-resolution profiling interval of approximately 1 millisecond. In the earlier versions of the Solaris 8 Operating System, high-resolution profiling must be explicitly enabled. See "Limitations on Clock-Based Profiling" on page 60 for information on enabling high-resolution profiling.
  - *value* – Set the profiling interval to *value*. The default units for *value* are milliseconds. You can specify *value* as an integer or a floating-point number. The numeric value can optionally be followed by the suffix `m` to select millisecond units or `u` to select microsecond units. The value should be a multiple of the clock resolution. If the value is larger than the clock resolution but not a multiple it is rounded down. If the value is smaller than the clock resolution it is set to the clock resolution. In both cases a warning message is printed.

  The default setting is approximately 10 milliseconds.

The Collector collects clock-based profiling data by default, unless the collection of hardware-counter overflow profiling data is turned on using the `hwprofile` subcommand.

`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 Lists" on page 38 for a description of the list. If your system does not support hardware counter overflow profiling, `dbx` returns a warning message.
- `counter` *counter_definition*...`[ ,` *counter_definition* `]` – A counter definition can take one of the following forms, depending on whether the processor supports attributes for hardware counters.

  `[+]`*counter_name*`[/`*register_number*`][ ,`*interval*`]`

  [+]*counter_name*[*~attribute_1=value_1*]...[*~attribute_n= value_n*]`[/`*register_number*`][ ,`*interval*`]`

  Selects the hardware counter *name*, and sets its overflow value to *interval*; optionally selects additional hardware counter names and sets their overflow values to the specified intervals. The overflow value can be one of the following.

  - `on`, or a null string – The default overflow value, which you can determine by typing `collect` with no arguments.
  - `hi[gh]` – The high-resolution value for the chosen counter, which is approximately ten times shorter than the default overflow value. The abbreviation `h` is also supported for compatibility with previous software releases.
  - `lo[w]` – The low-resolution value for the chosen counter, which is approximately ten times longer than the default overflow value.
  - *interval* – A specific overflow value, which must be a positive integer and can be in decimal or hexadecimal format.

  If you specify more than one counter, they must use different registers. If they do not, a warning message is printed and the command is ignored.

  If the hardware counter counts events that relate to memory access, you can prefix the counter name with a + sign to turn on searching for the true PC of the instruction that caused the counter overflow. If the search is successful, the PC and the effective address that was referenced are stored in the event data packet.

The Collector does not collect hardware counter overflow profiling data by default. If hardware-counter overflow profiling is enabled and a `profile` command has not been given, clock-based profiling is turned off.

See also "Limitations on Hardware Counter Overflow Profiling" on page 62.

`synctrace` *option*

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

- `on` – Enable synchronization wait tracing with the default threshold.
- `off` – Disable synchronization wait tracing.
- `threshold` *value* – Sets the threshold for the minimum synchronization delay. The allowed values for *value* are:
  - `all` – Use a zero threshold. This option forces all synchronization events to be recorded.
  - `calibrate` – Set the threshold value by calibration at runtime. (Equivalent to `on`.)
  - `off` – Turn off synchronization wait tracing.
  - `on` – Use the default threshold, which is to set the value by calibration at runtime. (Equivalent to `calibrate`.)
  - *number* – Set the threshold to *number*, given as a positive integer in microseconds. If `value` is 0, all events are traced.

By default, the Collector does not collect synchronization wait tracing data.

`heaptrace` *option*

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

- `on` – Enables heap tracing.
- `off` – Disables heap tracing.

By default, the Collector does not collect heap tracing data.

`mpitrace` *option*

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

- `on` – Enables tracing of MPI calls.
- `off` – Disables tracing of MPI calls.

By default, the Collector does not collect MPI tracing data.

`sample` *option*

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

- `periodic` – Enables periodic sampling.
- `manual` – Disables periodic sampling. Manual sampling remains enabled.
- `period` *value* – Sets the sampling interval to *value*, given in seconds.

By default, periodic sampling is enabled, with a sampling interval *value* of 1 second.

`dbxsample { on | off }`

Controls the recording of samples when `dbx` stops the target process. The meanings of the keywords are as follows:

- `on` – A sample is recorded each time `dbx` stops the target process.
- `off` – Samples are not recorded when `dbx` stops the target process.

By default, samples are recorded when `dbx` stops the target process.

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 enabled, 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 not recorded while the Collector is paused. A sample is generated prior to a pause, and another sample is generated immediately following a resume. 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 is being collected.

`sample record` *name*

Record a sample packet with the label *name*. The label is displayed in the Event tab of the Performance Analyzer.

Output Subcommands

The following subcommands define storage options for the experiment. They are ignored with a warning if an experiment is active.

`archive` *mode*

Set the mode for archiving the experiment. The allowed values for *mode* are

- `on` – normal archiving of load objects
- `off` – no archiving of load objexts
- `copy` – copy load objects into experiment in addition to normal archiving

If you intend to move the experiment to a different machine, or read it from another machine, you should enable the copying of load objects. If an experiment is active, the command is ignored with a warning. This command does not copy source files or object files into the experiment.

`limit` *value*

Limit the amount of profiling data recorded to *value* megabytes. The limit applies to the sum of the amounts of clock-based profiling data, hardware counter overflow profiling data, and synchronization wait tracing data, but not to sample points. The limit is only approximate, and can be exceeded.

When the limit is reached, no more profiling data is recorded but the experiment remains open and sample points continue to be recorded.

The default limit on the amount of data recorded is 2000 Mbytes. This limit was chosen because the Performance Analyzer cannot process experiments that contain more than 2 Gbytes of data. To remove the limit, set *value* to `unlimited` or `none`.

`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 and any experiment group is stored. This subcommand is ignored with a warning if the directory does not exist.
- `experiment` *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 65 for more information on experiment names and how the 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. If the directory name has been set using the `store directory` subcommand and the group name is not an absolute path, the group name is prefixed with the directory name.

Information Subcommands

`show`

Shows the current setting of every Collector control.

`status`

Reports on the status of any open experiment.

Collecting Data From a Running Process

The 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 IDE), you can pause the process and enable data collection using the methods described in previous sections.

Note – For information on starting the Performance Analyzer from the IDE, see the Performance Analyzer Readme, which is available through the documentaton index at */installation_directory*`/docs/index.html`. The default installation directory on Solaris platforms is `/opt/SUNWspro`. The default installation directory on Linux platforms is `/opt/sun/sunstudio10`. If the Sun Studio 10 software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

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. If you want to collect performance data for selected descendant processes, you must attach `dbx` to each process.

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 Debug menu of the IDE, choose Debug → Attach 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 manual, *Debugging a Program With* `dbx`, for more details on attaching to a process. Attaching to a running process pauses the process.

3. **Start data collection.**
   - From the Debug menu of the IDE, choose Performance Toolkit → Enable Collector and use the Collect dialog box to set up the data collection parameters. Then choose Debug → Continue to resume execution of the process.
   - From `dbx`, use the `collector` command to set up the data collection parameters and the `cont` command to resume execution of the process.
4. **Detach from the process.**

   When you have finished collecting data, pause the program and then detach the process from `dbx`.
   - In the IDE, right-click the session for the process in the Sessions view of the Debugger window and choose Detach from the contextual menu. If the Sessions view is not displayed, click the Sessions button at the top of the Debugger window.
   - From `dbx`, type the following.

```
(dbx) detach
```

If you want to collect any kind of tracing data, you must preload the Collector library, `libcollector.so`, before you run your program, because the library provides wrappers to the real functions that enable data collection to take place. In addition, the Collector adds wrapper functions to other system library calls to guarantee the integrity of performance data. If you do not preload the Collector library, these wrapper functions cannot be inserted. See "Using System Libraries" on page 52 for more information on how the Collector interposes on system library functions.

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 variables `LD_LIBRARY_PATH`, `LD_LIBRARY_PATH_32`, and/or `LD_LIBRARY_PATH_64` to set

the path to the library. (LD_LIBRARY_PATH is used if the _32 and _64 variants are not defined.) If you have already defined these environment variables, add new values to them.

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_PATH_32	/opt/SUNWspro/lib
LD_LIBRARY_PATH_64	/opt/SUNWspro/lib/v9

If your Sun Studio 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 Collector for each process. When you attach dbx to the processes in an MPI job, each process is 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 is 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 87.

Collecting Data From MPI Programs

The Collector can collect performance data from multi-process programs that use the Sun Message Passing Interface (MPI) library. The MPI library is included in the Sun HPC ClusterTools™ software. You should use the latest version (5.0) of the ClusterTools™ software if possible, but you can use 3.1 or a compatible version. To start the parallel jobs, use the Sun Cluster Runtime Environment (CRE) command mprun. See the Sun HPC ClusterTools documentation 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 Collector are implemented, each MPI process records a separate experiment. Each experiment must have a unique name. Where and how the experiment is stored depends on the kinds of file systems that are available to your MPI job. Issues about storing experiments are discussed in the next subsection.

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 a subsequent subsection.

Storing MPI Experiments

Because multiprocessing environments can be complex, you should be aware of some issues about storing MPI experiments 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 65 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 four 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 205 for information on how to copy and move experiments.

If you do not specify an experiment name, the Collector uses the MPI rank to construct an experiment name with the standard form *experiment*.*n*`.er`, but in this case *n* is 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 do 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. 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. Also make sure that the file system has enough space for the experiments. See "Estimating Storage Requirements" on page 67 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 source lines in the annotated disassembly code unless you have access to the load objects and source files 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 65 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 is labelled with the MPI rank. Read the section "Storing MPI Experiments" on page 88 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
```

Using `collect` With `ppgsz`

You can use `collect` with `ppgsz`(1) by running `collect` on the `ppgsz` command and specifying the `-F on` or `-F all` flag. The founder experiment is on the `ppgsz` executable and uninteresting. If your path finds the 32-bit version of `ppgsz`, and the experiment is run on a system that supports 64-bit processes, the first thing it will do is `exec` its 64-bit version, creating `_x1.er`. That executable forks, creating `_x1_f1.er`.

The child process attempts to `exec` the named target in the first directory on your path, then in the second, and so forth, until one of the `exec`s succeed. If, for example, the third attempt succeeds, the first two descendant experiments are named `_x1_f1_x1.er` and `_x1_f1_x2.er`, and both are completely empty. The experiment on the target is the one from the successful `exec`, the third one in the example, and is named `_x1_f1_x3.er`, stored under the founder experiment. It can be processed directly by invoking the Analyzer or the `er_print` utility on `test.1.er/_x1_f1_x3.er`.

If the 64-bit `ppgsz` is the initial process, or if the 32-bit `ppgsz` is invoked on a 32-bit kernel, the fork child that `exec`s the real target has its data in `_f1.er`, and the real target's experiment is in `_f1_x3.er`, assuming the same path properties as in the example above.

CHAPTER 4

The Performance Analyzer Tool

The Performance Analyzer is a graphical data-analysis tool that analyzes performance data collected by the Collector using the `collect` command, the IDE, or the `collector` commands in `dbx`. The Collector gathers performance information to create an experiment during the execution of a process, as described in Chapter 3. The Performance Analyzer reads in such experiments, analyzes the data, and displays the data in tabular and graphical displays. A command-line version of the Analyzer is available as the `er_print` utility, which is described in Chapter 6

Starting the Performance Analyzer

To start the Performance Analyzer, type the following on the command line:

```
% analyzer [control-options] [experiment-list]
```

Alternatively, use the Explorer in the IDE to navigate to an experiment and open it. The *experiment-list* command argument is a blank-separated list of experiment names, experiment group names, or both.

You can specify multiple experiments or experiment groups on the command line. If you specify an experiment that has descendant experiments inside it, all descendant experiments are automatically loaded, but the display of data for the descendant experiments is disabled. To load individual descendant experiments you must specify each experiment explicitly or create an experiment group. To create an experiment group, create a plain text file whose first line is as follows:

```
#analyzer experiment group
```

Then add the names of the experiments on subsequent lines. The file extension must be `erg`.

You can also use the File menu in the Analyzer window to add experiments or experiment groups. To open experiments recorded on descendant processes, you must type the file name in the Open Experiment dialog box (or Add Experiment dialog box) because the file chooser does not permit you to open an experiment as a directory.

When the Analyzer displays multiple experiments, however they were loaded, data from all the experiments is aggregated.

You can preview an experiment or experiment group for loading by single-clicking on its name in either the Open Experiment dialog or the Add Experiment dialog.

You can also start the Performance Analyzer from the command line to record an experiment as follows:

