Program Performance Analysis Tools

Sun™ ONE Studio 8

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part Number 817-0922-10
May 2003, Revision A

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

Please
Recycle

Contents

Figures

Tables

Before You Begin

This manual describes the performance analysis tools that are available with the Sun™ Open Net Environment (Sun ONE) Studio Compiler Collection product.

- 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 Java™, the Solaris™ operating environment, 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 is a tutorial that demonstrates how to use the Collector and Performance Analyzer to assess the performance of five example programs.

Chapter 3 describes the data collected by the Collector and how the data is converted into metrics of performance.

Chapter 4 describes how to use the Collector to collect timing data, synchronization delay data, and hardware event data from your program.

Chapter 5 describes the features of the Performance Analyzer graphical user interface. Note: you must have a license to use the Performance Analyzer.

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

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 8 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`

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 Compiler Collection Tools and Man Pages

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

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` 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 ONE Studio Compiler Collection components are installed in the `/opt` directory. If your software is not installed in the `/opt` 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. **Review the output to find a string of paths that contain `/opt/SUNWspro/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. **Add the following to your** PATH **environment variable.**

   `/opt/SUNWspro/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** `dbx` **man page by typing the following at a command prompt.**

   ```
   % man dbx
   ```

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

   If the dbx(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

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

   `/opt/SUNWspro/man`

Accessing Compiler Collection 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`.

  If your software is not installed in the `/opt` directory, 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.

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 and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any 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 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`
Release notes	HTML at `http://docs.sun.com`

Related Compiler Collection Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html` and `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
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.
Language user's guides	Describe compilation and compiler options.

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 environment.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
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://www.sun.com/developers/studio` and click the Compiler Collection link to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of compiler collection 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://www.sun.com/developers/`.

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`

Sun Welcomes Your Comments

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

`docfeedback@sun.com`

Please include the part number (817-0922-10) of your document in the subject line of your email.

CHAPTER 1

Overview of Program Performance Analysis Tools

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

This manual deals primarily with the 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 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, 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 3 for information about the data collected and Chapter 4 for detailed information about the Collector. The Collector can be run from the IDE, from the `dbx` command line tool, and using the `collect` command.

The Performance Analyzer 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: timing metrics, hardware counter metrics, synchronization delay metrics, memory allocation metrics, and 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 improve the order of function loading in the program's address space. See Chapter 5 for detailed information about the Performance Analyzer, and Chapter 6 for information about the command-line analysis tool, `er_print`. 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 8 for more information).

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 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 LWP, 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 more 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, and statistics for the process. The Performance Analyzer can be navigated from the keyboard as well as using a mouse.

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

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`.

This manual also includes information about the following performance tools:

- `prof` and `gprof`

  `prof` and `gprof` are UNIX® tools for generating profile data and are included with the Solaris™ 7, 8 and 9 operating environments (SPARC® *Platform Edition*). Both tools are also provided and supported on the x86 platform.

- `tcov`

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

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

Note – For information on starting the Performance Analyzer from the IDE, see the Program Performance Analysis Tools Readme, which is available through the documentation index at `file:/opt/SUNWspro/docs/index.html`. If the Sun ONE Studio 8 software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

CHAPTER 2

Learning to Use the Performance Tools

This chapter shows you how to use the Collector and the Performance Analyzer by means of a tutorial. The tutorial has three main purposes:

- To provide simple examples of performance problems and how they can be identified.
- To demonstrate the capabilities of the Performance Analyzer.
- To show how the Performance Analyzer presents performance data and how it handles various code constructions.

Note – For information on starting the Performance Analyzer from the IDE, see the Program Performance Analysis Tools Readme, which is available through the documentation index at `file:/opt/SUNWspro/docs/index.html`. If the Sun ONE Studio software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Five example programs are provided that illustrate the capabilities of the Performance Analyzer in several different situations.

- Example 1: Basic Performance Analysis. This example demonstrates the use of timing data to identify a performance problem, shows how time is attributed to functions, source lines and instructions, and shows how the Performance Analyzer handles recursive calls, dynamic loading of object modules and descendant processes. The example illustrates the use of the main Analyzer displays: the Functions tab, the Callers-Callees tab, the Source tab, the Disassembly tab and the Timeline tab. The example program, `synprog`, is written in C.
- Example 2: Analyzing the Performance of a Mixed Java/C++ Application. This example demonstrates how the analyzer handles interpreted and dynamically-compiled Java methods. The example program, `jsynprog`, is written in the Java programming language and makes calls to native code using JNI.