```
% analyzer [Java-options] [control-options] target [target-arguments]
```

The Analyzer starts up with the Performance Tools Collect window showing the named target and its arguments, and settings for collecting an experiment. See "Recording Experiments" on page 107 for details.

Analyzer Options

These options control the behavior of the Analyzer and are divided into three groups:

- Java options
- Control options
- Information options

Java Options

-j | --jdkhome *jvm-path*

Specify the path to the JVM software for running the Analyzer. The default path is taken first by first examining environment variables for a path to Java, in the order JDK_HOME and then JAVA_PATH. If neither environment variable is set, the default path is where the Java™2 Software Development Kit was installed by the Sun Studio installer, if any, and if not, as set by the user's PATH.

-J *jvm-options*

Specify the JVM options.

Control Options

-f | --fontsize *size*

Specify the font size to be used in the Analyzer GUI.

-v | --verbose

Print version information and Java runtime arguments before starting.

Information Options

These options do not invoke the Performance Analyzer GUI, but print information about `analyzer` to standard output. The individual options below are stand-alone options; they cannot be combined with other `analyzer` options nor combined with target or experiment-list arguments.

-V | --version

Print version information and Java runtime arguments before starting.

-? | --h | --help

Print usage information and exit.

Performance Analyzer GUI

The Analyzer window has a menu bar, a tool bar, and a split pane that contains tabs for the various data displays.

The Menu Bar

The menu bar contains a File menu, a View menu, a Timeline menu, and a Help menu.

The File menu is for opening, adding, and dropping experiments and experiment groups. The File menu allows you to collect data for an experiment using the Performance Analyzer GUI. For details on using the Performance Analyzer to collect data, refer to "Recording Experiments" on page 107. From the File menu, you can also create a mapfile, which is used to optimize the size of an executable or optimize its effective cache behavior. For more details on mapfiles, refer to "Generating Mapfiles and Function Reordering" on page 108.

The View menu allows you to configure how experiment data is displayed.

The Timeline menu, as its name suggests, helps you to navigate the timeline display, described in "Analyzer Data Displays" on page 96.

The Help menu provides online help for the Performance Analyzer, provides a summary of new features, has quick-reference and shortcut sections, and has a troubleshooting section.

Toolbar

The toolbar provides sets of icons as menu shortcuts, and includes a Find function to help you find functions when using the data displays. For more details about the Find function, refer to "Finding Text and Data" on page 105

Analyzer Data Displays

The Performance Analyzer uses a split-window to divide the data presentation into two panes. Each pane is tabbed to allow you to select different data displays for the same experiment or experiment group.

Data Display, Left Pane

The left pane contains tabs for the principal Analyzer displays:

- The Functions tab
- The Callers-Callees tab
- The Source tab
- The Lines tab
- The Disassembly tab
- The PCs tab
- The DataLayout tab
- The DataObjects tab
- The Timeline tab
- The Leaklist tab
- The Statistics tab
- The Experiments tab

If you invoke the Analyzer without a target, you are prompted for an experiment to open.

If dataspace-profiling data is recorded into the experiment being read, the Data Layout and Data Objects tabs will also show in addition to the tabs listed above.

The Functions Tab

The Functions tab shows a list consisting of functions and their metrics. The metrics are derived from the data collected in the experiment. Metrics can be either exclusive or inclusive. Exclusive metrics represent usage within the function itself. Inclusive metrics represent usage within the function and all the functions it called.

The list of available metrics for each kind of data collected is given in the `collect`(1) man page. Only the functions that have non-zero metrics are listed.

Time metrics are shown as seconds, presented to millisecond precision. Percentages are shown to a precision of 0.01%. If a metric value is precisely zero, its time and percentage is shown as "0." If the value is not exactly zero, but is smaller than the precision, its value is shown as "0.000" and its percentage as "0.00". Because of rounding, percentages may not sum to exactly 100%. Count metrics are shown as an integer count.

The metrics initially shown are based on the data collected and on the default settings read from various `.er.rc` files. When the Performance Analyzer is initially installed, the defaults are as follows:

- For clock-based profiling, the default set consists of inclusive and exclusive User CPU time.
- For synchronization delay tracing, the default set consists of inclusive synchronization wait count and inclusive synchronization time.

- For hardware counter overflow profiling, the default set consists of inclusive and exclusive times (for counters that count in cycles) or event counts (for other counters).
- For heap tracing, the default set consists of heap leaks and bytes leaked.

If more than one type of data has been collected, the default metrics for each type are shown.

The metrics that are shown can be changed or reorganized; see the online help for details.

To search for a function, use the `Find` tool in the toolbar. For further details about the Find tool, refer to "Finding Text and Data" on page 105.

The Callers-Callees Tab

The Callers-Callees tab shows the selected function in a pane in the center, with callers of that function in a pane above, and callees of that function in a pane below.

In addition to showing exclusive and inclusive metric values for each function, the tab also shows attributed metrics. For the selected function, the attributed metric represents the exclusive metric for that function. For the callees, the attribute metric represents the portion of the callee's inclusive metric that is attributable to calls from the center function. The sum of attributed metrics for the callees and the selected function will add up to the inclusive metric for the selected function.

For the callers, the attributed metrics represent the portion of the selected function's inclusive metric that is attributable to calls from the callers. The sum of the attributed metrics for all callers should also add up to the inclusive metric for the selected function.

The metrics shown in the Callers-Callees tab are can be changed or reorganized; see the online help for details.

Clicking once on a function in the caller or callee pane selects that function, causing the window contents to be redrawn so that the selected function appears in the center pane.

The Source Tab

If available, the Source tab shows the file containing the source code of the selected function, annotated with performance metrics for each source line. The full names of the source file, the corresponding object file and the load object are given in the column heading for the source code. In the rare case where the same source file is used to compile more than one object file, the Source tab shows the performance data for the object file containing the selected function.

The Analyzer looks for the file containing the selected function under the absolute pathname as recorded in the executable. If the file is not there, the Analyzer tries to find a file of the same basename in the current working directory. If you have moved the sources, or the experiment was recorded in a different file system, you can put a symbolic link from the current directory to the real source location in order to see the annotated source.

When you select a function in the Functions tab and the Source tab is opened, the source file displayed is the default source context for that function. The default source context of a function is the file containing the function's first instruction, which, for C code, is the function's opening brace. Immediately following the first instruction, the annotated source file adds an index line for the function. The source window displays index lines as text in red italics within angle brackets in the form:

.<Function: f_name>

A function might have an alternate source context, which is another file that contains instructions attributed to the function. Such instructions might come from include files or from other functions inlined into the selected function. If there are any alternate source contexts, the beginning of the default source context includes a list of extended index lines that indicate where the alternate source contexts are located.

<Function: f, instructions from source file src.h>

Double clicking on an index line that refers to another source context opens the file containing that source context, at the location associated with the indexed function.

To aid navigation, alternate source contexts also start with a list of index lines that refer back to functions defined in the default source context and other alternate source contexts.

The source code is interleaved with any compiler commentary that has been selected for display. The classes of commentary shown can be set in the Set Data Presentation dialog box. The default classes can be set in a defaults file.

The metrics displayed in the Source tab can be changed or reorganized; see the online help for details.

Lines with metrics that are equal to or exceed a threshold percentage of the maximum of that metric for any line in the source file are highlighted to make it easier to find the important lines. The threshold can be set in the Set Data Presentation dialog box. The default threshold can be set in a defaults file. Tick marks are shown next to the scrollbar, corresponding to the position of over-threshold lines within the source file. For example, if there were two over-threshold lines near the end of the source file, two ticks would be shown next to the scrollbar near the bottom of the source window. Positioning the scrollbar next to a tick mark will position the source lines displayed in the source window so that the corresponding over-threshold line is displayed.

The Lines Tab

The Lines tab shows a list consisting of source lines and their metrics. Source lines are labeled with the function from which they came and the line number and source file name. If no line-number information is available for a function, or the source file for the function is not known, all of the function's PCs appear aggregated into a single entry for the function in the lines display. PCs from functions that are from load-objects whose functions are hidden appear aggregated as a single entry for the load-object in the lines display. Selecting a line in the Lines tab shows all the metrics for that line in the Summary tab. Selecting the Source or Disassembly tab after selecting a line from the Lines tab positions the display at the appropriate line.

The Disassembly Tab

The Disassembly tab shows a disassembly listing of the object file containing the selected function, annotated with performance metrics for each instruction.

Interleaved within the disassembly listing is the source code, if available, and any compiler commentary chosen for display. The algorithm for finding the source file in the Disassembly tab is the same as the algorithm used in the Source tab.

Just as with the Source tab, index lines are displayed in Disassembly tab. But unlike with the Source tab, index lines for alternate source contexts cannot be used directly for navigation purposes. Also, index lines for alternate source contexts are displayed at the start of where the `#included` or inlined code is inserted, rather than just being listed at the beginning of the Disassembly view. Code that is `#included` or inlined from other files shows as raw disassembly instructions without interleaving the source code. However, placing the cursor on one of these instructions and selecting the Source tab opensthe source file containing the `#included` or inlined code. Selecting the Disassembly tab with this file displayed opens the Disassembly view in the new context, thus displaying the disassembly code with interleaved source code.

The classes of commentary shown can be set in the Set Data Presentation dialog box. The default classes can be set in a defaults file.

The Analyzer highlights lines with metrics that are equal to or exceed a metric-specific threshold, to make it easier to find the important lines. You can set the threshold in the Set Data Presentation dialog box. You can set the default threshold in a defaults file. As with the Source tab, tick marks are shown next to the scrollbar, corresponding to the position of over-threshold lines within the disassembly code.

The PCs Tab

The PCs tab shows a list consisting of PCs and their metrics. PCs are labeled with the function from which they came and the offset within that function. PCs from functions that are from load-objects whose functions are hidden appear aggregated as a single entry for the load-object in the PCs display. Selecting a line in the PCs tab shows all the metrics for that PC in the Summary tab. Selecting the Source tab or Disassembly tab after selecting a line from the PCs tab positions the display at the appropriate line.

The DataObjects Tab

The DataObjects Tab shows the list of data objects with their metrics. The tab is visible by default if dataspace data is recorded in the Formats tab of the Set Data Presentation dialog box, or setting a `datamode on` command in one of the `.er.rc` files read when the Analyzer starts. The tab is applicable only to hardware counter overflow experiments where the aggressive backtracking option was enabled, and for source files that were compiled with the `-xhwcprof` option in the C compiler.

When enabled, it shows hardware counter memory operation metrics against the various data structures and variables in the program.

The DataLayout Tab

The DataLayout tab shows the annotated data object layouts for all program data objects with data-derived metric data. The layouts appear in the order they are defined in the experiment's load objects. The tab shows each aggregate data object with the total metrics attributed to it, followed by all of its elements in offset order. Each element, in turn, has its own metrics and an indicator of its size and location in 32-byte blocks.

As with the DataObjects tab, the DataLayout tab is visible by default if dataspace data is recorded in the experiment. Also, the tab can be shown by turning Data Space Display to `on` in the Data Presentation Panel, or setting a `datamode on` command in one of the `er.rc` files read when the Analyzer starts.

The Timeline Tab

The Timeline tab shows a chart of the events and the sample points recorded by the Collector as a function of time. Data is displayed in horizontal bars. For each experiment there is a bar for sample data and a set of bars for each LWP. The set for an LWP consists of one bar for each data type recorded: clock-based profiling, hardware counter profiling, synchronization tracing, heap tracing and MPI tracing.

The bars that contain sample data show a color-coded representation of the time spent in each microstate for each sample. Samples are displayed as a period of time because the data in a sample point represents time spent between that point and the previous point. Clicking a sample displays the data for that sample in the Event tab.

The profiling data or tracing data bars show an event marker for each event recorded. The event markers consist of a color-coded representation of the call stack recorded with the event, as a stack of colored rectangles. Clicking a colored rectangle in an event marker selects the corresponding function and PC and displays the data for that event and that function in the Event tab. The selection is highlighted in both the Event tab and the Legend tab, and selecting the Source tab or Disassembly tab positions the tab display at the line corresponding to that frame in the call stack.

For some kinds of data, events may overlap and not be visible. Whenever two or more events would appear at exactly the same position, only one is drawn; if there are two or more events within one or two pixels, all are drawn. although they may not be visually distinguishable. In either case, a small gray tick mark appeas below the drawn events indicating the overlap.

You can change the types of event-specific data shown in the Timeline tab, as well as the colors mapped to selected functions. For details about using the Timeline tab, refer to the online help.

The LeakList Tab

The LeakList tab shows two lines, the upper one representing leaks, and the lower one representing allocations. Each contains a call stack, similar to that shown in the Timeline tab, in the center with a bar above proportional to the bytes leaked or allocated, and a bar below proportional to the number of leaks or allocations.

Selection of a leak or allocation displays the data for the selected leak or allocation in the Leak tab, and selects a frame in the call stack, just as it does in the Timeline tab.

The Statistics Tab

The Statistics tab shows totals for various system statistics summed over the selected experiments and samples. The totals are followed by the statistics for the selected samples of each experiment. For information on the statistics presented, see the `getrusage`(3C) and `proc`(4) man pages.

The Experiments Tab

The Experiments tab is divided into two panels. The top panel contains a tree that is divided into two areas: a Notes area and an Info area.

The Notes area displays the contents of any notesfile in the experiment. You can edit the notes by typing directly in the Notes area. The Notes area includes its own toolbar with buttons for saving or discarding the notes and for undoing or redoing any edits since the last save.

The Info area contains information about the experiments collected and the load objects accessed by the collection target, including any error messages or warning messages generated during the processing of the experiment or the load objects.

The bottom panel lists error and warning messages from the Analyzer session.

Data Display, Right Pane

The right pane contains the Summary tab, the Event tab, and the Legend tab. By default the Summary tab is displayed. The other two tabs are dimmed unless the Timeline tab is selected.

The Summary Tab

The Summary tab shows all the recorded metrics for the selected function or load object, both as values and percentages, and information on the selected function or load object. The Summary tab is updated whenever a new function or load object is selected in any tab.

The Event Tab

The Event tab shows detailed data for the event that is selected in the Timeline tab, including the event type, leaf function, LWP ID, thread ID, and CPU ID. Below the data panel the call stack is displayed with the color coding for each function in the stack. Clicking a function in the call stack makes it the selected function.

When a sample is selected in the Timeline tab, the Event tab shows the sample number, the start and end time of the sample, and the microstates with the amount of time spent in each microstate and the color coding.

The Legend Tab

The Legend tab shows a legend for the mapping of colors to functions and to microstates in the Timeline tab. You can change the color that is mapped to an item by selecting the item in the legend and selecting the color chooser from the Timeline menu, or by double-clicking the color box.

The Leak Tab

The Leak tab shows detailed data for the selected leak or allocation in the Leaklist tab. Below the data panel, the Leak tab shows the call stack at the time when the selected leak or allocation was detected. Clicking a function in the call stack makes it the selected function.

Setting Data Presentation Options

You can control the presentation of data from the Set Data Presentation dialog box. To open this dialog box, click the Set Data Presentation button in the toolbar or choose View → Set Data Presentation.

The Set Data Presentation dialog box has a tabbed pane with six tabs:

- Metrics
- Sort
- Source/Disassembly
- Formats
- Timeline
- Search Path

The Metrics tab shows all of the available metrics. Each metric has check boxes in one or more of the columns labeled `Time`, `Value` and`%`, depending on the type of metric. Alternatively, instead of setting individual metrics, you can set all metrics at once by selecting or deselecting the check boxes in the bottom row of the dialog box and then clicking on the Apply to all metrics button.

The Sort tab shows the order of the metrics presented, and the choice of metric to sort by.

The Source/Disassembly tab presents a list of checkboxes that you can use to select the information presented, as follows:

- The compiler commentary that is shown in the source listing and the disassembly listing
- The threshold for highlighting important lines in the source listing and the disassembly listing
- The interleaving of source code in the disassembly listing
- The metrics on the source lines in the disassembly listing
- The display of instructions in hexadecimal in the disassembly listing.

The Formats tab presents a choice for the long form, short form, or mangled form of C++ function names and Java method names. In addition, a checkbox labelled `Append SO name to Function name` adds the name of the shared object, where

the function or method is located, to the end of the function or method name. The Formats tab also presents a choice for Java Mode of `On`, `Expert`, or `Off`; and a choice for Data Space Display of either `Enable` or `Disable`.

The Timeline tab presents choices for the types of event-specific data that are shown, the display of event-specific data for threads, LWP, or CPUs; the alignment of the call stack representation at the root or at the leaf; and the number of levels of the call stack that are displayed.

The Search Path tab allows you to manage a list of directories to be used for searching for source and object files. The special name `$expts` refers to the experiments loaded; all other names should be paths in the file system.

The Set Data Presentation dialog box has a Save button to store the current settings.

Note – Since the defaults for the Analyzer, the `er_print` utility and the `er_src` utility are set by a common `.er.rc` file, output from the `er_print` utility and the `er_src` utility is affected as a result of saving changes in the Analyzer's Set Data Preferences dialog box.

Finding Text and Data

The Analyzer has a Find tool in the toolbar, with two options for search targets that are given in a combo box. You can search for text in the Name column of the Functions tab or Callers-Callees tabs and in the code column of the Source tab and Disassembly tab. You can search for a high-metric item in the Source tab and Disassembly tab. The metric values on the lines containing high-metric items are highlighted in green. Use the arrow buttons next to the Find field to search up or down.

Showing or Hiding Functions

By default, all functions in each load object are shown in the Functions tab and Callers-Callees tab. You can hide all the functions in a load object using the Show/Hide Functions dialog box; see the online help for details.

When the functions in a load object are hidden, the Functions tab and Callers-Callees tab show a single entry representing the aggregate of all functions from the load object. Similarly, the Lines tab and PCs tab show a single entry aggregating all PCs from all functions from the load object.

In contrast to filtering, metrics corresponding to hidden functions are still represented in some form in all displays.

Filtering Data

By default, data is shown in each tab for all experiments, all samples, all threads, all LWPs, and all CPUs. A subset of data can be selected using the Filter Data dialog box. For details about using the Filter Data dialog box, refer to the online help.

Experiment Selection

The Analyzer allows filtering by experiment when more than one experiment is loaded. The experiments can be loaded individually, or by naming an experiment group.

Sample Selection

Samples are numbered from 1 to *N*, and you can select any set of samples. The selection consists of a comma-separated list of sample numbers or ranges such as 15.

Thread Selection

Threads are numbered from 1 to *N*, and you can selecte any set of threads. The selection consists of a comma-separated list of thread numbers or ranges. Profile data for threads only covers that part of the run where the thread was actually scheduled on an LWP.

LWP Selection

LWPs are numbered from 1 to *N*, and you can select any set of LWPs. The selection consists of a comma-separated list of LWP numbers or ranges. If synchronization data is recorded, the LWP reported is the LWP at entry to a synchronization event, which might be different from the LWP at exit from the synchronization event.

On Linux systems, threads and LWPs are synonymous.

CPU Selection

Where CPU information is recorded (Solaris 9 OS), any set of CPUs can be selected. The selection consists of a comma-separated list of CPU numbers or ranges.

Recording Experiments

When you invoke the Analyzer with a target name and target arguments, it starts up with the Performance Tools Collect window open, which allows you to record an experiment on the named target. If you invoke the Analyzer with no arguments, or with an experiment list, you can record a new experiment by choosing File → Collect Experiment to open the Performance Tools Collect window

The Collect Experiment tab of the Performance Tools Collect window has a panel you use to specify the target, its arguments, and the various parameters to be used to run the experiment. They correspond to the options available in the `collect` command, as described in Chapter 3.

Immediately below the panel is a Preview Command button, and a text field. When you click the button, the text field is filled in with the `collect` command that would be used when you click the Run button.

In the Data to Collect tab, you can select the types of data you want to collect.

The Input/Output tab has two panels: one that receives output from the Collector itself, and a second for output from the process.

A set of buttons allows the following operations:

- Run the experiment
- Terminate the run
- Send Pause, Resume, and Sample signals to the process during the run (enabled if the corresponding signals are specified),
- Close the window.

If you close the window while an experiment is in progress, the experiment continues. If you reopen the window, it shows the experiment in progress, as if it had been left open during the run. If you attempt to exit the Analyzer while an experiment is in progress, a dialog box is posted asking whether you want the run terminated or allowed to continue.

Generating Mapfiles and Function Reordering

In addition to analyzing the data, the Analyzer also provides a function-reordering capability. Based on the data in an experiment, the Analyzer can generate a mapfile which, when used with the static linker (ld) to relink the application, creates an executable with a smaller working set size, or better I-cache behavior, or both.

The order of the functions that is recorded in the mapfile and used to reorder the functions in the executable is determined by the metric that is used for sorting the function list. Exclusive User CPU time or Exclusive CPU Cycle time are normally used for producing a mapfile. Some metrics, such as those from synchronization delay or heap tracing, or name or address do not produce meaningful ordering for a mapfile.

Defaults

The Analyzer processes directives from an `.er.rc` file in the current directory, if present; from a `.er.rc` file in your home directory, if present; and from a system-wide `.er.rc` file. These files can contain default settings for metrics, for sorting, and for specifying compiler commentary options and highlighting thresholds for source and disassembly output. They also specify default settings for the Timeline tab, and for name formatting, setting Java™ mode (`javamode`) and Data Space Display mode (`datamode`). The files can also contain directives to control the search path for source files and object files.

In the Analyzer GUI, you can save an `.er.rc` file by clicking the Save button in the Set Data Presentation dialog, which you can open from the View menu. Saving an `.er.rc` file from the Set Data Presentation dialog not only affects subsequent invocations of the Analyzer, but also the `er_print` utility and `er_src` utility.

The Analyzer puts a message into its Errors/Warning Logs areas naming the user `.er.rc` files it processed.

CHAPTER 5

Kernel Profiling

This chapter describes how you can use the Sun Studio performance tools to profile the kernel while the Solaris OS is running a load. Kernel profiling is available if you are running Sun Studio software on the Solaris 10 OS.

Kernel Experiments

You can record kernel profiles with the `er_kernel` utility.

The `er_kernel` utility uses the DTrace driver, a comprehensive dynamic tracing facility that is built into Solaris 10 OS.

The `er_kernel` utility captures kernel profile data and records the data as an Analyzer experiment in the same format as a user profile. The experiment can be processed by the `er_print` utility or the Performance Analyzer. A kernel experiment can show function data, caller-callee data, instruction-level data, and a timeline, but not source-line data (because most Solaris OS modules do not contain line-number tables).

Setting Up Your System for Kernel Profiling

Before you can use the `er_kernel` utility for kernel profiling, you need to set up access to the DTrace driver.

Normally, the DTrace driver is restricted to user `root`. To run `er_kernel` utility as a user other than `root`, you must have specific privileges assigned, and be a member of group `sys`. To assign the necessary privileges, add the following line to the file `/etc/user_attr`:

```
username::::defaultpriv=basic,dtrace_kernel,dtrace_proc
```

To add yourself to the group `sys`, add your user name to the `sys` line in the file `/etc/group`.

Running the `er_kernel` Utility

You can run the `er_kernel` utility to profile only the kernel or both the kernel and the load you are running. For a complete description of the `er_kernel` command, see the `er_kernel`(1) man page.

Profiling the Kernel

1. **Collect the experiment by typing:**

```
% er_kernel -p on
```

2. **Run whatever load you want in a separate shell.**
3. **When the load completes, terminate the `er_kernel` utility by typing ctrl-C.**
4. **Load the resulting experiment, named `ktest.1.er` by default, into the Performance Analyzer or the `er_print` utility.**

   Kernel clock profiling produces one performance metric, labeled KCPU Cycles. In the Performance Analyzer, it is shown for kernel functions in the Functions Tab, for callers and callees in the Caller-Callee Tab, and for instructions in the Disassembly Tab. The Source Tab does not show data, because kernel modules, as shipped, do not usually contain file and line symbol table information (stabs).

   You can replace the `-p on` argument to the `er_kernel` utility with `-p high` for high-resolution profiling or `-p low` for low-resolution profiling,. If you expect the run of the load to take 2 to 20 minutes, the default clock profiling is appropriate. If you expect the run to take less than 2 minutes, use `-p high`; if you expect the run to take longer than 20 minutes, use `-p low`.

You can add a `-t` *n* argument, which will cause the `er_kernel` utility to terminate itself after *n* seconds.

You can add the `-v` argument if you want more information about the run printed to the screen. The `-n` argument lets you see a preview of the experiment that would be recorded, without actually recording anything.

By default, the experiment generated by the `er_kernel` utility is named `ktest.1.er`; the number is incremented for successive runs

Profiling Under Load

If you have a single command, either a program or a script, that you wish to use as a load:

1. **Collect the experiment by typing:**

```
% er_kernel -p on load
```

2. **Analyze the experiment by typing:**

```
% analyzer ktest.1.er
```

The `er_kernel` utility forks a child process and pauses for a quiet period, and then the child process runs the specified load. When the load terminates, the `er_kernel` utility pauses again for a quiet period and then exits. The experiment shows the behavior of the Solaris OS during the running of the load, and during the quiet periods before and after. You can specify the duration of the quiet period in seconds with the `-q` argument to the `er_kernel` command.

Profiling the Kernel and Load Together

If you have a single program that you wish to use as a load, and you are interested in seeing its profile in conjunction with the kernel profile:

1. **Collect both a kernel profile and a user profile by typing both the `er_kernel` command and the `collect` command:**

```
% er_kernel collect load
```

2. **Analyze the two profiles together by typing:**

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

The data displayed by the Analyzer shows both the kernel profile from `ktest.1.er` and the user profile from `test.1.er`. The timeline allows you to see correlations between the two experiments.

Note – To use a script as the load, and profile the various parts of it, prepend the `collect`, command, with the appropriate arguments, to the various commands within the script.

Profiling a Specific Process or Kernel Thread

You can invoke the `er_kernel` utility with one or more `-T` arguments to specify profiling for specific processes or threads:

- `-T` *pid*/*tid* for a specific process and kernel-thread
- -T 0/*did* for a specific pure-kernel thread

The target threads must have been created before you invoke the `er_kernel` utility for them.

When you give one or more `-T` arguments, an additional metric, labeled `Kthr Time`, is produced. Data is captured for all profiled threads, whether running on a CPU or not. Special single-frame call stacks are used for indicating the process is suspended (the function `<SLEEPING>`) or waiting for the CPU (the function `<STALLED>`).

Functions with high Kthr Time metrics, but low KCPU Cycles metrics, are functions that are spending a lot of time for the profiled threads waiting for some other events.

Analyzing a Kernel Profile

A few of the recorded fields in kernel experiments have a different meaning from the same fields in user-mode experiments. A user-mode experiment contains data for a single process ID only; a kernel experiment has data that may apply to many different process IDs. To better present that information, some of the field labels in the Analyzer have different meanings in the two types of experiments:

TABLE 5-1 Field Label Meanings for Kernel Experiments in the Analyzer

Analyzer Label	Meaning in User-mode Experiments	Meaning in Kernel Experiments
LWP	User process LWP ID	Process PID; 0 for kernel threads
Thread	Thread ID within process	Kernel TID; kernel DID for kernel threads

For example, in an kernel experiment, if you want to filter to only a few process IDs, enter the PID(s) of interest in the LWP filter field in the Filter Data dialog box.

CHAPTER 6

The er_print Command Line Performance Analysis Tool

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. You must give the er_print utility the name of one or more experiments or experiment groups generated by the Collector as arguments. You can use the er_print utility to display the performance metrics for functions, for callers and callees; the source code listing and disassembly listing; sampling information; data-space data; and execution statistics.

This chapter covers the following topics.

- er_print Syntax
- Metric Lists
- Commands That Control the Function List
- Commands That Control the Callers-Callees List
- Commands That Control the Leak and Allocation Lists
- Commands That Control the Source and Disassembly Listings
- Commands That Control the Data Space List
- Commands That List Experiments, Samples, Threads, and LWPs
- Commands That Control Selections
- Commands That Control Load Object Selection
- Commands That List Metrics
- Commands That Control Output
- Commands That Print Other Displays
- Commands That Set Defaults
- Commands That Set Defaults Only For the Performance Analyzer
- Miscellaneous Commands
- Examples

For a description of the data collected by the Collector, see Chapter 2.

For instructions on how to use the Performance Analyzer to display information in a graphical format, see Chapter 4 and the online help.

`er_print` Syntax

The command-line syntax for the `er_print` utility is.:

```
er_print [ -script script | -command | - | -V ] experiment-list
```

The options for the `er_print` utility are:

`-`	Read `er_print` commands entered from the keyboard.
`-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.
`-V`	Display version information and exit.

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 the `er_print` utility 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:

```
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 6-1. 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 6-1 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 6-2. 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 6-2 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 period (`.`), plus (`+`), and percent sign (`%`), are equivalent for the purposes of defining the sort order. Thus `sort i%user`, `sort i.user`, and `sort i+user` all mean that the Analyzer should sort by inclusive user CPU time if it is visible in any form, and `sort i!user` means the Analyzer should sort by inclusive user CPU time, whether or not it is visible.

TABLE 6-3 lists the available `er_print` metric name strings for timing metrics, synchronization delay metrics, memory allocation metrics, MPI tracing 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 Profiling Data" on page 37 for more information on hardware counters.

TABLE 6-3 Metric Name Strings

Category	String	Description
Timing metrics	`user`	User CPU time
	`wall`	Wall-clock time
	`total`	Total LWP time
	`system`	System CPU time
	`wait`	CPU wait time
	`ulock`	User lock time
	`text`	Text-page fault time
	`data`	Data-page fault time
	`owait`	Other wait time
Synchronization delay metrics	`sync`	Synchronization wait time
	`syncn`	Synchronization wait count
MPI tracing metrics	`mpitime`	Time spent in MPI calls
	`mpisend`	Number of MPI send operations
	`mpibytessent`	Number of bytes sent in MPI send operations
	`mpireceive`	Number of MPI receive operations
	`mpibytesrecv`	Number of bytes received in MPI receive operations
	`mpiother`	Number of calls to other MPI functions

TABLE 6-3 Metric Name Strings *(Continued)*

Category	String	Description
Memory allocation metrics	`alloc`	Number of allocations
	`balloc`	Bytes allocated
	`leak`	Number of leaks
	`bleak`	Bytes leaked
Hardware counter overflow metrics	`cycles`	CPU cycles
	`insts`	Instructions issued

In addition to the name strings listed in TABLE 6-3, 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. Also note that `cycles` and `insts` are common to SPARC® platforms and x86 platforms, but other flavors also exist that are architecture-specific. To list all available counters, use the `collect` command with no arguments.

Commands That Control the Function List

The following commands control how the function information is displayed.

`functions`

Write the function list with the currently selected metrics. The function list includes all functions in load objects that are selected for display of functions, and any load objects whose functions are hidden with the `object_select` command.

You can limit the number of lines written by using the `limit` command (see "Commands That Control Output" on page 133).

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, which you must issue before you issue the `functions` command. You can also change the defaults with the `dmetrics` command in an `.er.rc` file.

For applications written in the Java programming language, the displayed function information varies depending on whether Java mode is set to `on`, `expert`, or `off`.

- When Java mode is set to `on`, the displayed function information includes metrics against the Java methods, and any native methods called. (A native method is a C/C++ or Fortran function called by a Java target.) Java mode shows data for interpreted and HotSpot-compiled methods aggregated together, and suppresses data for non-user Java threads.
- Setting the Java mode to `expert` shows details of JVM internals that are suppressed in Java mode. In the timeline, all threads are shown, not just Java user threads. In other respects, the displayed function information is the same as when Java mode is set to `on`.
- Setting the Java mode to `off` shows functions from the JVM software itself, rather than from the Java application being interpreted by the JVM software, along with any HotSpot-compiled methods and native methods. All threads are shown.

`metrics` *metric_spec*

Specify a selection of function-list metrics. The string *metric_spec* 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 the `er_print` utility 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 processed, 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 116. 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.

`sort` *metric_spec*

Sort the function list on the specified metric. The string *metric_spec* is one of the metric keywords described in "Metric Lists" on page 116, as shown in this example.

```
% sort i.user
```

This command tells the `er_print` utility 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.

`fsummary`

Write a summary metrics panel for each function in the function list. You can limit the number of panels written by using the `limit` command (see "Commands That Control Output" on page 133).

The summary metrics panel includes the name, address and size of the function or load object, and for functions, the name of the source file, object file and load object, and all the recorded metrics for the selected function or load object, both exclusive and inclusive, as values and percentages.

`fsingle` *function_name* `[`*N*`]`

Write a summary metrics panel for the specified function. The optional parameter *N* is needed for those cases where there are several functions with the same name. The summary metrics panel is written for the *N*th function with the given function name. When the command is given on the command line, *N* is required; if it is not needed it is ignored. When the command is given interactively without *N* but *N* is required, a list of functions with the corresponding *N* value is printed.

For a description of the summary metrics for a function, see the `fsummary` command description.

Commands That Control the Callers-Callees List

The following commands control how the caller and callee information is displayed.

`callers-callees`

Print the callers-callees panel for each of the functions, in the order in which they are sorted. You can limit the number of panels written by using the `limit` command (see "Commands That Control Output" on page 133). The selected (center) function is marked with an asterisk, as shown in this example.

```
Attr.     Excl.     Incl.      Name
User CPU  User CPU  User CPU
 sec.      sec.       sec.
4.440     0.        42.910     commandline
0.        0.         4.440    *gpf
4.080     0.         4.080     gpf_b
0.360     0.         0.360     gpf_a
```

In this example, `gpf` is the selected function; it is called by `commandline`, and it calls `gpf_a` and `gpf_b`.

`cmetrics` *metric_spec*

Specify a selection of callers-callees metrics. *metric_spec* 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 116. To see a listing of the available metrics, use the `cmetric_list` command.

`csingle` *function_name* `[N]`

Write the callers-callees panel for the named function. The optional parameter *N* is needed for those cases where there are several functions with the same name. The callers-callees panel is written for the *N*th function with the given function name. When the command is given on the command line, *N* is required; if it is not needed it is ignored. When the command is given interactively without *N* but *N* is required, a list of functions with the corresponding *N* value is printed.

`csort` *metric_spec*

Sort the callers-callees display by the specified metric. The string *metric_spec* is one of the metric keywords described in "Metric Lists" on page 116, as shown in this example.

```
% csort a.user
```

This command tells the `er_print`utility to sort the callers-callees display by attributed user CPU time. When the command finishes, the sort metric is printed.

Commands That Control the Leak and Allocation Lists

This section describes the commands that relate to memory allocations and deallocations.

`leaks`

Display a list of memory leaks, aggregated by common call stack. Each entry presents the total number of leaks and the total bytes leaked for the given call stack. The list is sorted by the number of bytes leaked.

`allocs`

Display a list of memory allocations, aggregated by common call stack. Each entry presents the number of allocations and the total bytes allocated for the given call stack. The list is sorted by the number of bytes allocated.

Commands That Control the Source and Disassembly Listings

The following commands control how annotated source and disassembly code is displayed.

`pcs`

Write a list of program counters (PCs) and their metrics, ordered by the current sort metric. The list includes lines that show aggregated metrics for each load object whose functions are hidden with the `object_select` command.

psummary

Write the summary metrics panel for each PC in the PC list, in the order specified by the current sort metric.

lines

Write a list of source lines and their metrics, ordered by the current sort metric. The list includes lines that show aggregated metrics for each function that does not have line-number information, or whose source file is unknown, and lines that show aggregated metrics for each load object whose functions are hidden with the `object_select` command.

lsummary

Write the summary metrics panel for each line in the lines list, in the order specified by the current sort metric.

source { *filename* | *function_name* } [*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 the `er_print` utility 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.

The function name can also be specified as *function'file'*, where *file* is used to specify an alternate source context for the function. Immediately following the first instruction, an index line is added for the function. Index lines are displayed as text within angle brackets in the following form:

```
<Function: f_name>
```

The default source context for any function is defined as the source file to which the first instruction in that function is attributed. It is normally the source file compiled to produce the object module containing the function. Alternate source contexts consist of other files that contain instructions attributed to the function. Such contexts include instructions coming from include files and instructions from

functions inlined into the named function. If there are any alternate source contexts, include a list of extended index lines at the beginning of the default source context to indicate where the alternate source contexts are located in the following form:

```
<Function: f, instructions from source file src.h>
```

Note – If you use the `-source` argument when invoking the `er_print` utiity on the command line, the backslash escape character must prepend the file quotes. In other words, the function name is of the form function`\\'file\\'`. The backslash is not required, and should not be used, when the `er_print` utiity is in interactive mode.

`disasm { `*filename*` | `*function_name*` } [`*N*`]`

Write out annotated disassembly code for either the specified file, or the file containing the specified function. The file must be in a directory in your path.

The optional parameter *N* is used in the same way as for the `source` command.

`scc` *com_spec*

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.

TABLE 6-4 Compiler Commentary Message Classes

Class	Meaning
`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.

TABLE 6-4 Compiler Commentary Message Classes *(Continued)*

Class	Meaning
`f[e]`	Show front-end messages.
`all`	Show all messages.
`none`	Do not show any messages.

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 an `.er.rc` file.

`sthresh` *value*

Specify the threshold percentage for highlighting metrics in the annotated source code. If the value of any metric is equal to or greater than *value* % of the maximum value of that metric for any source line in the file, the line on which the metrics occur have `##` inserted at the beginning of the line.

`dcc` *com_spec*

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. You can add the following options to the class list.

TABLE 6-5 Additional Options for the `dcc` Command

Option	Meaning
`h[ex]`	Show the hexadecimal value of the instructions.
`noh[ex]`	Do not show the hexadecimal value of the instructions.
`s[rc]`	Interleave the source listing in the annotated disassembly listing.
`nos[rc]`	Do not interleave the source listing in the annotated disassembly listing.
`as[rc]`	Interleave the annotated source code in the annotated disassembly listing.

dthresh *value*

Specify the threshold percentage for highlighting metrics in the annotated disassembly code. If the value of any metric is equal to or greater than *value* % of the maximum value of that metric for any instruction line in the file, the line on which the metrics occur have ## inserted at the beginning of the line.

setpath *path_list*

Set the path used to find source, object, etc. files. *path_list* is a colon-separated list of directories. If any directory has a colon character in it, escape it with a backslash. The special directory name, `$expts`, refers to the set of current experiments, in the order in which they were loaded; youc can abbreviate it with a single `$` character.

The default setting is: `$expts:.` The compiled-in full pathname is used if a file is not found in searching the current path setting.

`setpath` with no argument prints the current path.

addpath *path_list*

Append *path_list* to the current `setpath` settings.

Commands That Control the Data Space List

data_objects

Write the list of data objects with their metrics. Applicable only to hardware counter overflow experiments where aggressive backtracking was specified, and for objects in files that were compiled with `-xhwcprof`. (Available on SPARC-based systemsfor C only). See the *C User's Guide* or the `cc`(1) man page for further information.

data_osingle *name* [*N*]

Write the summary metrics panel for the named data object. The optional parameter *N* is needed for those cases where the object name is ambiguous. When the directive is on the command-line, *N* is required; if it is not needed, it is ignored. Applicable

only to hardware counter overflow experiments where aggressive backtracking was specified, and for objects in files that were compiled with `-xhwcprof`. (Available on SPARC-based systems for C only). See the *C User's Guide* or the `cc`(1) man page for further information.

`data_olayout`

Write the annotated data object layouts for all program data objects with data-derived metric data, in the order they are defined in the experiment's load objects. Each aggregate data object is shown with the total metrics attributed to it, followed by all of its elements in offset order, each with their own metrics and an indicator of its size and location relative to 32-byte blocks.

`data_osort`

Set the sort metric for data objects. No prefix, such as i., e., or a., is needed.

Commands That List Experiments, Samples, Threads, and LWPs

This section describes the commands that list experiments, samples, threads, and LWPs.

`experiment_list`

Display the full list of experiments loaded with their ID number. Each experiment is listed with an index, which is used when selecting samples, threads, or LWPs.

The following example is an example of an experiment list.

```
(er_print) experiment_list
ID Experiment
== ==========
1 test.1.er
2 test.6.er
```

sample_list

Display the list of samples currently selected for analysis.

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
```

lwp_list

Display the list of LWPs currently selected for analysis.

thread_list

Display the list of threads currently selected for analysis.

cpu_list

Display the list of CPUs currently selected for analysis.

Commands That Control 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, CPUs, 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.

`sample_select` *sample_spec*

Select the samples for which you want to display information. The list of samples you selected is displayed when the command finishes.

`lwp_select` *lwp_spec*

Select the LWPs about which you want to display information. The list of LWPs you selected is displayed when the command finishes.

`thread_select` *thread_spec*

Select the threads about which you want to display information. The list of threads you selected is displayed when the command finishes.

`cpu_select` *cpu_spec*

Select the CPUs about which you want to display information. The list of CPUs you selected is displayed when the command finishes.

Commands That Control Load Object Selection

`object_list`

Display the list of load objects. The name of each load object is preceded either by a `yes` that indicates that the functions of that object are shown in the function list, or by a `no` that indicates that the functions of that object are not shown in the function list.

The following is an example of a load object list.

```
(er_print) object_list
Sel Load Object
=== ================================================
yes /tmp/var/synprog/synprog
yes /opt/SUNWspro/lib/libcollector.so
yes /usr/lib/libdl.so.1
yes /usr/lib/libc.so.1
```

`object_select` *object1,object2,...*

Select the load objects for which you want to display information about the functions in the load object. *object-list* is a list of load objects, separated by commas but no spaces. For load objects that are not selected, information for the entire load object is displayed instead of information for the functions in the load object.

The names of the load objects should be either full path names or the basename. If an object name itself contains a comma, you must surround the name with double quotation marks.

Commands That List Metrics

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.

Commands That Control Output

The following commands control `er_print` display output.

`outfile { `*filename*` | - }`

Close any open output file, then open *filename* for subsequent output. When opening *filename*, clear any pre-existing content. If you specify a dash (-) instead of *filename*, output is written to standard output.

appendfile *filename*

Close any open output file and open *filename*, preserving any pre-existing content, so that subsequent output is appended to the end of the file. If *filename* does not exist, the functionality of the `appendfile` command is the same as for the `outfile` command.

`limit` *n*

Limit output to the first *n* entries of the report; *n* is an unsigned positive integer.

`name { long | short }` [:{*shared_object_name* | *no_shared_object_name* }]"

Specify whether to use the long or the short form of function names (C++ and Java only). If *shared_object_name* is specified, append the shared-object name to the function name.

`javamode { on | expert | off }`

Set the mode for Java experiments to on (show the Java model), expert (show the Java model and additionally show internal JVM details), or off (show the machine model).

Commands That Print Other Displays

`header` *exp_id*

Display descriptive information about the specified experiment. The *exp_id* can be obtained from the `exp_list` command. If the *exp_id* is `all` or is not given, the information is displayed for all experiments loaded.

Following each header, any errors or warnings are printed. Headers for each experiment are separated by a line of dashes.

If the experiment directory contains a file named `notes`, the contents of that file are prepended to the header information. A notes file may be manually added or edited or specified with `-C` "*comment*" arguments to the `collect` command.

exp_id is required on the command line, but not in a script or in interactive mode.

`objects`

List the load objects with any error or warning messages that result from the use of the load object for performance analysis. The number of load objects listed can be limited by using the `limit` command (see "Commands That Control Output" on page 133).

`overview` *exp_id*

Write out the sample data of each of the currently selected samples for the specified experiment. The *exp_id* can be obtained from the `exp_list` command. If the *exp_id* is `all` or is not given, the sample data is displayed for all experiments. *exp_id* is required on the command line, but not in a script or in interactive mode.

`statistics` *exp_id*

Write out execution statistics, aggregated over the current sample set for the specified experiment. For information on the definitions and meanings of the execution statistics that are presented, see the `getrusage`(3C) and `proc`(4) man pages. The execution statistics include statistics from system threads for which the Collector does not collect any data. The standard threads library in the Solaris 7 OS and Solaris 8 OS creates system threads that are not profiled. These threads spend most of their time sleeping, and the time shows in the statistics display as Other Wait time.

The *exp_id* can be obtained from the `experiment_list` command. If the *exp_id* is not given, the sum of data for all experiments is displayed, aggregated over the sample set for each experiment. If *exp_id* is `all`, the sum and the individual statistics for each experiment are displayed.

Commands That Set Defaults

You can use the following commands to set the defaults for `er_print` and for the Performance Analyzer. You can only use these commands for setting defaults: they cannot be used in input for the `er_print` utility.They can be included in a defaults file named `.er.rc`. Some of the commands only apply to the Performance Analyzer.

You can include a defaults file in your home directory to set defaults for all experiments, or in any other directory to set defaults locally. When the `er_print` utility, the `er_src` utility, 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`, `sthresh`, `dcc`, and `dthresh` commands. Multiple `dmetrics` and `dsort` commands can be given in a defaults file, and the commands within a file are concatenated.

`dmetrics` *metric_spec*

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 116. 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_spec*

Specify the default metric by which the function list is sorted. The sort metric is the first metric in this list that matches a metric in any loaded experiment, subject to the following conditions:

- If the entry in *metric_spec* has a visibility string of an exclamation point, !, the first metric whose name matches is used, whether it is visible or not.
- If the entry in *metric_spec* has any other visibility string, the first visible metric whose name matches is used.

The syntax and use of the metric list is described in the section "Metric Lists" on page 116.

The default sort metric for the Callers-Callees list is the attributed metric corresponding to the default sort metric for the function list.

Commands That Set Defaults Only For the Performance Analyzer

`tlmode` *tl_mode*

Set the display mode options for the Timeline tab of the Performance Analyzer. The list of options is a colon-separated list. The allowed options are described in the following table.

TABLE 6-6 Timeline Display Mode Options

Option	Meaning
`lw[p]`	Display events for LWPs
`t[hread]`	Display events for threads
`c[pu]`	Display events for CPUs
`r[oot]`	Align call stack at the root
`le[af]`	Align call stack at the leaf
`d[epth]` *nn*	Set the maximum depth of the call stack that can be displayed

The options `lwp`, `thread`, and `cpu` are mutually exclusive, as are `root` and `leaf`. If more than one of a set of mutually exclusive options is included in the list, the last one is the only one that is used.

`tldata` *tl_data*

Select the default data types shown in the Timeline tab of the Performance Analyzer. The types in the type list are separated by colons. The allowed types are listed in the following table.

TABLE 6-7 Timeline Display Data Types

Type	Meaning
`sa[mple]`	Display sample data
`c[lock]`	Display clock profiling data
`hw[c]`	Display hardware counter profiling data

TABLE 6-7 Timeline Display Data Types *(Continued)*

Type	Meaning
`sy[nctrace]`	Display thread synchronization tracing data
`mp[itrace]`	Display MPI tracing data
`he[aptrace]`	Display heap tracing data

`datamode { on | off }`

Set the mode for showing dataspace-related screens to on (tabs are visible), or off (do not have them visible).

Miscellaneous Commands

`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.

`script` *file*

Process additional commands from the script file *file*.

`version`

Print the current release number of the `er_print` utility

`quit`

Terminate processing of the current script, or exit interactive mode.

`help`

Print a list of `er_print` commands.

Examples

- The following example generates a `gprof`-like list from an experiment. The output is a file called `er_print.out` which lists the top 100 functions, followed by caller-callee data, sorted by attributed user time for each.

  ```
  er_print -outfile er_print.out -metrics e.user:e%user\
  -sort e.user -limit 100 -functions -cmetrics a.user:a%user\
  -csort a.user -callers-callees test.1.er
  ```

  You can also simplify this example into the following independent commands. However, keep in mind that each call to `er_print` in a large experiment or application can be time intensive:

  - `er_print -metrics e.user:e%user -sort e.user \`
    `-limit 100 -functions test.1.er`
  - `er_print -cmetrics a.user:a%user -csort a.user \`
    `-callers-callees test.1.er`

- This example summarizes how time is spent in functions.

  ```
  er_print -functions test.*.er
  ```

- This example shows the caller-callee relationships.

  ```
  er_print -callers-callees test.*.er
  ```

- This example shows which source lines are hot. Source-line information assumes the code was compiled and linked with `-g`. Append a trailing underscore to the function name for Fortran functions and routines. The `1` after the function name is used to distinguish between multiple instances of the *myfunction*.

  er_print -source *myfunction* 1 test.*.er

- This example shows only the compiler commentary. It is not necessary to run your program in order to use this command.

  er_src -*myfile*.o

- These examples uses wall-clock profiling to list functions and callers-callees.

  ```
  er_print -metrics ei.%wall -functions test.*.er
  ```

  ```
  er_print -cmetrics aei.%wall -callers-callees test.*.er
  ```

- This example shows high-level MPI functions. MPI has many internal software layers, but this example shows one way to see just the entry points. There might be some duplicates of symbols, which you can ignore.