- Example 3: OpenMP Parallelization Strategies. This example demonstrates the efficiency of different approaches to parallelization of a Fortran program, `omptest`, using OpenMP directives.
- Example 4: Locking Strategies in Multithreaded Programs. This example demonstrates the efficiency of different approaches to scheduling of work among threads and the effect of data management on cache performance, making use of synchronization delay data. The example uses an explicitly multithreaded C program, `mttest`, that is a model of a client/server application.
- Example 5: Cache Behavior and Optimization. This example demonstrates the effect of memory access and compiler optimization on execution speed for a Fortran 90 program, `cachetest`. The example illustrates the use of hardware counter data and compiler commentary for performance analysis.

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

The instructions for collecting performance data in this tutorial are given only for the command line. For most of the examples you can also use the IDE to collect performance data. To collect data from the IDE, you use the `dbx` Debugger and the Performance Toolkit submenu of the Debug menu.

Setting Up the Examples for Execution

The examples are provided with the Sun™ ONE Studio 8 software release. The source code and makefiles for each of the example programs are in the Performance Analyzer example directory.

install-directory`/examples/analyzer`

The default for *install-directory* is `/opt/SUNWspro`. The Performance Analyzer example directory contains a separate subdirectory for each example, named `synprog`, `jsynprog`, `omptest`, `mttest` and `cachetest`.

To compile the examples with the default options:

1. **Ensure that** *install-directory*`/bin` **appears in your path before any other installation of the Sun ONE Studio or Forte Developer software.**

2. **Copy the files in one or more of the example subdirectories to your own work directory.**

```
% mkdir -p work-directory/example
% cp -r install-directory/examples/analyzer/example  work-directory
```

Choose *example* from the list of example subdirectory names given above. This tutorial assumes that your directory is set up as described in the preceding code box.

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

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

System Requirements

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

- `synprog` requires only a single CPU, but will run correctly on hardware with more than one CPU.
- `jsynprog` requires only a single CPU, but will run correctly on hardware with more than one CPU.
- `omptest` will run on SPARC® hardware with any number of CPUs, but the example (and provided Makefiles) assume 4 CPUs.
- `mttest` will run on SPARC hardware with any number of CPUs, but the example (and provided Makefiles) assume 4 CPUs. You should run the test under the Solaris 7 or 8 operating environment with the standard threads library. If you use the alternate threads library in the Solaris 8 operating environment or the threads library in the Solaris 9 operating environment some of the details of the example are different.
- `cachetest` requires that you run the program on UltraSPARC® III hardware with at least 160 Mbytes of memory.

Choosing Alternative Compiler Options

The default compiler options have been chosen to make the examples work in a particular way. Some of them can affect the performance of the program, such as the `-xarch` option, which selects the instruction set architecture. This option is set to

`native` so that you use the instruction set that is best suited to your computer. If you want to use a different setting, change the definition of the `ARCH` environment variable in the makefile.

If you run the examples on a SPARC platform with the default V7 architecture, the compiler generates code that calls the `.mul` and `.div` routines from `libc.so` rather than using integer multiply and divide instructions. The time spent in these arithmetic operations shows up in the `<Unknown>` function; see "The `<Unknown>` Function" on page 221 for more information.

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

Basic Features of the Performance Analyzer

This section describes some basic features of the Performance Analyzer.

The Performance Analyzer displays the Functions tab when it is started. If the default data options were used in the Collector, the Functions tab shows a list of functions with the default clock-based profiling metrics, which are:

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

The function list is sorted on exclusive CPU time by default. For a more detailed discussion of metrics, see "How Metrics Are Assigned to Program Structure" on page 97.

Selecting a function in the Functions tab and clicking the Callers-Callees tab displays information about the callers and callees of a function. The tab is divided into three horizontal panes:

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

In addition to exclusive and inclusive metrics, additional data can be viewed with the following tabs.

- The Callers-Callees tab displays attributed metrics for callers and callees. Attributed metrics are the parts of the inclusive metric of the selected function that are due to calls from a caller or calls to a callee.
- The lines tab shows a list of source lines and their metrics. The source lines are represented by the function name followed by the line number and the source file name.
- The PCs tab shows a list of program counter addresses and the metrics for the corresponding instructions. The PCs are represented by the function name and the offset relative to the start of the function.
- The Source tab displays the source code, if it is available, for the selected function, with performance metrics for each line of code.
- The Disassembly tab displays the instructions for the selected function with performance metrics for each instruction.
- The Timeline tab displays global timing data for each experiment and the data for each event recorded by the Collector. The data is presented for each LWP and each data type for each experiment.
- The LeakList tab shows a list of all the leaks and allocations that occurred in the program. Each leak entry includes the number of bytes leaked and the callstack for the allocation. Each allocation entry includes the number of bytes allocated and the call stack for the allocation.
- The Statistics tab shows totals for various system statistics summed over the selected experiments and samples, followed by the statistics for the selected samples of each experiment.
- The Experiments tab shows information on the experiments collected and on the load objects accessed by the collection target. The information includes any error or warning messages generated during the processing of the experiments or load objects.
- The Summary tab summarizes data for a load object, function, source line, or PC.
- The Event tab shows the available data for the selected event, including the event type, leaf function, LWP ID, thread ID, and CPU ID.
- The Legend tab shows the mapping of colors to functions for the display of events in the TimeLine tab.