  ```
  er_print -functions test.*.er | grep PMPI_
  ```

CHAPTER 7

Understanding the Performance Analyzer and Its Data

The Performance Analyzer reads the event data that is collected by the 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 and discusses the effect on what you see in the Performance Analyzer displays.

This chapter covers the following topics:

- How Data Collection Works
- Interpreting Performance Metrics
- Call Stacks and Program Execution
- Mapping Addresses to Program Structure
- Mapping Data Addresses to Program Data Objects

How Data Collection Works

The output from a data collection run is an experiment, which is stored as a directory with various internal files and subdirectories in the file system.

Experiment Format

All experiments must have three files:

- A log file; an ASCII file that contains information about what data was collected, the versions of various components, a record of various events during the life of the target, and the word size of the target.
- A map file; an ASCII file that records the time-dependent information about what loadobjects are loaded into the address space of the target, and the times at which they are loaded or unloaded.
- An overview file; a binary file containing usage information recorded at every sample point in the experiment.

In addition, experiments have binary data files representing the profile events in the life of the process. Each data file has a series of events, as described below under "Interpreting Performance Metrics" on page 145. Separate files are used for each type of data, but each file is shared by all LWPs in the target. The data files are named as follows:

TABLE 7-1 Data Types and Corresponding File Names

Data Type	File Name
Clock-based profiling	`profile`
Hardware counter overflow profiling	`hwcounters`
Synchronization tracing	`synctrace`
Heap tracing	`heaptrace`
MPI tracing	`mpitrace`

For clock-based profiling, or hardware counter overflow profiling, the data is written in a signal handler invoked by the clock tick or counter overflow. For synchronization tracing, heap tracing, or MPI tracing, data is written from `libcollector.so` routines that are interposed by the `LD_PRELOAD` environment variable on the normal user-invoked routines. Each such interposition routine partially fills in a data record, then invokes the normal user-invoked routine, and fills in the rest of the data record when that routine returns, and writes the record to the data file.

All data files are memory-mapped and filled in blocks. The records are filled in such a way as to always have a valid record structure, so that experiments can be read as they are being written. The buffer management strategy is designed to minimize contention and serialization between LWPs.

An experiment can optionally contain an ASCII file with the filename of `notes`. This file is automatically created when using the `-C comment` argument to the `collect` command. You can create or edit the file manually after the experiment has been created. The contents of the file are prepended to the experiment header.

The `archives` Directory

Each experiment has an `archives` directory that contains binary files describing each load object referenced in the `loadobjects` file. These files are produced by the `er_archive` utility, which runs at the end of data collection. If the process terminates abnormally, the `er_archive` utility may not be invoked, in which case, the archive files are written by the `er_print` utility or the Analyzer when first invoked on the experiment.

Descendant Processes

Descendant processes write their experiments into subdirectories within the founder-process' experiment. These subdirectories are named with an underscore, a code letter (`f` for fork, `x` for exec, and `c` for combination), and a number are added to its immediate creator's experiment name, giving the genealogy of the descendant. For example, if the experiment name for the founder process is `test.1.er`, the experiment for the child process created by its third fork is `test.1.er/_f3.er`. If that child process executes a new image, the corresponding experiment name is `test.1.er/_f3_x1.er`. Descendant experiments consist of the same files as the parent experiment, but they do not have descendant experiments (all descendants are represented by subdirectories in the founder experiment), and they do not have archive subdirectories (all archiving is done into the founder experiment).

Dynamic Functions

An experiment where the target creates dynamic functions has additional records in the `loadobjects` file describing those functions, and an additional file, `dyntext`, containing a copy of the actual instructions of the dynamic functions. The copy is needed to produce annotated disassembly of dynamic functions.

Java Experiments

A Java experiment has additional records in the `loadobjects` file, both for dynamic functions created by the JVM software for its internal purposes, and for dynamically-compiled (HotSpot) versions of the target Java methods.

In addition, a Java experiment has a JAVA_CLASSES file, containing information about all of the user's Java classes invoked.

Java heap tracing data and synchronization tracing data are recorded using a JVMPI agent, which is part of libcollector.so. The agent receives events that are mapped into the recorded trace events. The agent also receives events for class loading and HotSpot compilation, that are used to write the JAVA_CLASSES file, and the Java-compiled method records in the loadobjects file.

Recording Experiments

You can record an experiment in three different ways:

- With the collect command
- With dbx creating a process
- With dbx creating an experiment from a running process

The Performance Tools Collect window in the Analyzer GUI runs a collect experiment; the Collector dialog in the IDE runs a dbx experiment.

collect Experiments

When you use the collect command to record an experiment, the collect utility creates the experiment directory and sets the LD_PRELOAD environment variable to ensure that libcollector.so is preloaded into the target's address space. It then sets environment variables to inform libcollector.so about the experiment name, and data collection options, and executes the target on top of itself.

libcollector.so is responsible for writing all experiment files.

dbx Experiments That Create a Process

When dbx is used to launch a process with data collection enabled, dbx also creates the experiment directory and ensures preloading of libcollector.so. dbx stops the process at a breakpoint before its first instruction, and then calls an initialization routine in libcollector.so to start the data collection.

Java experiments can not be collected by dbx, since dbx uses a Java™ Virtual Machine Debug Interface (JVMDI) agent for debugging, and that agent can not coexist with the Java™ Virtual Machine Profiling Interface (JVMPI) agent needed for data collection.

dbx Experiments, on a Running Process

When dbx is used to start an experiment on a running process, it creates the experiment directory, but cannot use the LD_PRELOAD environment variable. dbx makes an interactive function call into the target tolopen libcollector.so, and then calls the libcollector.so initialization routine, just as it does when creating the process. Data is written by libcollector.so just as in a collect experiment.

Since libcollector.so was not in the target address space when the process started, any data collection that depends on interposition on user-callable functions (synchronization tracing, heap tracing, MPI tracing) might not work. In general, the symbols have already been resolved to the underlying functions, so the interposition can not happen. Furthermore, the following of descendant processes also depends on interposition, and does not work properly for experiments created by dbx on a running process.

If you have explicitly preloaded libcollector.so before starting the process with dbx, or before using dbx to attach to the running process, you can collect tracing data.

Interpreting Performance Metrics

The data for each event contains a high-resolution timestamp, a thread ID, an LWP ID, and a processor ID. The first three of these can be used to filter the metrics in the Performance Analyzer by time, thread or LWP. See the getcpuid(2) man page for information on processor IDs. On systems where getcpuid is not available, the processor ID is -1, which maps to Unknown.

In addition to the common data, 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. On the Solaris OS, an interval counter is provided At the end of the profiling interval, the appropriate interval counter is incremented by 1, and another profiling signal is scheduled. The array is recorded and reset only when the Solaris LWP thread enters CPU user mode. Resetting the array consists of setting the array element for the User-CPU state to 1, and the array elements for all the other states to 0. The array data is recorded on entry to user mode before the array is reset. Thus,

the array contains an accumulation of counts for each microstate that was entered since the previous entry into user mode. for each of the ten microstates maintained by the kernel for each Solaris LWP. On the Linux OS, microstates do not exist; the only interval counter is User CPU Time.

The call stack is recorded at the same time as the data. If the Solaris LWP is not in user mode at the end of the profiling interval, the call stack cannot change until the LWP or thread 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 on the Solaris OS are shown in TABLE 7-2.

TABLE 7-2 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	User Lock 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. Run your program for long enough or enough times to accumulate hundreds of profile packets for any function or source line you are interested in.

In addition to statistical sampling errors, specific errors arise from the way the data is collected and attributed and the way the program progresses through the system. The following are some of the circumstances in which inaccuracies or distortions can appear in the timing metrics:

- When a Solaris LWP or Linux thread is created, the time 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. If many LWP or threads are created, the error can be many times the profiling interval.
- When a Solaris LWP or Linux thread is destroyed, some time is spent after the last profile packet is recorded. If many LWPs or threads are destroyed, the error can be many times the profiling interval.
- Rescheduling of LWPs or threads can occur during a 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 Solaris LWPs or Linux threads to run than there are processors to run them.
- A program can behave in a way that is correlated with the system clock. In this case, the profiling interval always expires when the Solaris LWP or Linux thread is in a state that 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, the profiling signal can 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. On the Solaris OS, 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. The problem applies to the default threads library on the Solaris 8 OS only.
- When the system clock is being synchronized with an external source, the time stamps recorded in profile packets do not reflect the profiling interval but include any adjustment that was made to the clock. The clock adjustment can make it appear that profile packets are lost. The time period involved is usually several seconds, and the adjustments are made in increments.

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 the 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 a few 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 Solaris LWP or Linux thread 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 functions, source-lines, and such are not substantially distorted.

For multithreaded applications using unbound threads on the Solaris OS, differences in values returned by `gethrvtime()` could be meaningless 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 that appear in the Statistics tab of the Performance Analyzer and the `er_print` statistics display are based on process file system `/proc` 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. However, the values displayed in the Statistics tab can include other contributions that are not included in the timing metric values for `<Total>`. These contributions come from the following sources:

- Threads that are created by the system that are not profiled. The standard threads library in the Solaris 8 OS creates system threads that are not profiled. These threads spend most of their time sleeping, and the time shows in the Statistics tab as Other Wait time.
- Periods of time in which data collection is paused.

Synchronization Wait Tracing

Synchronization wait tracing is available only on Solaris platforms. The Collector collects synchronization delay events by tracing calls to the functions in the threads library, `libthread.so`, or to the real time extensions library, `librt.so`. 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 thread and LWP IDs are the IDs at the time the data is recorded. 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 User Lock Time. User Lock Time can be larger than Synchronization Wait Time because the synchronization delay threshold screens out delays of short duration.

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 User Lock 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. You can minimize the fraction of the wait time spent in overhead by increasing the threshold for recording events.

Hardware Counter Overflow Profiling

Hardware counter overflow profiling is available only on Solaris platforms. 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. The value is especially likely to be larger for 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 193 for more information. See also the discussion of "Traps" on page 154. Traps and trap handlers can cause significant differences between reported User CPU time and time reported by the cycle counter.

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 collecting data 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 of data 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_final_counters`. If you see a substantial fraction of the counts in this function, the overflow value is too large.

Heap Tracing

The Collector records tracing data for calls to the memory allocation and deallocation functions `malloc`, `realloc`, `memalign`, and `free` by interposing on these functions. If your program bypasses these functions to allocate memory, tracing data is not recorded. Tracing data is not recorded for Java memory management, which uses a different mechanism.

The functions that are traced could be loaded from any of a number of libraries. The data that you see in the Performance Analyzer might depend on the library from which a given function is loaded.

If a program makes a large number of calls to the traced functions in a short space of time, the time taken to execute the program can be significantly lengthened. The extra time is used in recording the tracing data.

Dataspace Profiling

A dataspace profile is a data collection in which memory- related events, such as cache misses, are reported against the data-object references that cause the events rather than just the instructions where the memory-related events occur. Dataspace profiling is not available on Linux systems.

To allow dataspace profiling, the target must be a C program, compiled for the SPARC architecture, with the `-xhwcprof` flag and `-xdebugformat=dwarf -g` flag. Furthermore, the data collected must be hardware counter profiles for memory-related counters and the optional + sign must be prepended to the counter name. The Performance Analyzer now includes two tabs related to dataspace profiling, the DataObject tab and the DataLayout tab.

MPI Tracing

MPI tracing is available only on Solaris platforms. MPI tracing records information about calls to MPI library functions. The event-specific data consists of high-resolution timestamps for the request and the grant (beginning and end of the call that is traced), the number of send and receive operations and the number of bytes sent or received. Tracing is done by interposing on the calls to the MPI library. The interposing functions do not have detailed information about the optimization of data transmission, nor about transmission errors, so the information that is presented represents a simple model of the data transmission, which is explained in the following paragraphs.

The number of bytes received is the length of the buffer as defined in the call to the MPI function. The actual number of bytes received is not available to the interposing function.

Some of the Global Communication functions have a single origin or a single receiving process known as the root. The accounting for such functions is done as follows:

- Root sends data to all processes, itself included.
- Root receives data from all processes. itself included.
- Each process communicates with each process, itself included

The following examples illustrate the accounting procedure. In these examples, G is the size of the group.

For a call to `MPI_Bcast()`,

- Root sends G packets of N bytes, one packet to each process, including itself
- All G processes in the group (including root) receive N bytes

For a call to `MPI_Allreduce()`,

- Each process sends G packets of N bytes
- Each process receives G packets of N bytes

For a call to `MPI_Reduce_scatter()`,

- Each process sends G packets of N/G bytes
- Each process receives G packets of N/G bytes

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. For information on unwind failures, see "Incomplete Stack Unwinds" on page 165.

The leaf PC in a call stack is used to assign 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 assign 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 167 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 annul 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.

If a profiling event occurs during the execution of one of the three PLT instructions, the PLT PCs are deleted, and exclusive time is attributed to the call instruction. If a profiling event occurs during the first call through a PLT entry, but the leaf PC is not one of the PLT instructions, any PCs that arise from the PLT and code in `ld.so` are replaced by a call to an artificial function, `@plt`, which accumulates inclusive time. There is one such artificial function for each shared object. If the program uses the `LD_AUDIT` interface, the PLT entries might never be patched, and non-leaf PCs from `@plt` can occur more frequently.

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 function, `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 function `sigacthandler()`, and `sigacthandler()` 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 function.

The Collector interposes on `sigaction()` to ensure that its handlers are the primary handlers for the `SIGPROF` signal when clock data is collected and `SIGEMT` signal when hardware counter overflow data is collected.

Traps

Traps can be issued by an instruction or by the hardware, and are caught by a trap handler. System traps are traps that 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 for some register-content values on the UltraSPARC® III platform), or when the instruction is not implemented in the hardware.

When a trap is issued, the Solaris LWP or Linux kernel enters system mode. On the Solaris OS, the microstate is usually switched from User CPU 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 functions that generate fast traps are `gethrtime` and `gethrvtime`. In these functions, 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 not switched.

In both cases, the time spent is recorded as User CPU time. However, the hardware counters are turned off because the CPU 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 and Cycles time, preferably recorded in the same experiment.

In one case 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 function does is to call another function. 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 function `A` calls functions `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 (lightweight process) in the Solaris OS. Multithreaded executables make calls to a thread creation function, 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 function 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 function. Inclusive metrics associated with the created thread 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 on Solaris 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. The threads library can create more LWPs at a later time if they are needed. Unbound threads are not part of the Solaris 9 OS or of the alternate threads library in the Solaris 8 OS.

  As an example of the scheduling of unbound threads, 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 is not accrued into the User Lock Time metric.

In addition to the user threads, the standard threads library in the Solaris 8 OS creates some threads that are used to perform signal handling and other tasks. If the program uses bound threads, additional LWPs are also created for these threads. Performance data is not collected or displayed for these threads, which spend most of their time sleeping. However, the time spent in these threads is included in the process statistics and in the times recorded in the sample data. The threads library in the Solaris 9 OS and the alternate threads library in the Solaris 8 OS do not create these extra threads.

The Linux OS provides P-threads (POSIX threads) for explicit multithreading. The data type `pthread_attr_t` controls the behavioral attributes of a thread. To create a bound thread, the attribute's scope must be set to `PTHREAD_SCOPE_SYSTEM` using the `pthread_attr_setscope()` function. Threads are unbound by default, or if the attribute scope is set to `PTHREAD_SCOPE_PROCESS`. To create a new thread, the application calls the P-thread API function `pthread_create()`, passing a pointer to an application-defined start routine as one of the function arguments. When the new thread starts execution, it runs in a Linux-specific system function, `clone()`, which calls another internal initialization function, `pthread_start_thread()`, which in turn calls the user-defined start routine originally passed to `pthread_create()`. The Linux metrics-gathering functions available to the Collector are thread-specific, whether the thread is bound to an LWP or not. Therefore, when the `collect` utility runs, it interposes a metrics-gathering function, named `collector_root()`, between `pthread_start_thread()` and the application-defined thread start routine.

Overview of Java Technology-Based Software Execution

To the typical developer, a Java technology-based application runs just like any other program. The application begins at a main entry point, typically named *`class`*`.main`, which may call other methods, just as a C or C++ application does.

To the operating system, an application written in the Java programming language, (pure or mixed with C/C++), runs as a process instantiating the JVM software. The JVM software is compiled from C++ sources and starts execution at `_start`, which calls `main`, and so forth. It reads bytecode from `.class` and/or `.jar` files, and performs the operations specified in that program. Among the operations that can be specified is the dynamic loading of a native shared object, and calls into various functions or methods contained within that object.

During execution of a Java technology-based application, most methods are interpreted by the JVM software; these methods are referred to in this document as *interpreted methods*. Other methods may be dynamically compiled by the Java HotSpot virtual machine, and are referred to as *compiled methods*. Dynamically compiled methods are loaded into the data space of the application, and may be unloaded at some later point in time. For any particular method, there is an interpreted version, and there may also be one or more compiled versions. Code written in the Java programming language might also call directly into native-compiled code, either C, C++, Fortran, or native-compiled SBA (SPARC® Bytecode Accelerator) Java; the targets of such calls are referred to as *native methods*.

The JVM software does a number of things that are typically not done by applications written in traditional languages. At startup, it creates a number of regions of dynamically-generated code in its data space. One of these regions is the actual interpreter code used to process the application's bytecode methods.

During the interpretive execution, the Java HotSpot virtual machine monitors performance, and may decide to take one or more methods that it has been interpreting, generate machine code for them, and execute the more-efficient machine code version, rather than interpret the original. That generated machine code is also in the data space of the process. In addition, other code is generated in the data space to execute the transitions between interpreted and compiled code.

Applications written in the Java programming language are inherently multithreaded, and have one JVM software thread for each thread in the user's program. Java applications also have several housekeeping threads used for signal handling, memory management, and Java HotSpot virtual machine compilation. Depending on the version of `libthread.so` used, there may be a one-to-one correspondence between threads and LWPs, or a more complex relationship. For the default `libthread.so` thread library on the Solaris 8 OS, a thread might be unscheduled at any instant, or scheduled onto an LWP. Data for a thread is not collected while that thread is not scheduled onto an LWP. A thread is never unscheduled when using the alternate `libthread.so` library on the Solaris 8 OS nor when using the Solaris 9 OS threads.

Java Call Stacks and Machine Call Stacks

The performance tools collect their data by recording events in the life of each Solaris LWP or Linux thread, along with the call stack at the time of the event. At any point in the execution of any application, the call stack represents where the program is in its execution, and how it got there. One important way that mixed-model Java applications differ from traditional C, C++, and Fortran applications is that at any instant during the run of the target there are two callstacks that are meaningful: a Java call stack, and a machine callstack. Both call stacks are recorded during profiling, and are reconciled during analysis.

Clock-based Profiling and Hardware Counter Overflow Profiling

Clock-based profiling and hardware counter overflow profiling for Java programs work just as for C, C++, and Fortran programs, except that both Java call stacks and machine call stacks are collected.

Synchronization Tracing

Synchronization tracing for Java programs is based on events generated when a thread attempts to acquire a Java Monitor. Both machine cal lstacks and Java call stacks are collected for these events, but no synchronization tracing data is collected for internal locks used within the JVM software.

Heap Tracing

Heap tracing data records object-allocation events, generated by the user code, and object-deallocation events, generated by the garbage collector. In addition, any use of C/C++ memory-management functions, such as `malloc` and `free`, also generates events that are recorded. Those events might come from native code, or from the JVM software itself.

Java Processing Representations

There are three representations for displaying performance data for applications written in the Java programming language: the Java representation, the Expert-Java representation, and the Machine representation. The Java representation is shown by default where the data supports it. The following section summarizes the main differences between these three representations.

The Java Representation

The Java representation shows compiled and interpreted Java methods by name, and shows native methods in their natural form. During execution, there might be many instances of a particular Java method executed: the interpreted version, and, perhaps, one or more compiled versions. In the Java representation all methods are shown aggregated as a single method. This representation is selected in the Analyzer by default.

A PC for a Java method in the Java representation corresponds to the method-id and a bytecode index into that method; a PC for a native function correspond to a machine PC. The call stack for a Java thread may have a mixture of Java PCs and machine PCs. It does not have any frames corresponding to Java housekeeping code, which does not have a Java representation. Under some circumstances, the JVM software cannot unwind the Java stack, and a single frame with the special function, `<no Java™ callstack recorded>`, is returned. Typically, it amounts to no more than 5-10% of the total time.

For the housekeeping threads, only a machine call stack is obtained, and, in the Java representation, data for those threads is attributed to the special function, `<JVM-System>`.

The function list in the Java representation shows metrics against the Java methods and any native methods called. The list also shows the pseudo-functions from JVM overhead threads. The caller-callee panel shows the calling relationships in the Java representation.

Source for a Java method corresponds to the source code in the `.java` file from which it was compiled, with metrics on each source line. The disassembly of any Java method shows the bytecode generated for it, with metrics against each bytecode, and interleaved Java source, where available.

The Timeline in the Java representation shows only Java threads. The callstack for each thread is shown with its Java methods.

Java programs allocate memory for instantiating classes and storing data, but, unlike C and C++ applications, they do not explicitly deallocate the memory. Rather memory is managed by a so-called *garbage collector*. That code, a part of the JVM, periodically scans memory to find allocated areas that are no longer pointed to elsewhere in the program, and it reclaims the memory, deallocating it and making it available for other uses. Heap tracing in the Java representation is based on the Java memory management, and JVMPI events; data from normal Heap tracing is shown in the Java™ representation, as well.

All Java programs may have explicit synchronization, usually performed by calling the `monitor-enter` routine.

Synchronization-delay tracing in the Java representation is based on the JVMPI synchronization events. Data from the normal synchronization tracing is not shown in the Java representation.

Data space profiling in the Java™ representation is not currently supported

The Expert-Java Representation

The Expert-Java representation is similar to the Java Representation, except that some details of the JVM internals that are suppressed in the Java Representation are exposed in the Expert-Java Representation. Native frames are used when `<no Java Callstack recorded>` would appear in the Java representation. With the Expert-Java representation, the Timeline shows all threads.

The Machine Representation

The Machine representation shows functions from the JVM software itself, rather than from the application being interpreted by the JVM software. It also shows all compiled and native methods. The machine representation looks the same as that of applications written in traditional languages. The call stack shows JVM frames, native frames, and compiled-method frames. Some of the JVM frames represent transition code between interpreted Java, compiled Java, and native code.

Source from compiled methods are shown against the Java source; the data represents the specific instance of the compiled-method selected. Disassembly for compiled methods show the generated machine assembler code, not the Java bytecode. Caller-callee relationships show all overhead frames, and all frames representing the transitions between interpreted, compiled, and native methods.

The Timeline in the machine representation shows bars for all threads, LWPs, or CPUs, and the call stack in each is the machine-representation call stack.

In the machine representation, memory is allocated and deallocated by the JVM software, typically in very large chunks. Memory allocation from the Java code is handled entirely by the JVM software and its garbage-collector by mapping memory. Heap tracing still shows JVM allocations, since a memory mapping operation is treated as a memory allocation when heap tracing.

In the machine representation, thread synchronization devolves into calls to `_lwp_mutex_lock`. No synchronization data is shown, since these calls are not traced.

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 Studio compilers and tools. Refer to the *OpenMP API User's Guide* and the relevant sections in the *Fortran Programming Guide* and *C User's Guide*, 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 Sun Studio 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 function. The microtasking library function is responsible for dispatching threads to execute the body function. The address of the body function is passed to the microtasking library function as an argument.

If the parallel construct is delimited with one of the directives in the following list, then the construct is replaced with a call to the microtasking library function `__mt_MasterFunction_()`.

- The Sun Fortran directive `!$par doall`
- The Cray Fortran directive `c$mic doall`
- A Fortran OpenMP `!$omp PARALLEL`, `!$omp PARALLEL DO`, or `!$omp PARALLEL SECTIONS` directive
- A C or C++ OpenMP `#pragma omp parallel`, `#pragma omp parallel for`, or `#pragma omp parallel sections` directive

A loop that is parallelized automatically by the compiler is also replaced by a call to `__mt_MasterFunction_()`.

If an OpenMP parallel construct contains one or more worksharing `do`, `for` or `sections` directives, each worksharing construct is replaced by a call to the microtasking library function `__mt_Worksharing_()` and a new body function is created for each.

The compiler assigns names to body functions that encode the type of parallel construct, the name of the function from which the construct was extracted, the line number of the beginning of the construct in the original source, and the sequence

number of the parallel construct. These mangled names vary from release to release of the microtasking library, but are shown demangled into more comprehensible names.

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 function, `thr_create()` to create worker threads. Each worker thread executes the microtasking library function `__mt_SlaveFunction_()`, which was passed as an argument to `thr_create()`.

In addition to worker threads, the standard threads library in the Solaris 8 OS creates some threads to perform signal handling and other tasks. Performance data is not collected for these threads, which spend most of their time sleeping. However, the time spent in these threads is included in the process statistics and the times recorded in the sample data. The threads library in the Solaris 9 OS and the alternate threads library in the Solaris 8 OS do not create these extra threads.

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, a parallel workshare directive, or a parallel directive.

- In the parallel sections case, `__mt_run_my_job_()` calls the body function directly.
- In the parallel do or for case, `__mt_run_my_job_()` calls other functions, which depend on the nature of the loop, and the other functions call the body function.
- In the parallel case, `__mt_run_my_job_()` calls the body function directly, and all threads execute the code in the body function until they encounter a call to `__mt_WorkSharing_()`. This function contains another call to `__mt_run_my_job_()`, which calls the worksharing body function, either directly in the case of a worksharing section, or through other library functions 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.

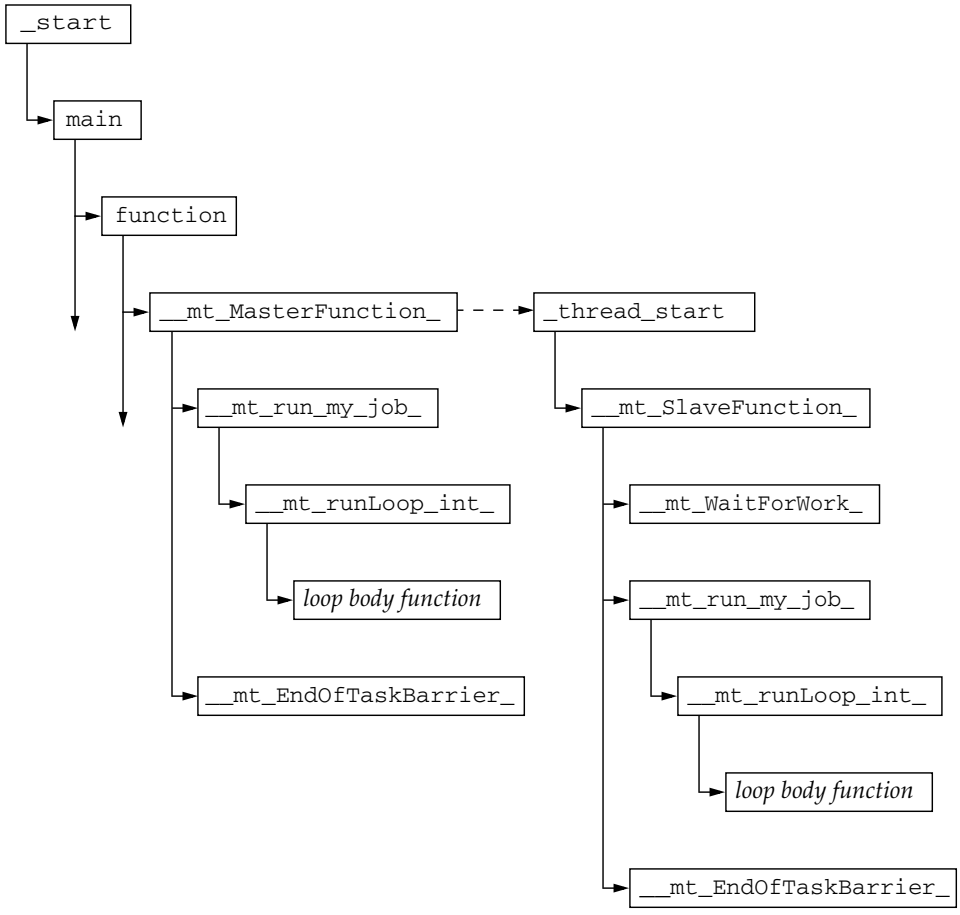

FIGURE 7-1 Schematic Call Tree for a Multithreaded Program That Contains a Parallel Do or Parallel For Construct

When all parallel work is finished, the threads return to either `__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.

The call sequence for a simple parallel `do` construct is illustrated in FIGURE 7-1. The call stack for a worker thread begins with the threads library function `_thread_start()`, the function 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 7-2. 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 7-1.

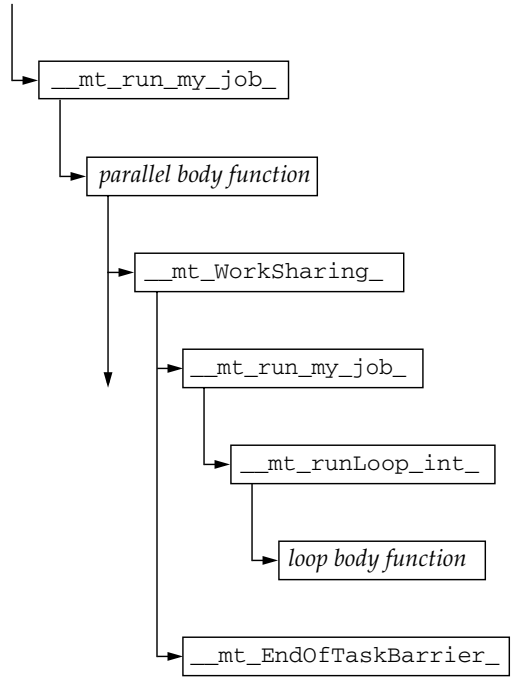

FIGURE 7-2 Schematic Call Tree for a Parallel Region With a Worksharing Do or Worksharing For Construct

In these call sequences, all the compiler-generated body functions are called from the same function (or functions) in the microtasking library, which makes it difficult to associate the metrics from the body function with the original user function. The Analyzer inserts an imputed call to the body function from the original user function, and the microtasking library inserts an imputed call from the body function to the barrier function, `__mt_EndOfTaskBarrier_()`. The metrics due to the synchronization are therefore attributed to the body function, and the metrics for the body function 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 functions. The side effect of these imputed calls is that the body function appears as a callee of both the original user function and the microtasking functions. In addition, the user function appears to

have microtasking library functions as its callers, and can appear to call itself. Double-counting of inclusive metrics is avoided by the mechanism used for recursive function calls (see "How Recursion Affects Function-Level Metrics" on page 47).

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 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 in the Solaris 8 OS by setting the environment variable `MT_BIND_LWP` to `FALSE`. Overriding the default is *not* recommended.

Note – The multiprocessing dispatch process is implementation-dependent and might change from release to release.

Incomplete Stack Unwinds

Stack unwind might fail for a number of reasons:

- If the stack has been corrupted by the user code; if so, the program might core dump, or the data collection code might core dump, depending on exactly how the stack was corrupted.
- If the user code does not follow the standard ABI conventions for function calls. In particular, on the SPARC platform, if the return register, %o7, is altered before a save instruction is executed.

  On any platform, hand-written assembler code might violate the conventions.
- On the x86 platform, if C or Fortran code is compiled at high optimization, which means that it does not have frame pointers to assist in the unwind.
- On the x86 platform, if C++ code is compiled at any optimization level with the `-noex` or `-features=no%except` options.
- If the leaf PC is in a function after the callee's frame is popped from the stack, but before the function returns.

- If the call stack contains more than about 250 frames, the Collector does not have the space to completely unwind the call stack. In this case, PCs for functions from `_start` to some point in the call stack are not recorded in the experiment. The artificial function `<Truncated-stack>` is shown as called from `<Total>` to tally the topmost frames recorded.
- On x86 and x64 platforms, stack unwind can be problematic if the code does not preserve frame pointers. To preserve frame pointers, compile with the `-xreg=no%frameptr` option.

Intermediate Files

If you generate intermediate files using the `-E` or `-P` compiler options, the Analyzer uses the intermediate file for annotated source code, not the original source file. The `#line` directives generated with `-E` can cause problems in the assignment of metrics to source lines.

The following line appears in annotated source if there are instructions from a function that do not have line numbers referring to the source file that was compiled to generate the function:

function_name `-- <instructions without line numbers>`

Line numbers can be absent under the following circumstances:

- You compiled without specifying the `-g` option.
- The debugging information was stripped after compilation, or the executables or object files that contain the information are moved or deleted or subsequently modified.
- The function contains code that was generated from #include files rather than from the original source file.
- At high optimization, if code was inlined from a function in a different file.
- The source file has #line directives referring to some other file; compiling with the `-E` option, and then compiling the resulting `.i` file is one way in which this happens. It may also happen when you compile with the `-P` flag.
- The object file cannot be found to read line number information.
- The file was compiled without the `-g` flag, or the file was stripped.
- The compiler used generates incomplete line number tables.

Mapping Addresses to Program Structure

Once a call stack is processed into PC values, the 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 that 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 at different times 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 Java programming language, 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. In the Summary tab, 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 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 tab, 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, under some circumstances, 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 Analyzer. In the Summary tab, 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 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 Collector wraps some library functions, and both the wrapper function and the real function can appear in the 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 you might use. When libraries are stripped, the names of static functions are deleted from the symbol table. In such cases, the 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 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 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 functions.

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 subroutine 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 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 appears 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.

Cloned Functions

The compilers have the ability to recognize calls to a function for which extra optimization can be performed. An example of such calls is a call to a function for which some of the arguments are constants. When the compiler identifies particular calls that it can optimize, it creates a copy of the function, which is called a clone, and generates optimized code. The clone function name is a mangled name that identifies the particular call. The Analyzer demangles the name, and presents each instance of a cloned function separately in the function list. Each cloned function has a different set of instructions, so the annotated disassembly listing shows the cloned functions separately. Each cloned function has the same source code, so the annotated source listing sums the data over all copies of the function.

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 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, and is recommended.

- 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 functions 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 161.

The 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 metrics 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 162.

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.

Note – The names of compiler-generated body functions can only be demangled for modules compiled with `-g`

Outline Functions

Outline functions can be created during feedback-optimized compilations. They represent code that is not normally executed, specifically code that is not executed during the training run used to generate the feedback for the final optimized compilation. A typical example is code that performs error checking on the return value from library functions; the error-handling code is never normally run. To improve paging and instruction-cache behavior, such code is moved elsewhere in the address space, and is made into a separate function. The name of the outline function encodes information about the section of outlined code, including the name of the function from which the code was extracted and the line number of the beginning of the section in the source code. These mangled names can vary from release to release. The Analyzer provides a readable version of the function 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 Analyzer imputes an artificial call from the main function 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.

For further details on feedback-optimized compilations, refer to the description of the `-xprofile` compiler option in *Appendix B* of the *C User's Guide*, *Appendix A* of the *C++ User's Guide*, or *Chapter 3* of the *Fortran User's Guide*.

Dynamically Compiled Functions

Dynamically compiled functions are functions that are compiled and linked while the program is executing. The Collector has no information about dynamically compiled functions that are written in C or C++, unless the user supplies the required information using the Collector API functions. See "Dynamic Functions and Modules" on page 59 for information about the API functions. If information is not supplied, the function appears in the performance analysis tools as `<Unknown>`.

For Java programs, the Collector obtains information on methods that are compiled by the Java HotSpot virtual machine, and there is no need to use the API functions to provide the information. For other methods, the performance tools show information for the JVM software that executes the methods. In the Java representation, all methods are merged with the interpreted version. In the machine representation, each HotSpot-compiled version is shown separately, and JVM functions are shown for each interpreted method.

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 show PCs mapping to <Unknown>:

- When a function written in C or C++ is dynamically generated, and information about the function is not provided to the Collector using the Collector API functions. See "Dynamic Functions and Modules" on page 59 for more information about the Collector API functions.
- When a Java method is dynamically compiled but Java profiling is disabled.
- When the PC corresponds to an address in the data section of the executable or a shared object. One case is the SPARC V7 version of `libc.so`, which 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 SPARC V9 platform.
- When the PC corresponds to a shared object in the address space of the executable that is not recorded in the experiment.
- When the PC is not within any known load object. The most likely cause 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.
- When a PC maps to an internal part of the JVM software for which the Collector has no symbolic information.

Callers and callees of the <Unknown> function represent the previous and next PCs in the call stack, and are treated normally.

The <JVM-System> Function

In the Java representation, the <JVM-System> function represents time used by the JVM software performing actions other than running a Java program. In this time interval, the JVM software is performing tasks such as garbage collection and HotSpot compilation. By default, <JVM-System> is visible in the Function list.

The <no Java callstack recorded> Function

The <no Java callstack recorded> function is similar to the <Unknown> function, but for Java threads, in the Java representation only. When the Collector receives an event from a Java thread, it unwinds the native stack and calls into the

JVM software to obtain the corresponding Java stack. If that call fails for any reason, the event is shown in the Analyzer with the artificial function <no Java callstack recorded>. The JVM software might refuse to report a call stack either to avoid deadlock, or when unwinding the Java stack would cause excessive synchronization.

The <Truncated-stack> Function

The size of the buffer used by the Analyzer for recording the metrics of individual functions in the call stack is limited. If the size of the call stack becomes so large that the buffer becomes full, any further increase in size of the callstack will force the analyzer to drop function profile information. Since in most programs the bulk of exclusive CPU time is spent in the leaf functions, the Analyzer drops the metrics for functions the less critical functions at the bottom of the stack, starting with the entry functions _start() and main(). The metrics for the dropped functions are consolidated into the single artificial <Truncated-stack> function. The <Truncated-stack> function may also appear in Java programs.

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>. The function 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. If the stack unwind was incomplete, the <Total> function can appear as the caller of <Truncated-stack>.

Functions Related to Hardware Counter Overflow Profiling

The following functions are related to hardware countr overflow profiling:

- collector_not_program_related: The counter does not relate to the program.
- collector_lost_hwc_overflow: The counter appears to have exceeded the overflow value without generating an overflow signal. The value is recorded and the counter reset.

- `collector_lost_sigemt`: The counter appears to have exceeded the overflow value and been halted but the overflow signal appears to be lost. The value is recorded and the counter reset.
- `collector_hwc_ABORT`: Reading the hardware counters has failed, typically when a privileged process has taken control of the counters, resulting in the termination of hardware counter collection.
- `collector_final_counters`: The values of the counters taken immediately before suspending or terminating collection, with the count since the previous overflow. If this corresponds to a significant fraction of the `<Total>` count, a smaller overflow interval (that is, a higher resolution configuration) is recommended.
- `collector_record_counters`: The counts accumulated while handling and recording hardware counter events, partially accounting for hardware counter overflow profiling overhead. If this corresponds to a significant fraction of the `<Total>` count, a larger overflow interval (that is, a lower resolution configuration) is recommended.

Mapping Data Addresses to Program Data Objects

Once a PC from a hardware counter event corresponding to a memory operation has been processed to successfully backtrack to a likely causal memory-referencing instruction, the Analyzer uses instruction identifiers and descriptors provided by the compiler in its hardware profiling support information to derive the associated program data object.

The term *data object* is used to refer to program constants, variables, arrays and aggregates such as structures and unions, along with distinct aggregate elements, described in source code. Depending on the source language, data object types and their sizes vary. Many data objects are explicitly named in source programs, while others may be unnamed. Some data objects are derived or aggregated from other (simpler) data objects, resulting in a rich, often complex, set of data objects.

Each data object has an associated scope, the region of the source program where it is defined and can be referenced, which may be global (such as a load object), a particular compilation unit (an object file), or function. Identical data objects may be defined with different scopes, or particular data objects referred to differently in different scopes.

Data-derived metrics from hardware counter events for memory operations collected with backtracking enabled are attributed to the associated program data object type and propagate to any aggregates containing the data object and the

artificial <Total>, which is considered to contain all data objects (including <Unknown> and <Scalars>). The different subtypes of <Unknown> propagate up to the <Unknown> aggregate. The following section describes the <Total>, <Scalars>, and <Unknown> data objects.

Data Object Descriptors

Data objects are fully described by a combination of their declared type and name. A simple scalar data object {int i} describes an variable called i of type int, while {const+pointer+int p} describes a constant pointer to a type int called p. Spaces in the type names are replaced with underscore (_), and unnamed data objects are represented with a name of dash (-), for example: {double_precision_complex -}.

An entire aggregate is similarly represented {structure:foo_t} for a structure of type foo_t. An element of an aggregate requires the additional specification of its container, for example, {structure:foo_t}.{int i} for a member i of type int of the previous structure of type foo_t. Aggregates can also themselves be elements of (larger) aggregates, with their corresponding descriptor constructed as a concatenation of aggregate descriptors and ultimately a scalar descriptor.

While a fully-qualified descriptor may not always be necessary to disambiguate dataobjects, it provides a generic complete specification to assist with dataobject identification.

The <Total> Data Object

The <Total> data object is an artificial construct used to represent the program's data objects as a whole. All performance metrics, in addition to being attributed to a distinct data object (and any aggregate to which it belongs), are attributed to the special data object <Total>. It appears at the top of the data object list and its data can be used to give perspective to the data for other data objects.

The <Scalars> Data Object

While aggregate elements have their performance metrics additionally attributed into the metric value for their associated aggregate, all of the scalar constants and variables have their performance metrics additionally attributed into the metric value for the artificial <Scalars> data object.

The <Unknown> Data Object and Its Elements

Under various circumstances, event data can not be mapped to a particular data object. In such cases, the data is mapped to the special data object named <Unknown> and one of its elements as described below.

- Module with trigger PC not compiled with -xhwcprof

  No event-causing instruction or data object was identified because the object code was not compiled with hardware counter profiling support.

- Backtracking failed to find a valid branch target

  No event-causing instruction was identified because the hardware profiling support information provided in the compilation object was insufficient to verify the validity of backtracking.

- Backtracking traversed a branch target

  No event-causing instruction or data object was identified because backtracking encountered a control transfer target in the instruction stream.

- No identifying descriptor provided by the compiler

  Backtracking determined the likely causal memory-referencing instruction, but its associated data object was not specified by the compiler.

- No type information

  Backtracking determined the likely event-causing instruction, but the instruction was not identified by the compiler as a memory-referencing instruction.

- Not determined from the symbolic information provided by the compiler

  Backtracking determined the likely causal memory-referencing instruction, but it was not identified by the compiler and associated data object determination is therefore not possible. Compiler temporaries are generally unidentified.

- Backtracking was prevented by a jump or call instruction

  No event-causing instructions were identified because backtracking encountered a branch or call instruction in the instruction stream.

- Backtracking did not find trigger PC

  No event-causing instructions were found within the maximum backtracking range.

- Could not determine VA because registers changed after trigger instruction

  The virtual address of the data object was not determined because registers were overwritten during hardware counter skid.

- Memory-referencing instruction did not specify a valid VA

  The virtual address of the data object did not appear to be valid.

CHAPTER 8

Understanding Annotated Source and Disassembly Data

Annotated source code and annotated disassembly code are useful for determining which source lines or instructions within a function are responsible for poor performance, and to view commentary on how the compiler has performed transformations on the code. This section describes the annotation process and some of the issues involved in interpreting the annotated code.

Annotated Source Code

Annotated source code for an experiment can be viewed in the Performance Analyzer by selecting the Source tab in the left pane of the Analyzer window. Alternatively, annotated source code can be viewed without running an experiment, using the `er_src` utility. This section of the manual describes how source code is displayed in the Performance Analyzer. For details on viewing annotated source code with the `er_src` utility, see "Viewing Source/Disassembly Without An Experiment" on page 202.

Annotated source in the Analyzer contains the following information:

- The contents of the original source file
- The performance metrics of each line of executable source code
- Highlighting of code lines with metrics exceeding a specific threshold
- Index lines
- Compiler commentary

Performance Analyzer Source Tab Layout

The Source tab is divided into columns, with fixed-width columns for individual metrics on the left and the annotated source taking up the remaining width on the right.

Identifying the Original Source Lines

All lines displayed in black in the annotated source are taken from the original source file. The number at the start of a line in the annotated source column corresponds to the line number in the original source file. Any lines with characters displayed in a different color are either index lines or compiler commentary lines.

Index Lines in the Source Tab

A source file is any file compiled to produce an object file or interpreted into byte code. An object file normally contains one or more regions of executable code corresponding to functions, subroutines, or methods in the source code. The Analyzer analyzes the object file, identifies each executable region as a function, and attempts to map the functions it finds in the object code to the functions, routines, subroutines, or methods in the source file associated with the object code. When the analyzer succeeds, it adds an index line in the annotated source file in the location corresponding to the first instruction in the function found in the object code.

The annotated source shows an index line for every function—including inline functions—even though inline functions are not displayed in the list displayed by the Function tab. The Source tab displays index lines in red italics with text in angle-brackets. The simplest type of index line corresponds to the function's default context. The default source context for any function is defined as the source file to which the first instruction in that function is attributed. The following example shows an index line for a C function `icputime`.