For a complete description of each tab, see Chapter 5, *The Performance Analyzer Graphical User Interface.*

Example 1: Basic Performance Analysis

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

- "Simple Metric Analysis" on page 37 demonstrates how to use the function list, the annotated source code listing and the annotated disassembly code listing to do a simple performance analysis of two routines that shows the cost of type conversions.
- "Metric Attribution and the `gprof` Fallacy" on page 40 demonstrates the Callers-Callees tab and shows how time that is used in a low-level routine is attributed to its callers. `gprof` is a standard UNIX performance tool that properly identifies the function where the program is spending most of its CPU time, but in this case wrongly reports the caller that is responsible for most of that time. See Appendix A for a description of `gprof`.
- "The Effects of Recursion" on page 43 shows how time is attributed to callers in a recursive sequence for both direct recursive function calls and indirect recursive function calls.
- "Loading Dynamically Linked Shared Objects" on page 46 demonstrates the handling of load objects and shows how a function is correctly identified even if it is loaded in different locations at different times.
- "Descendant Processes" on page 48 demonstrates the use of the Timeline tab and filtering to analyze experiments on a program that creates descendant processes.

Collecting Data for `synprog`

Read the instructions in the sections, "Setting Up the Examples for Execution" on page 32 and "Basic Features of the Performance Analyzer" on page 34, if you have not done so. Compile `synprog` before you begin this example.

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

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

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

Simple Metric Analysis

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

1. **Locate `cputime()` and `icputime()` in the Functions tab.**

   You can use the Find tool to find the functions instead of scrolling the display.

   Compare the exclusive user CPU time for the two functions. Much more time is spent in `cputime()` than in `icputime()`.

2. **Choose File → Create New Window (Alt-F, N).**

   A new Analyzer window is displayed with the same data. Position the windows so that you can see both of them.

3. **In the Functions tab of the first window, click `cputime()` to select it, then click the Source tab.**

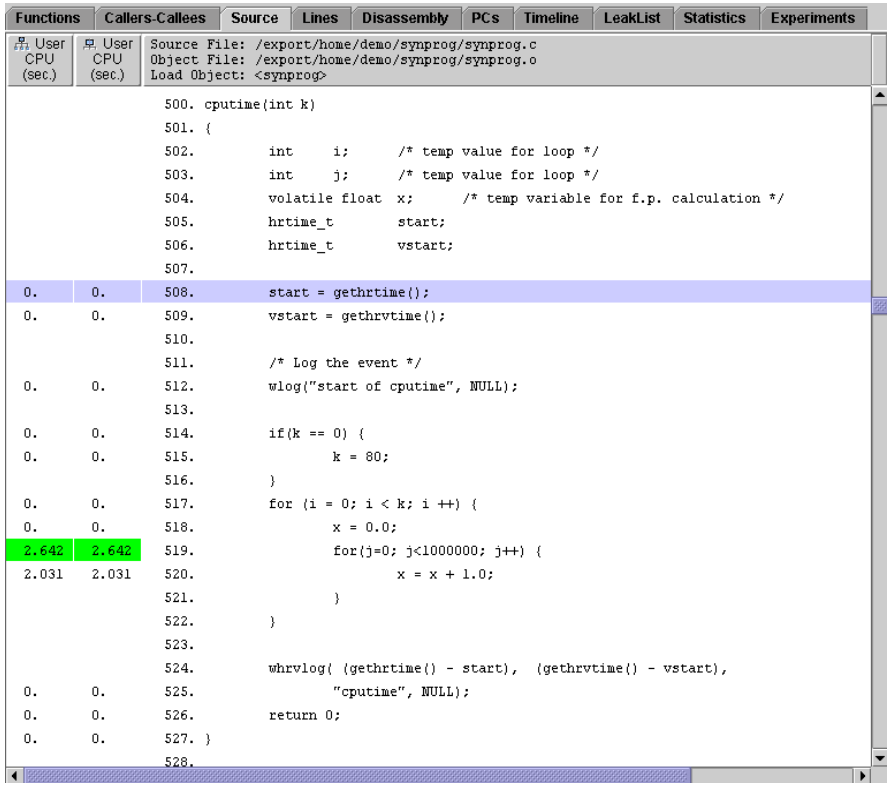


FIGURE 2-1 Source Tab Showing Annotated Source Code for Function `cputime`

4. **In the Functions tab of the second window, click `icputime()` to select it, then click the Source tab.**