```
              600. int
              601. icputime(int k)
0.    0.      602. {
                     <Function: icputime>
```

As can be seen from the above example, the index line appears on the line following the first instruction. For C source, the first instruction corresponds to the opening brace at the start of the function body. In Fortran source, the index line for each

subroutine follows the line containing the `subroutine` keyword. Also, a `main` function index line follows the first Fortran source instruction executed when the application starts, as shown in the following example:

```
             1. ! Copyright 02/04/2000 Sun Microsystems, Inc. All Rights Reserved
             2. !
             3. ! Synthetic f90 program, used for testing openmp directives and
             4. !       the analyzer
             5.
             6. !$PRAGMA C (gethrtime, gethrvtime)
             7.
0.    81.497[  8] 9000    format('X ', f7.3, 7x, f7.3, 4x, a)
                 <Function: main>
```

Sometimes, the Analyzer might not be able to map a function it finds in the object code with any programming instructions in the source file associated with that object code; for example, code may be `#included` or inlined from another file, such as a header file.

If the source for the object code is not the default source context for a function contained in the object code, then the annotated source corresponding to the object code contains a special index line cross-referring to the function's default source context. For example, compiling the `synprog` demo creates an object module, `endcases.o` corresponding to source file `endcases.c`. The source in `endcases.c` declares function `inc_func`, which is defined in header `inc_func.h`. Headers often include inline function definitions in this way. Because `endcases.c` declares function `inc_func`, but no source line within `endcases.c` corresponds to the instructions of `inc_func`, the top of the annotated source file for `endcases.c` displays a special index line, as follows:

```
0.650 0.650   <Function: inc_func, instructions from source file inc_func.h>
```

The metrics on the index line indicates that part of the code from the `endcases.o` object module does not have line-mappings (within the source file `endcases.c`).

The Analyzer also adds a standard index line in the annotated source for `inc_func.h`, where the function `inc_func` is defined.

```
                2.
                3. void
                4. inc_func(int n)
0.    0.        5. {
                    <Function: inc_func>
```

Similarly, if a function has an *alternate* source context[1], then an index line cross referring to that context is displayed in the annotated source for the *default* source context.

```
                142. inc_body(int n)
0.650 0.650       <Function: inc_body, instructions from source file inc_body.h>
0.    0.        143. {
                        <Function: inc_body>
                144. #include "inc_body.h"
0.    0.        145. }
```

Double-clicking on an index line referring to another source context will display the contents of that source file in the source window.

Also displayed in red are special index lines and other special lines that are not compiler commentary. For example, as a result of compiler optimization, a special index line might be created for a function in the object code that does not correspond to code written in any source file. For details, refer to "Special Lines in the Source, Disassembly and PCs Tabs" on page 194.

Compiler Commentary

Compiler commentary indicates how compiler-optimized code has been generated. Compiler commentary lines are displayed in blue, to distinguish them from index lines and original source lines. 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.

1. Alternate source contexts consist of other files that contain instructions attributed to the function. Such contexts include instructions coming from include files (as shown in the above example) and instructions from functions inlined into the named function.

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 following shows an example of compiler commentary.

```
0.   0.   Function freegraph inlined from source file ptraliasstr.c into the
code for the following line
            47.       freegraph();
          > 48.    }
0.   0.     49.    for (j=0;j<ITER;j++) {

          Function initgraph inlined from source file ptraliasstr.c into the
code for the following line
0.   0.     50.       initgraph(rows);

          Function setvalsmod inlined from source file ptraliasstr.c into the
code for the following line
          Loop below fissioned into 2 loops
          Loop below fused with loop on line 51
          Loop below had iterations peeled off for better unrolling and/or
parallelization
          Loop below scheduled with steady-state cycle count = 3
          Loop below unrolled 8 times
          Loop below has 0 loads, 3 stores, 3 prefetches, 0 FPadds, 0 FPmuls,
and 0 FPdivs per iteration
             51.       setvalsmod();
```

Note that the commentary for line 51 includes loop commentary because the function `setvalsmod()` contains loop code, and the function has been inlined.

The above extract shows the compiler commentary wrapping at the end of the line, which does not occur in the Source tab of the Analyzer because the tab is not constrained by width.

You can set the types of compiler commentary displayed in the Source tab using the Source/Disassembly tab in the Set Data Presentation dialog box; for details, see "Setting Data Presentation Options" on page 104.

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.

Loop Optimizations

The compiler can do several types of loop optimization. Some of the more common ones are as follows:

- Loop unrolling
- Loop peeling
- Loop interchange
- Loop fission
- Loop fusion

Loop unrolling consists of repeating several iterations of a loop within the loop body, and adjusting the loop index accordingly. As the body of the loop becomes larger, the compiler can schedule the instructions more efficiently. Also reduced is the overhead caused by the loop index increment and conditional check operations. The remainder of the loop is handled using loop peeling.

Loop peeling consists of removing a number of loop iterations from the loop, and moving them in front of or after the loop, as appropriate.

Loop interchange changes the ordering of nested loops to minimize memory stride, to maximize cache-line hit rates.

Loop fusion consists of combining adjacent or closely located loops into a single loop. The benefits of loop fusion are similar to loop unrolling. In addition, if common data is accessed in the two pre-optimized loops, cache locality is improved by loop fusion, providing the compiler with more opportunities to exploit instruction-level parallelism.

Loop fission is the opposite of loop fusion: a loop is split into two or more loops. This optimization is appropriate if the number of computations in a loop becomes excessive, leading to register spills that degrade performance. Loop fission can also come into play if a loop contains conditional statements. Sometimes it is possible to split the loops into two: one with the conditional statement and one without. This can increase opportunities for software pipelining in the loop without the conditional statement.

Sometimes, with nested loops, the compiler applies loop fission to split a loop apart, and then performs loop fusion to recombine the loop in a different way to increase performance. In this case, you see compiler commentary similar to the following:

```
    Loop below fissioned into 2 loops
    Loop below fused with loop on line 116
    [116]     for (i=0;i<nvtxs;i++) {
```

Inlining

With an inline function, the compiler inserts the function instructions directly at the locations where it is called instead of making actual function calls. Thus, similar to a C/C++ macro, the instructions of an inline function are replicated at each call location. The compiler performs explicit or automatic inlining at high optimization levels (4 and 5). Inlining saves the cost of a function call and provides more instructions for which register usage and instruction scheduling can be optimized, at the cost of a larger code footprint in memory. The following is an example of inlining compiler commentary.

```
         Function initgraph inlined from source file ptralias.c into the code
for the following line
0.    0.      44.        initgraph(rows);
```

The compiler commentary does not wrap onto two lines in the Source tab of the Analyzer.

Parallelization

If your code contains Sun, Cray, or OpenMP parallelization directives, it can be compiled for parallel execution on multiple processors. The compiler commentary indicates where parallelization has and has not been performed, and why. The following shows an example of parallelization computer commentary.

```
0.    6.324  9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)

           Loop below parallelized by explicit user directive
           Loop below interchanged with loop on line 12
0.010 0.010[10]            do i = 2, n-1

           Loop below not parallelized because it was nested in a parallel loop
           Loop below interchanged with loop on line 12
0.170 0.170 11.               do j = 2, i
```

For more details about parallel execution and compiler-generated body functions, refer to "Parallel Execution and Compiler-Generated Body Functions" on page 161.

Special Lines in the Annotated Source

Several other annotations for special cases can be shown under the Source tab, either in the form of compiler commentary, or as special lines displayed in the same color as index lines. For details, refer to "Special Lines in the Source, Disassembly and PCs Tabs" on page 194.

Source Line Metrics

Source code metrics are displayed, for each line of executable code, in fixed-width columns. The metrics are the same as in the function list. You can change the defaults for an experiment using a `.er.rc` file; for details, see "Commands That Set Defaults" on page 135. You can also change the metrics displayed and highlighting thresholds in the Analyzer using the Set Data Presentation dialog box; for details, see "Setting Data Presentation Options" on page 104.

Annotated source code shows the metrics of an application at the source-line level. It is produced by taking the PCs (program counts) that are recorded in the application's call stack, and mapping each PC to a source line. To produce an annotated source file, the 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 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 Analyzer searches for the source file, object file, and executable files in the following default locations in turn, and stops when it finds a file of the correct basename:

- The archive directories of experiments
- The current working directory
- The absolute pathname as recorded in the executables or compilation objects

The default can be changed by the `addpath` or `resetpath` directive, or by the Analyzer GUI.

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.

Interpreting Source Line Metrics

Metrics for an instruction must be interpreted as metrics accrued while waiting for the instruction to be executed. If the instruction being executed when an event is recorded comes from the same source line as the leaf PC, the metrics can be interpreted as due to execution of that source line. However, if the leaf PC comes from a different source line than the instruction being executed, at least some of the metrics for the source line that the leaf PC belongs to must be interpreted as metrics accumulated while this line was waiting to be executed. An example is when a value that is computed on one source line is used on the next source line.

The issue of how to interpret the metrics matters most when there is a substantial delay in execution, such as at a cache miss or a resource queue stall, or when an instruction is waiting for a result from a previous instruction. In such cases the metrics for the source lines can seem to be unreasonably high, and you should look at other lines in the code to find the line responsible for the high metric value.

Metric Formats

The four possible formats for the metrics that can appear on a line of annotated source code are explained in TABLE 8-1.

TABLE 8-1 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 from that line with an incorrect line number.
`0.`	Some PCs in the program were tagged as derived from this line, but no data referred to those PCs: they were never in a call stack that was sampled statistically or traced. The `0.` metric does not indicate that the line was not executed, only that it did not show up statistically in a profiling data packet or a recorded tracing data packet.
`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.

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 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 Analyzer can interleave the source with the disassembly, depending on the display preference.
- If the compiler has inserted any commentary into the object code, it too, is interleaved in the disassembly if the corresponding preferences are set.

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, and can be shown on any interleaved source code if the corresponding preference is set. Possible metric values are as described for source-code annotations, in TABLE 8-1.

The disassembly listing for code that is `#included` in multiple locations repeatsthe disassembly instructions once for each time that the code has been `#included`. The source code is interleaved only for the first time a repeated block of disassembly code is shown in a file. For example, if a block of code defined in a header called `inc_body.h` is `#included` by four functions named `inc_body`, `inc_entry`, `inc_middle`, and `inc_exit`, then the block of disassembly instructions appears four times in the disassembly listing for `inc_body.h`, but the source code is interleaved only in the first of the four blocks of disassembly instructions. Switching to Source tab reveals index lines corresponding to each of the times that the disassembly code was repeated.

Index lines can be displayed in the Disassembly tab. Unlike with the Source tab, these index lines cannot be used directly for navigation purposes. However, placing the cursor on one of the instructions immediately below the index line and selecting the Source tab navigates you to the file referenced in the index line.

Files that `#include` code from other files show the included code as raw disassembly instructions without interleaving the source code. However, placing the cursor on one of these instructions and selecting the Source tab opens the file containing the `#included` code. Selecting the Disassembly tab with this file displayed shows the disassembly code with interleaved source code.

Source code can be interleaved with disassembly code for inline functions, but not for macros.

When code is not optimized, the line numbers for each instruction are in sequential order, and the interleaving of source lines and disassembled instructions occurs in the expected way. When optimization takes place, instructions from later lines sometimes appear before those from earlier lines. The 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. One effect of optimization is that source code can appear between a control transfer instruction and its delay slot instruction. Compiler commentary associated with line *N* of the source is written immediately before that line.

Interpreting Annotated Disassembly

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 known as instruction issue groups. 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. As a result, 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. And when the call stack is recorded, there might be several instructions that could be considered the next instruction to execute.

Instruction issue rules vary from one processor type to another, and depend on the instruction alignment within cache lines. Since the linker forces instruction alignment at a finer granularity than the cache line, changes in a function that might seem unrelated can cause different alignment of instructions. The different alignment can cause a performance improvement or degradation.

The following artificial situation shows the same function compiled and linked in slightly different circumstances. The two output examples shown below are the annotated disassembly listings from the `er_print` utility . The instructions for the two examples are identical, but the instructions are aligned differently.

In this example the instruction alignment maps the two instructions `cmp` and `bl,a` to different cache lines, and a significant amount of time is used waiting to execute these two instructions.

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                          1. static int
                          2. ifunc()
                          3. {
                          4.     int i;
                          5.
                          6.     for (i=0; i<10000; i++)
                             <function: ifunc>
   0.010     0.010            [ 6]    1066c:  clr         %o0
   0.        0.               [ 6]    10670:  sethi       %hi(0x2400), %o5
   0.        0.               [ 6]    10674:  inc         784, %o5
                          7.          i++;
   0.        0.               [ 7]    10678:  inc         2, %o0
## 1.360     1.360            [ 7]    1067c:  cmp         %o0, %o5
## 1.510     1.510            [ 7]    10680:  bl,a        0x1067c
   0.        0.               [ 7]    10684:  inc         2, %o0
   0.        0.               [ 7]    10688:  retl
   0.        0.               [ 7]    1068c:  nop
                          8.      return i;
                          9. }
```

In this example, the instruction alignment maps the two instructions `cmp` and `bl,a` to the same cache line, and a significant amount of time is used waiting to execute only one of these instructions.

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                            1. static int
                            2. ifunc()
                            3. {
                            4.     int i;
                            5.
                            6.     for (i=0; i<10000; i++)
                               <function: ifunc>
   0.        0.                 [ 6]    10684:  clr         %o0
   0.        0.                 [ 6]    10688:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    1068c:  inc         784, %o5
                            7.         i++;
   0.        0.                 [ 7]    10690:  inc         2, %o0
## 1.440    1.440              [ 7]    10694:  cmp         %o0, %o5
   0.        0.                 [ 7]    10698:  bl,a        0x10694
   0.        0.                 [ 7]    1069c:  inc         2, %o0
   0.        0.                 [ 7]    106a0:  retl
   0.        0.                 [ 7]    106a4:  nop
                            8.     return i;
                            9. }
```

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 instruction 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).

- On UltraSPARC® III processors, a cache miss on a load instruction blocks all instructions that follow it until the miss is resolved, regardless of whether these instructions use the data item that is being loaded. UltraSPARC® II processors only block instructions that use the data item that is being loaded.

Attribution of Hardware Counter Overflows

Apart from TLB misses, 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 interrupt generated by the overflow. For some counters, such as cycles or instructions issued, this delay 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 PC that caused the event 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 PC that caused the event. For hardware counters that count memory access events, the Collector searches for the PC that caused the event if the counter name is prefixed with a plus, +.

Special Lines in the Source, Disassembly and PCs Tabs

Outline Functions

Outline functions can be created during feedback-optimized compilations. They are displayed as special index lines in the Source tab and Diassembly tab. In the Source tab, an annotation is displayed in the block of code that has been converted into an outline function.

```
            Function binsearchmod inlined from source file ptralias2.c into the
              58.         if( binsearchmod( asize, &element ) ) {
0.240 0.240   59.             if( key != (element << 1) ) {
0.    0.      60.                 error |= BINSEARCHMODPOSTESTFAILED;
                <Function: main -- outline code from line 60 [_$o1B60.main]>
0.040 0.040[ 61]                 break;
              62.             }
              63.         }
```

In the Disassembly tab, the outline functions are typically displayed at the end of the file.

```
                <Function: main -- outline code from line 85 [_$o1D85.main]>
0.    0.        [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.    0.        [ 86] 100001038:  bset        4, %i3
0.    0.        [ 85] 10000103c:  or          %i5, 1, %l7
0.    0.        [ 85] 100001040:  sllx        %l7, 12, %l5
0.    0.        [ 85] 100001044:  call        printf ! 0x100101300
0.    0.        [ 85] 100001048:  add         %l5, 336, %o0
0.    0.        [ 90] 10000104c:  cmp         %i3, 0
0.    0.        [ 20] 100001050:  ba,a        0x1000010b4
                <Function: main -- outline code from line 46 [_$o1A46.main]>
0.    0.        [ 46] 100001054:  mov         1, %i3
0.    0.        [ 47] 100001058:  ba          0x100001090
0.    0.        [ 56] 10000105c:  clr         [%i2]
                <Function: main -- outline code from line 60 [_$o1B60.main]>
0.    0.        [ 60] 100001060:  bset        2, %i3
0.    0.        [ 61] 100001064:  ba          0x10000109c
0.    0.        [ 74] 100001068:  mov         1, %o3
```

The name of the outline function is displayed in square brackets, and encodes information about the section of outlined code, including the name of the function from which the code was extracted and the line number of the beginning of the section in the source code. These mangled names can vary from release to release. The Analyzer provides a readable version of the function name. For further details, refer to "Outline Functions" on page 172.

If an outline function is called when collecting the performance data for an application, the Analyzer displays a special line in the annotated disassembly to show inclusive metrics for that function. For further details, see "Inclusive Metrics" on page 201.

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 161.

The compiler assigns mangled names to body functions that encode the type of parallel construct, the name of the function from which the construct was extracted, the line number of the beginning of the construct in the original source, and the sequence number of the parallel construct. These mangled names vary from release to release of the microtasking library, but are shown demangled into more comprehensible names.

The following shows a typical compiler-generated body function as displayed in the functions list.

```
7.415  14.860  psec_ -- OMP sections from line 9 [_$s1A9.psec_]
3.873   3.903  craydo_ -- MP doall from line 10 [_$d1A10.craydo_]
```

As can be seen from the above examples, the name of the function from which the construct was extracted is shown first, followed by the type of parallel construct, followed by the line number of the parallel construct, followed by the mangled name of the compiler-generated body function in square brackets. Similarly, in the disassembly code, a special index line is generated.

```
                <Function: psec_ -- OMP sections from line 9 [_$s1A9.psec_]>
0.    7.445     [24]    1d8cc:  save        %sp, -168, %sp
0.    0.        [24]    1d8d0:  ld          [%i0], %g1
0.    0.        [24]    1d8d4:  tst         %i1
```

```
                <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_]>
0.    0.030      [ ?]     197e8:  save        %sp, -128, %sp
0.    0.         [ ?]     197ec:  ld          [%i0 + 20], %i5
0.    0.         [ ?]     197f0:  st          %i1, [%sp + 112]
0.    0.         [ ?]     197f4:  ld          [%i5], %i3
```

With Cray directives, the function may not be correlated with source code line numbers. In such cases, a `[ ?]` is displayed in place of the line number. If the index line is shown in the annotated source code, the index line indicates instructions without line numbers, as shown below.

```
              9. c$mic  doall shared(a,b,c,n) private(i,j,k)

            Loop below fused with loop on line 23
            Loop below not parallelized because autoparallelization is not
enabled
            Loop below autoparallelized
            Loop below interchanged with loop on line 12
            Loop below interchanged with loop on line 12
3.873 3.903      <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_],
instructions without line numbers>
0.    3.903 10.             do i = 2, n-1
```

Note – Index lines and compiler-commentary lines do not wrap in the real displays.

Dynamically Compiled Functions

Dynamically compiled functions are functions that are compiled and linked while the program is executing. The Collector has no information about dynamically compiled functions that are written in C or C++, unless the user supplies the required information using the Collector API function `collector_func_load()`. Information displayed by the Function tab, Source tab, and Disassembly tab depends on the information passed to `collector_func_load()` as follows:

- If information is not supplied—`collector_func_load()` is not called—the dynamically compiled and loaded function appears in the function list as `<Unknown>`. Neither function source nor disassembly code is viewable in the Analyzer.
- If no source file name and no line-number table is provided, but the name of the function, its size, and its address are provided, the name of the dynamically compiled and loaded function and its metrics appear in the function list. The

annotated source is available, but the disassembly instructions are viewable, although the line numbers are specified by [?] to indicate that they are unknown.

- If the source file name is given, but no line-number table is provided, the information displayed by the Analyzer is similar to the case where no source file name is given, except that the beginning of the annotated source displays a special index line indicating that the function is composed of instructions without line numbers. For example:

```
1.121 1.121       <Function func0, instructions without line numbers>
              1. #include        <stdio.h>
```

- If the source file name and line-number table is provided, the function and its metrics are displayed by the Function tab , Source tab, and Disassembly tab in the same way as conventionally compiled functions.

For more information about the Collector API functions, see "Dynamic Functions and Modules" on page 59.

For Java programs, most methods are interpreted by the JVM software. The Java HotSpot virtual machine, running in a separate thread, monitors performance during the interpretive execution. During the monitoring process, the virtual machine may decide to take one or more interpreted methods, generate machine code for them, and execute the more-efficient machine-code version, rather than interpret the original.

For Java programs, there is no need to use the Collector API functions; the Analyzer signifies the existence of Java HotSpot-compiled code in the annotated disassembly listing using a special line underneath the index line for the method, as shown in the following example.

```
              11.    public int add_int () {
              12.       int       x = 0;
                  <Function: Routine.add_int()>
2.832 2.832       Routine.add_int() <HotSpot-compiled leaf instructions>
0.    0.          [ 12] 00000000: iconst_0
0.    0.          [ 12] 00000001: istore_1
```

The disassembly listing only shows the interpreted byte code, not the compiled instructions. By default, the metrics for the compiled code are shown next to the special line. The exclusive and inclusive CPU times are different than the sum of all the inclusive and exclusive CPU times shown for each line of interpreted byte code. In general, if the method is called on several occasions, the CPU times for the

compiled instructions are greater than the sum of the CPU times for the interpreted byte code, because the interpreted code is executed only once when the method is initially called, whereas the compiled code is executed thereafter.

The annotated source does not show Java HotSpot-compiled functions. Instead, it displays a special index line to indicate instructions without line numbers. For example, the annotated source corresponding to the disassembly extract shown above is as follows:

```
              11.    public int add_int () {
2.832 2.832     <Function: Routine.add_int(), instructions without line numbers>
0.    0.        12.       int       x = 0;
                  <Function: Routine.add_int()>
```

Java Native Functions

Native code is compiled code originally written in C, C++, or Fortran, called via the Java Native Interface (JNI) by Java code. The following example is taken from the annotated disassembly of file `jsynprog.java` associated with demo program `jsynprog`.

```
              5. class jsynprog
                 <Function: jsynprog.<init>()>
0.    5.504      jsynprog.JavaCC() <Java native method>
0.    1.431      jsynprog.JavaCJava(int) <Java native method>
0.    5.684      jsynprog.JavaJavaC(int) <Java native method>
0.    0.         [  5] 00000000: aload_0
0.    0.         [  5] 00000001: invokespecial <init>()
0.    0.         [  5] 00000004: return
```

Because the native methods are not included in the Java source, the beginning of the annotated source for `jsynprog.java` shows each Java native method using a special index line to indicate instructions without line numbers.

```
0.    5.504      <Function: jsynprog.JavaCC(), instructions without line numbers>
0.    1.431       <Function: jsynprog.JavaCJava(int), instructions without line
numbers>
0.    5.684       <Function: jsynprog.JavaJavaC(int), instructions without line
numbers>
```

Note – The index lines do not wrap in the real annotated source display.

Cloned Functions

The compilers have the ability to recognize calls to a function for which extra optimization can be performed. An example of such is a call to a function where some of the arguments passed are constants. When the compiler identifies particular calls that it can optimize, it creates a copy of the function, which is called a clone, and generates optimized code.

In the annotated source, compiler commentary indicates if a cloned function has been created:

```
          Function foo from source file clone.c cloned, creating cloned function
_$c1A.foo; constant parameters propagated to clone
0.    0.570 27.     foo(100, 50, a, a+50, b);
```

Note – Compiler commentary lines do not wrap in the real annotated source display.

The clone function name is a mangled name that identifies the particular call. In the above example, the compiler commentary indicates that the name of the cloned function is `_$c1A.foo`. This function can be seen in the function list as follows:

```
0.350 0.550 foo
0.340 0.570 _$c1A.foo
```

Each cloned function has a different set of instructions, so the annotated disassembly listing shows the cloned functions separately. They are not associated with any source file, and therefore the instructions are not associated with any source line numbers. The following shows the first few lines of the annotated disassembly for a cloned function.

```
                <Function: _$c1A.foo>
0.    0.        [?]     10e98:  save        %sp, -120, %sp
0.    0.        [?]     10e9c:  sethi       %hi(0x10c00), %i4
0.    0.        [?]     10ea0:  mov         100, %i3
0.    0.        [?]     10ea4:  st          %i3, [%i0]
0.    0.        [?]     10ea8:  ldd         [%i4 + 640], %f8
```

Static Functions

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 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 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. Examples of static functions can be found in the functions list of the `jsynprog` demo, reproduced below.

```
0.   0.   <static>@0x18780
0.   0.   <static>@0x20cc
0.   0.   <static>@0xc9f0
0.   0.   <static>@0xd1d8
0.   0.   <static>@0xe204
```

In the PCs tab, the above functions are represented with an offset, as follows:

```
0.   0.   <static>@0x18780 + 0x00000818
0.   0.   <static>@0x20cc + 0x0000032C
0.   0.   <static>@0xc9f0 + 0x00000060
0.   0.   <static>@0xd1d8 + 0x00000040
0.   0.   <static>@0xe204 + 0x00000170
```

An alternative representation in the PCs tab of functions called within a stripped library is: <*library*.so> -- no functions found + 0x0000F870

Inclusive Metrics

In the annotated disassembly, special lines exist to tag the time taken by slave threads and by outline functions.

The following is an example of *<inclusive metrics for slave threads>*, taken from the demo program `omptest`.

```
              3.          subroutine pardo(n,m,a,b,c)
                 <Function: pardo_>
0.    0.          [ 3]    1d200:  save        %sp, -240, %sp
0.    0.          [ 3]    1d204:  ld          [%i0], %i5
0.    0.          [ 3]    1d208:  st          %i5, [%fp - 24]
              4.
              5.          real*8 a(n,n), b(n,n), c(n,n)
              6.
              7.          call initarray(n,a,b,c)
              8.
0.    12.679       <inclusive metrics for slave threads>
              9. c$omp  parallel shared(a,b,c,n) private(i,j,k)
0.    3.653       <inclusive metrics for slave threads>
             10. c$omp  do
```

The following shows an example of the annotated disassembly displayed when an outline function is called:

```
             43.          else
             44.          {
             45.                  printf("else reached\n");
0.    2.522       <inclusive metrics for outlined functions>
```

Branch Target

An artificial line, `<branch target>`, shown in the annotated disassembly listing, corresponds to a PC of an instruction where the backtracking to find its effective address fails because the backtracking algorithm runs into a branch target.

Viewing Source/Disassembly Without An Experiment

You can view annotated source code and annotated disassembly code using the `er_src` utility, without running an experiment. The display is generated in the same way as in the Analyzer, except that it does not display any metrics. The syntax of the `er_src` command is

```
er_src [ -func | -{source,src} item tag | -disasm item tag |
-{cc,scc,dcc} com_spec | -outfile filename | -V ] object
```

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. *item* can also be specified in the form *functon'file'*, in which case `er_src` displays the source or disassembly of the named function in the source context of the named file

tag is an index used to determine which *item* is being referred to when multiple functions have the same name. It is required, but is ignored if not necessary to resolve the function.

The special item and tag, `all -1`, tells `er_src` to generate the annotated source or disassembly for all functions in the object.

Note – The output generated as a result of using `all -1` on executables and shared objects may be very large.

The following sections describe the options accepted by the `er_src` utility.

`-func`

List all the functions from the given object.

`-{source,src}` *item tag*

Show the annotated source for the listed *item*.

-disasm *item tag*

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.

-{c,scc,dcc} *com-spec*

Specify which classes of compiler commentary classes to show. *com-spec* is a list of classes separated by colons. The *com-spec* is applied to source compiler commentary if the -scc option is used, to disassembly commentary if the -dcc option is used, or to both source and disassembly commentary if -c is used. See "Commands That Control the Source and Disassembly Listings" on page 124 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 an .er.rc file in the user's home directory, if present, then an .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 Analyzer and the er_print utility, but only the settings for source and disassembly compiler commentary are used by the er_src utility. See "Commands That Set Defaults" on page 135 for a description of the defaults files. Commands in a defaults file other than scc and dcc are ignored y the er_src utility.

-outfile *filename*

Open the file *filename* for output of the listing. By default, or if the filename is a dash (-), output is written to stdout.

-V

Print the current release version.

CHAPTER 9

Manipulating Experiments

This chapter describes the utilities which are available for use with the Collector and Performance Analyzer.

This chapter covers the following topics:

- Manipulating Experiments
- Other Utilities

Manipulating Experiments

Experiments are stored in a directory that is created by the Collector. To manipulate experiments, you can use the usual UNIX® commands `cp`, `mv` and `rm` and apply them to the directory. You cannot do so for experiments from releases earlier than Forte Developer 7 (Sun™ ONE Studio 7, Enterprise Edition for Solaris). Three utilities which behave like the UNIX commands have been provided to copy, move and delete experiments. These utilities are `er_cp`(1), `er_mv`(1) and `er_rm`(1), and are described below.

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.

Copying Experiments With the `er_cp` Utility

Two forms of the `er_cp` command exist:

```
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 the er_mv eutility exits with an error message. The -V option prints the version of the er_cp utility. This command does not copy experiments created with software releases earlier than the Forte Developer 7 release.

Moving Experiments With the er_mv Utility

Two forms of the er_mv command exist:

```
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 the er_mv utiilty 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, the er_mv utility exits with an error message. The -V option prints the version of the er_mv utility. This command does not move experiments created with software releases earlier than the Forte Developer 7 release.

Deleting Experiments With the er_rm Utility

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 syntax of the er_rm command is as follows:

```
er_rm [-f] [-V] experiment-list
```

The -f option suppresses error messages and ensures successful completion, whether or not the experiments are found. The -V option prints the version of the er_rm utility. This command removes experiments created with software releases earlier than the Forte Developer 7 release.

Other Utilities

Some other utilities 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 [-qAF] experiment
er_archive -V
```

The `er_archive` utility is automatically run when an experiment completes normally, or when the Performance Analyzer or `er_print` utility is started 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, the `er_archive` utility might not be run by the Collector. If you want to examine the experiment from an abnormally-terminated run on a different machine from the one on which it was recorded, you must run the `er_archive` utility on the experiment, on the machine on which the data was recorded. To ensure that the load objects are available on the machine to which the experiment is copied, use the `-A` option.

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 cannot be found when the `er_archive` utility is run, or if it has a time stamp differing from that recorded in the experiment, or if the `er_archive` utility 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 the `er_archive` utility 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 Performance Analyzer or output from the `er_print` utility.

`-A`

Request writing of all load objects into the experiment. This argument can be used to generate experiments that are more readily copied to a machine other than the one on which the experiment was recorded.

`-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 for the `er_archive` utility and exit.

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 the Solaris OS. `tcov` is a code coverage tool provided with the Sun Studio software.

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 Collector and Performance Analyzer. See Chapter 3 and the online help 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

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 function under these column headings:

   - `%Time`—The percentage of the total CPU time consumed by this function.
   - `Seconds`—The total CPU time accounted for by this function.
   - `Cumsecs`—A running sum of the number of seconds accounted for by this function and those listed before it.
   - `#Calls`—The number of times this function is called.
   - `msecs/call`—The average number of milliseconds this function consumes each time it is called.
   - `Name`—The name of the function.

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()` function; after that, most of the CPU time is spent in the `_strlen()` library function. 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 212. In this particular case, improving the algorithm also reduces the number of calls.

Note – For the Solaris 7 OS and the Solaris 8 OS, 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 performance analyzer tutorial on the `developers.sun.com` web site 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 – The profile of CPU time is accurate for programs that use multiple CPUs for the Solaris 7 OS and the Solaris 8 OS, 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 function 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.

```
                                    called/total
                                    parents
index   %time    self    descendants  called+self       name                    index
                                    called/total
                                    children

-----------------------------------------------
```

```
                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 information on how often a program executes segments of code. It produces a copy of the source file, annotated with execution frequencies. The code can be annotated at the basic block level or the source line level. A basic block is a linear segment of source code with no branches. The statements in a basic block are executed the same number of times, so a count of basic block executions also tells you how many times each statement in the block was executed. The `tcov` utility 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 basic block 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);
                            }
                }
```

The `tcov` utility places a summary like the following at the end of the annotated program listing. The statistics for the most frequently executed basic blocks are listed in order of execution frequency. The line number is the number of the first line in the block.

The following is the summary for the `index.assist` program:

```
              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 functions 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 functions 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 functions automatically when a program is linked for tcov profiling. At program exit, these functions 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 functions 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 Functions

The following error messages may be reported by the tcov runtime functions:

- 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 223).

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

accessible documentation, 23
`addpath` command, 128
address spaces, text and data regions, 167
aggressive backtracking, 101
aliased functions, 168
alternate entry points in Fortran functions, 169
alternate source context, 125
`analyzer` command
  font size (`-f`) option, 95
  help (`-h`) option, 95
  JVM options (`-J`) option, 95
  JVM path (`-j`) option, 94
  verbose (`-v`) option, 95
  version (`-V`) option, 95
Analyzer, *See* Performance Analyzer
annotated disassembly code, *See* disassembly code, annotated
annotated source code, *See* source code, annotated
API, Collector, 54
appending path to files, 128
arc, call graph, defined, 212
archiving load objects in experiments, 76, 83
asynchronous I/O library, interaction with data collection, 54
attaching the Collector to a running process, 85
attributed metrics
  defined, 45
  effect of recursion on, 47
  illustrated, 46
  use of, 46

B

body functions, compiler-generated
  defined, 161
  displayed by the Performance Analyzer, 171, 195
  names, 161, 195
  propagation of inclusive metrics, 164
branch target, 201

C

C compiler option, `xhwcprof`, 101
call stack, 102
  default alignment and depth in the Timeline tab, 137
  defined, 151
  effect of tail-call optimization on, 155
  in the Event tab, 103
  in Timeline tab, 102
  incomplete unwind, 166
  mapping addresses to program structure, 167
  unwinding, 152
callers-callees metrics
  attributed, defined, 45
  displaying list of in `er_print` utility, 133
  printing for a single function in `er_print` utility, 123
  printing in `er_print` utility, 122
  selecting in `er_print` utility, 122
  sort order in `er_print` utility, 123
Callers-Callees tab, 98, 106
clock-based profiling
  accuracy of metrics, 148

collecting data in dbx, 79
collecting data with the collect command, 70
comparison with gethrtime and gethrvtime, 148
data in profile packet, 145
default metrics, 97
defined, 35
distortion due to overheads, 147
interval, *See* profiling interval
metrics, 35, 146
cloned functions, 170, 199
collect command
archiving (-A) option, 76
clock-based profiling (-p) option, 70
collecting data with, 69
data collection options, 69, 94
data limit (-L) option, 77
dry run (-n) option, 77
experiment control options, 73
experiment directory (-d) option, 76
experiment group (-g) option, 76
experiment name (-o) option, 77
follow descendant processes (-F) option, 73
hardware counter overflow profiling (-h) option, 70
heap tracing (-H) option, 72
Java version (-j) option, 74
listing the options of, 69
miscellaneous options, 77
MPI tracing (-m) option, 72
output options, 76
pause and resume data recording (-y) option, 75
periodic sampling (-S) option, 72
readme display (-R) option, 78
record sample point (-l) option, 75
stop target after exec (-x) option, 75
synchronization wait tracing (-s) option, 71
syntax, 69
verbose (-v) option, 78
version (-V) option, 78
with ppgsz command, 91
Collect Experiment
Preview command, 107
Collector
API, using in your program, 54, 56
attaching to a running process, 85
defined, 30, 34
disabling in dbx, 82
enabling in dbx, 82
running in dbx, 78
running with the collect command, 69
collectorAPI.h, 56
common subexpression elimination, 183
compiler commentary, 99
classes defined, 126
cloned functions, 199
common subexpression elimination, 183
description of, 182
filtering in er_src utility, 203
filtering types displayed, 183
inlined functions, 185
loop optimizations, 184
parallelizaton, 186
selecting for annotated disassembly listing in er_print utility, 127
selecting for annotated source listing in er_print utility, 126
compiler optimization
inlining, 185
parallelization, 186
compiler-generated body functions
defined, 161
displayed by the Performance Analyzer, 171, 195
names, 161, 195
propagation of inclusive metrics, 164
compilers, accessing, 19
compiling
affects of optimization on program analysis, 50
debug symbol information format, 50
effect of static linking on data collection, 50
for gprof, 212
for Lines analyses, 50
for prof, 210
for tcov, 215
for tcov Enhanced, 221
Java programming language, 51
linking for data collection, 50
source code for annotated Source and Disassembly, 50
static linking of libraries, 50
copying an experiment, 206
correlation, effect on metrics, 147
CPU filtering, 107
CPUs

listing selected, in er_print utility, 130
selecting in er_print utility, 132

D

data collection
controlling from your program, 54
disabling from your program, 58
disabling in dbx, 82
dynamic memory allocation effects, 51
enabling in dbx, 82
from MPI programs, 87
linking for, 50
MPI program, using dbx, 90
MPI program, using the collect command, 90
pausing for collect command, 75
pausing from your program, 57
pausing in dbx, 83
preparing your program for, 51
program control of, 54
rate of, 67
resuming for collect command, 75
resuming from your program, 58
resuming in dbx, 83
segmentation faults, 52
using dbx, 78
using the collect command, 69
data objects
<Scalar> descriptors, 176
<Total> descriptor, 176
defined, 175
in hardware counter overflow experiments, 128
layout, 101
scope, 175
set the sort metric for, 129
Data Space Display, 101
Data Space Display mode, 108
data types, 34
clock-based profiling, 35
default, in the Timeline tab, 137
hardware counter overflow profiling, 37
heap tracing, 41
MPI tracing, 42
synchronization wait tracing, 40
data_objects command, 128
data_olayout command, 129
data_osingle command, 128
data_osort command, 129
DataLayout tab, 101
datamode on command, 101
datamode setting, 108
DataObjects tab, 101
dbx
collecting data under MPI, 90
running the Collector in, 78
dbx collector subcommands
archive, 83
dbxsample, 82
disable, 82
enable, 82
enable_once (obsolete), 84
hwprofile, 80
limit, 83
pause, 83
profile, 79
quit (obsolete), 84
resume, 83
sample, 82
sample record, 83
show, 84
status, 84
store, 84
store filename (obsolete), 84
synctrace, 81
default metrics, 97
defaults, setting in a defaults file, 135
descendant experiments, 93
descendant processes
collecting data for all followed, 73
collecting data for selected, 85
experiment location, 65
experiment names, 66
followed by Collector, 63
limitations on data collection for, 63
directives, parallelization, microtasking library calls from, 161
disassembly code, annotated
branch target, 201
cloned functions, 199
description, 189
for cloned functions, 170, 199
hardware counter metric attribution, 193
HotSpot-compiled instructions, 197
inclusive metrics, 201
instruction issue dependencies, 190

interpreting, 190
Java native methods, 198
location of executable, 67
metric formats, 188
printing in er_print utility, 126
setting preferences in er_print utility, 127
setting the highlighting threshold in er_print utility, 128
viewing with er_src utility, 202
Disassembly tab, 100
disk space, estimating for experiments, 67
documentation index, 22
documentation, accessing, 22 to 24
DTrace driver
described, 109
setting up access to, 110
dynamically compiled functions
Collector API for, 59
definition, 172, 196

E

entry points, alternate, in Fortran functions, 169
environment variables
JAVA_PATH, 63
JDK_HOME, 63
LD_LIBRARY_PATH, 87
LD_PRELOAD, 86
PATH, 63
SUN_PROFDATA, 223
SUN_PROFDATA_DIR, 224
TCOVDIR, 215, 224
.er.rc file, 97, 101, 108, 203
er_archive utility, 207
er_cp utility, 206
er_export utility, 208
er_kernel utility, 109
er_mv utility, 206
er_print commands
addpath, 128
allocs, 124
callers-callees, 122
cmetric_list, 133
cmetrics, 122
cpu_list, 130
cpu_select, 132
csingle, 123
csort, 123
data_objects, 128
data_olayout, 129
data_osingle, 128
data_osort, 129
dcc, 127
disasm, 126
dmetrics, 136
dsort, 136
exp_list, 129
fsingle, 121
fsummary, 121
functions, 119
header, 134
help, 138
javamode, 134
leaks, 124
limit, 134
lines, 125
lsummary, 125
lwp_list, 130
lwp_select, 131
mapfile, 138
metric_list, 133
metrics, 120
name, 134
object_list, 132
object_select, 132
objects, 135
outfile, 133
overview, 135
pcs, 124
psummary, 125
quit, 138
sample_list, 130
sample_select, 131
scc, 126
script, 138
setpath, 128
sort, 121
source, 125
src, 125
statistics, 135
sthresh, 127, 128
thread_list, 130
thread_select, 132
tldata, 137
tlmode, 137
Version, 138
version, 138

er_print utility
command-line options, 116
commands, *See* er_print commands
metric keywords, 118
metric lists, 116
purpose, 115
syntax, 116
er_rm utility, 206
er_src utility, 202
errors reported by tcov, 219
event marker, 102
Event tab, 102, 103
events
default display type in the Timeline tab, 137
displayed in Timeline tab, 101
exclusive metrics
defined, 45
for PLT instructions, 153
how computed, 152
illustrated, 46
use of, 45
execution statistics
comparison of times with the <Total> function, 148
printing in er_print utility, 135
experiment directory
default, 65
specifying in dbx, 84
specifying with collect command, 76
experiment filtering, 106
experiment groups
adding, 94
creating, 93
default name, 65
defined, 65
multiple, 93
name restrictions, 65
preview, 94
removing, 206
specifying name in dbx, 84
specifying name with collect command, 76
experiment names
default, 65
MPI default, 66, 89
MPI, using MPI_comm_rank and a script, 90
restrictions, 65
specifying in dbx, 84
experiments
See also experiment directory; experiment groups; experiment names
adding, 94
appending current path, 128
archiving load objects in, 76, 83
copying, 206
data aggregation, 94
default name, 65
defined, 65
descendant, 93
groups, 65
header information in er_print utility, 134
limiting the size of, 77, 83
listing in er_print utility, 129
location, 65
moving, 66, 206
moving MPI, 89
MPI storage issues, 88
multiple, 93
naming, 65
opening, 93
preview, 94
removing, 206
setting mode for Java, 134
setting path to find files, 128
storage requirements, estimating, 67
terminating from your program, 58
where stored, 76, 84
Experiments tab, 102
explicit multithreading, 155

F

fast traps, 154
filter CPU, 107
Filter Data dialog box, 106
filter experiment, 106
filter LWPs, 107
filter sample, 106
filter threads, 106
Find tool, 105
Fortran
alternate entry points, 169
Collector API, 54
subroutines, 168
frames, stack, *See* stack frames
function calls

between shared objects, 153
imputed, in OpenMP programs, 164
in single-threaded programs, 152
recursive, metric assignment to, 47
function list
compiler-generated body function, 195
printing in er_print utility, 119
sort order, specifying in er_print utility, 121
function names, C++, choosing long or short form in er_print utility, 134
function PCs, aggregation, 100, 101, 106
function reordering, 108
function-list metrics
displaying list of in er_print utility, 133
selecting default in .er.rc file, 136
selecting in er_print utility, 120
setting default sort order in .er.rc file, 136
functions
@plt, 153
address within a load object, 168
aliased, 168
alternate entry points (Fortran), 169
body, compiler-generated, *See* body functions, compiler-generated
cloned, 170, 199
Collector API, 54, 59
definition of, 168
dynamically compiled, 59, 172, 196
global, 168
inlined, 170
MPI, traced, 42
non-unique, names of, 168
outline, 172, 194
static, in stripped shared libraries, 169, 200
static, with duplicate names, 168
system library, interposition by Collector, 52
<Total>, 174
<Unknown>, 173
variation in addresses of, 167
wrapper, 169
Functions tab, 97, 106

G

gprof
limitations, 212
output from, interpreting, 213
summary, 209
using, 212

H

hardware counter attribute options, 71
hardware counter library, libcpc.so, 62
hardware counter list
description of fields, 38
obtaining with collect command, 69
obtaining with dbx collector command, 80
raw counters, 39
well-known counters, 38
hardware counter metrics, displayed in DataObjects tab, 101
hardware counter overflow profiling
collecting data with collect command, 70
collecting data with dbx, 80
data in profile packet, 149
default metrics, 98
defined, 37
hardware counter overflow value
consequences of too small or too large, 149
defined, 37
experiment size, effect on, 67
setting in dbx, 80
setting with collect, 71
hardware counters
choosing with collect command, 70
choosing with dbx collector command, 80
counter names, 70
data objects and metrics, 128
list described, 38
obtaining a list of, 69, 80
overflow value, 37
heap tracing
collecting data in dbx, 81
collecting data with collect command, 72
default metrics, 98
metrics, 41
preloading the Collector library, 86
high metric values
in annotated disassembly code, 128
in annotated source code, 127

I

inclusive metrics
defined, 45
effect of recursion on, 47
for outlined functions, 201
for PLT instructions, 153

for slave threads, 201
how computed, 152
illustrated, 46
use of, 46
index lines, 180
in Disassembly tab, 100, 190
in er_print utility, 125, 126
in Source tab, 99, 180, 189
index lines, special
compiler-generated body functions, 195
HotSpot-compiled instructions, 197
instructions without line numbers, 197
Java native methods, 198
outline functions, 194
inlined functions, 170
input file
terminating in er_print utility, 138
to er_print utility, 138
instruction issue
delay, 192
grouping, effect on annotated disassembly, 190
intermediate files, use for annotated source listings, 166
interposition by Collector on system library functions, 52
interval, profiling, *See* profiling interval
interval, sampling, *See* sampling interval

J

Java
dynamically compiled methods, 59, 172
memory allocations, 41
monitors, 40
profiling limitations, 63
setting er_print display output for, 134
Java Virtual Machine path, analyzer command option, 94
JAVA_PATH environment variable, 63
javamode command, 134
javamode setting, 108
JDK_HOME environment variable, 63
--jdkhome analyzer command option, 94
JVM versions, 63

K

kernel clock profiling, 110
kernel experiment
field label meanings, 113
types of data, 109
kernel profile, analyzing, 113
kernel profiling
profiling a specific process or kernel thread, 113
profiling the kernel and load together, 112
profiling under load, 111
setting up your system for, 109
keywords, metric, er_print utility, 118

L

LD_LIBRARY_PATH environment variable, 87
LD_PRELOAD environment variable, 86
leaf PC, defined, 151
Leak tab, 104
LeakList tab, 102
leaks, memory, definition, 41
Legend tab, 102, 103
libaio.so, interaction with data collection, 54
libcollector.h, 55
as part of C and C++ interface to collector, 55
as part of Java programming language interface to collector, 56
libcollector.so shared library
preloading, 86
using in your program, 54
libcpc.so, use of, 62
libfcollector.h, 55
libraries
collectorAPI.h, 56
interposition on, 52
libaio.so, 54
libcollector.so, 54, 86
libcpc.so, 53, 62
libthread.so, 52, 155, 156, 162
MPI, 53, 87
static linking, 50
stripped shared, and static functions, 169, 200
system, 52
limitations
descendant process data collection, 63
experiment group names, 65
experiment name, 65
Java profiling, 63
profiling interval value, 60

tcov, 215
limiting output in er_print utility, 134
limiting the experiment size, 77, 83
Lines tab, 100, 106
load objects
addresses of functions, 168
contents of, 167
defined, 167
listing selected, in er_print utility, 132
printing list in er_print utility, 135
selecting in er_print utility, 132
symbol tables, 167
writing layouts of, 129
lock file management
tcov, 219
tcov Enhanced, 222
loop optimizations, 184
LWPs
creation by threads library, 155
filtering, 107
listing selected, in er_print utility, 130
selecting in er_print utility, 131

M

man pages, accessing, 19
MANPATH environment variable, setting, 21
mapfile
generating, 108
generating with er_print utility, 138
memory allocations, 41
and leaks, 102
effects on data collection, 51
memory leaks, definition, 41
methods, *See* functions
metrics
attributed, 98
attributed, *See* attributed metrics
clock-based profiling, 35, 146
default, 97
defined, 33
effect of correlation, 147
exclusive, *See* exclusive metrics
function-list, *See* function-list metrics
hardware counter, attributing to instructions, 193
heap tracing, 41
inclusive and exclusive, 97, 98
inclusive, *See* inclusive metrics
interpreting for instructions, 190
interpreting for source lines, 187
memory allocation, 41
MPI tracing, 42
synchronization wait tracing, 40
threshold, 100
threshold, setting, 99
time precision, 97
timing, 35
microstates, 103
contribution to metrics, 146
switching, 154
microtasking library routines, 161
moving an experiment, 66, 206
MPI experiments
default name, 66
moving, 89
storage issues, 88
MPI programs
attaching to, 87
collecting data from, 87
collecting data with collect command, 90
collecting data with dbx, 90
experiment names, 66, 88, 89
experiment storage issues, 88
MPI tracing
collecting data in dbx, 81
collecting data with collect command, 72
data in profile packet, 151
functions traced, 42
interpretation of metrics, 151
metrics, 42
preloading the Collector library, 86
multithreaded applications
attaching the Collector to, 85
execution sequence, 162
multithreading
explicit, 155
parallelization directives, 161

N

naming an experiment, 65
networked disks, 65
nfs, 65
non-unique function names, 168

O

OpenMP parallelization, 161, 186
optimizations
  common subexpression elimination, 183
  program analysis affect of, 50
  tail-call, 155
options, command-line, `er_print` utility, 116
outline functions, 172, 194
output file, in `er_print` utility, 133
overflow value, hardware counter, *See* hardware counter overflow value
overview data, printing in `er_print` utility, 135

P

parallel execution
  call sequence, 162
  directives, 161, 186
`PATH` environment variable, 63
  setting, 20
path to files, 128
pausing data collection
  for `collect` command, 75
  from your program, 57
  in `dbx`, 83
PCs
  defined, 151
  from PLT, 153
  ordered list in `er_print` utility, 124
PCs tab, 101, 106
Performance Analyzer
  Callers-Callees tab, 98, 106
  command-line options, 94
  DataLayout tab, 101
  DataObjects tab, 101
  defaults, 108
  defined, 30
  definition, 93
  Disassembly tab, 100
  Event tab, 102, 103
  Experiments tab, 102
  File menu, 96
  Filter Data dialog box, 106
  Find tool, 105
  Functions tab, 97, 106
  Help menu, 96
  Leak tab, 104
  LeakList tab, 102
  Legend tab, 102, 103
  Lines tab, 100, 106
  PCs tab, 101, 106
  recording an experiment, 94
  Show/Hide Functions, 105
  Source tab, 98
  starting, 93
  Statistics tab, 102
  Summary tab, 101, 103
  Timeline menu, 96
  Timeline tab, 101, 103
  View menu, 96
performance data, conversion into metrics, 33
performance metrics, *See* metrics
PLT (Program Linkage Table), 153
`@plt` function, 153
`ppgsz` command, 91
preloading `libcollector.so`, 86
printing the current path, 128
process address-space text and data regions, 167
`prof`
  limitations, 212
  output from, 211
  summary, 209
  using, 210
profile bucket, `tcov` Enhanced, 221, 223
profile packet
  clock-based data, 145
  hardware counter overflow data, 149
  MPI tracing data, 151
  size of, 67
  synchronization wait tracing data, 149
profiled shared libraries, creating
  for `tcov`, 218
  for `tcov` Enhanced, 222
profiling interval
  defined, 35
  experiment size, effect on, 67
  limitations on value, 60
  setting with `dbx collector` command, 79
  setting with the `collect` command, 70, 79
profiling, defined, 34
program counter (PC), defined, 151
program execution
  call stacks described, 151
  explicit multithreading, 155
  OpenMP parallel, 162

shared objects and function calls, 153
signal handling, 153
single-threaded, 152
tail-call optimization, 155
traps, 154
Program Linkage Table (PLT), 153
program structure, mapping call stack addresses to, 167

R

raw hardware counters, 38, 39
recursive function calls
apparent, in OpenMP programs, 165
metric assignment to, 47
removing an experiment or experiment group, 206
restrictions, *See* limitations
resuming data collection
for `collect` command, 75
from your program, 58
in `dbx`, 83

S

sample filtering, 106
sample points, displayed in Timeline tab, 101
samples
circumstances of recording, 43
defined, 44
information contained in packet, 43
interval, *See* sampling interval
listing selected, in `er_print` utility, 130
manual recording in `dbx`, 83
manual recording with `collect`, 75
periodic recording in `dbx`, 82
periodic recording with `collect` command, 72
recording from your program, 57
recording when `dbx` stops a process, 82
selecting in `er_print` utility, 131
Sampling Collector, *See* Collector
sampling interval
defined, 43
setting in `dbx`, 82
setting with the `collect` command, 72
`<Scalar>` data object descriptor, 176
segmentation faults during data collection, 52
Set Data Presentation dialog box, Data Space Display, 101
`setpath` command, 128
`setuid`, use of, 54
shared objects, function calls between, 153
shell prompts, 18
Show/Hide Functions dialog box, 105
signal handlers
installed by Collector, 53, 153
user program, 53
signals
calls to handlers, 153
profiling, 53
profiling, passing from `dbx` to `collect` command, 75
use for manual sampling with `collect` command, 75
use for pause and resume with `collect` command, 75
single-threaded program execution, 152
sort order
callers-callees metrics, in `er_print` utility, 123
function list, specifying in `er_print` utility, 121
source code, annotated
cloned functions, 199
compiler commentary, 182
compiler-generated body functions, 195
description, 179, 186
discerning annotations from source, 180
for cloned functions, 170
from `tcov`, 217
index lines, 180
instructions without line numbers, 197
interpreting, 187
location of source files, 67
metric formats, 188
outline functions, 194
printing in `er_print` utility, 125
setting compiler commentary classes in `er_print` utility, 126
setting the highlighting threshold in `er_print` utility, 127
use of intermediate files, 166
viewing in Performance Analyzer, 179
viewing with `er_src` utility, 202
source code, compiler commentary, 99
source lines, ordered list in `er_print` utility, 125
Source tab, 98
stack frames

defined, 152
from trap handler, 154
reuse of in tail-call optimization, 155
static functions
duplicate names, 168
in stripped shared libraries, 169, 200
static linking, effect on data collection, 50
Statistics tab, 102
storage requirements, estimating for experiments, 67
subroutines, *See* functions
summary metrics
for a single function, printing in `er_print` utility, 121
for all functions, printing in `er_print` utility, 121
Summary tab, 101, 103
`SUN_PROFDATA` environment variable, 223
`SUN_PROFDATA_DIR` environment variable, 224
symbol tables, load-object, 167
synchronization delay events
data in profile packet, 149
defined, 40
metric defined, 40
synchronization delay tracing
default metrics, 97
synchronization wait time
defined, 40, 149
metric, defined, 40
with unbound threads, 149
synchronization wait tracing
collecting data in `dbx`, 81
collecting data with `collect` command, 71
data in profile packet, 148
defined, 40
metrics, 40
preloading the Collector library, 86
threshold, *See* threshold, synchronization wait tracing
wait time, 40, 149
syntax
`er_archive` utility, 207
`er_export` utility, 208
`er_print` utility, 116
`er_src` utility, 202

T

tail-call optimization, 155
`tcov`
annotated source code, 217
compiling a program for, 215
errors reported by, 219
limitations, 215
lock file management, 219
output, interpreting, 217
profiled shared libraries, creating, 218
summary, 209
using, 215
`tcov` Enhanced
advantages of, 221
compiling a program for, 221
lock file management, 222
profile bucket, 221, 223
profiled shared libraries, creating, 222
using, 221
`TCOVDIR` environment variable, 215, 224
thread filtering, 106
threads
bound and unbound, 155, 165
creation of, 155
library, 52, 155, 156, 162
listing selected, in `er_print` utility, 130
main, 162
scheduling of, 155, 161
selecting in `er_print` utility, 132
system, 148, 162
wait mode, 165
worker, 155, 162
threshold, highlighting
in annotated disassembly code, `er_print` utility, 128
in annotated source code, `er_print` utility, 127
threshold, synchronization wait tracing
calibration, 40
defined, 40
effect on collection overhead, 149
setting with `dbx collector`, 81
setting with the `collect` command, 72, 81
time metrics, precision, 97
Timeline menu, 96
Timeline tab, 101, 103
TLB (translation lookaside buffer) misses, 154, 193
`<Total>` data object descriptor, 176

<Total> function
comparing times with execution statistics, 148
described, 174
traps, 154
typographic conventions, 17

U

<Unknown> function
callers and callees, 173
mapping of PC to, 173
unwinding the call stack, 152

V

version information
for collect command, 78
for er_cp utility, 206
for er_mv utility, 206
for er_print utility, 138
for er_rm utility, 206
for er_src utility, 203

W

wait time, *See* synchronization wait time
well-known hardware counters, 38
wrapper functions, 169

X

-xdebugformat, setting debug symbol information format, 50
xhwcprof C compiler option, 101