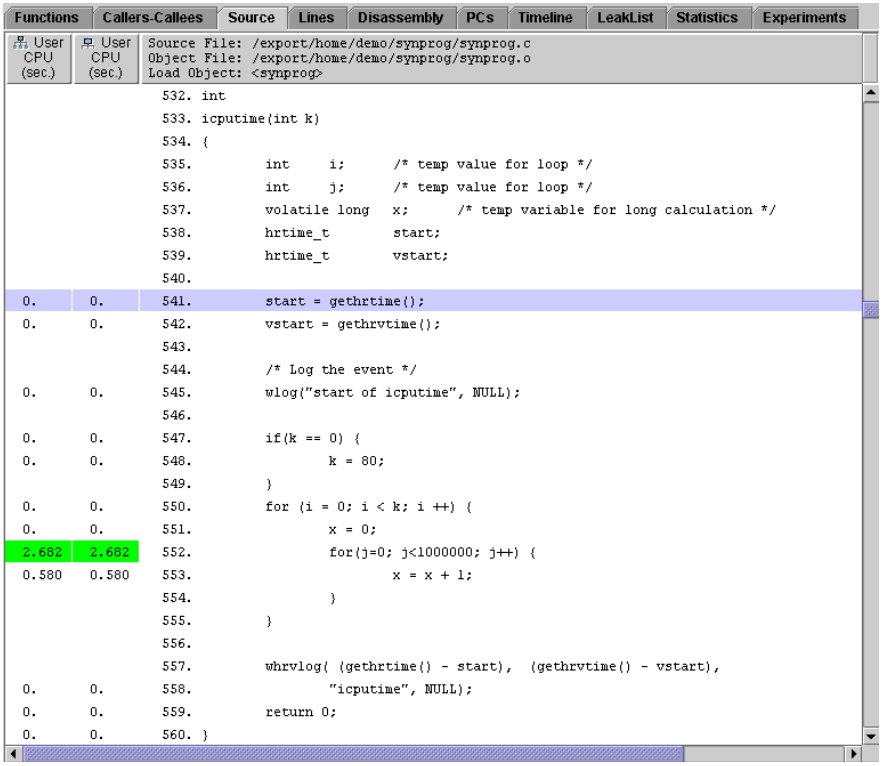


FIGURE 2-2 Source Tab Showing Annotated Source Code for Function `icputime`

The annotated source listing tells you which lines of code are responsible for the CPU time. Most of the time in both functions is used by the loop line and the line in which `x` is incremented.

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

5. **In both windows, click the Disassembly tab and locate the instructions for the line of source code in which x is incremented.**

   You can find these instructions by choosing High Metric Value in the Find tool combo box and searching.

   The time given for an instruction is the time spent waiting for the instruction to be issued, not the time spent executing the instruction.

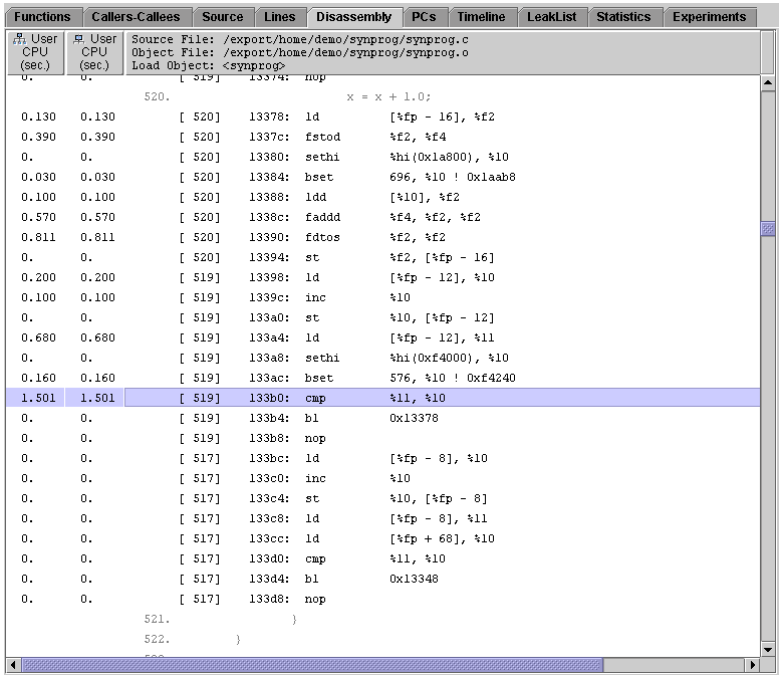


FIGURE 2-3 Disassembly Tab Showing Instructions for the Line in Which `x` Is Incremented in Function `cputime`

In `cputime()`, there are six instructions that must be executed to add 1 to x. A significant amount of time is spent loading 1.0, which is a double floating-point constant, and adding it to `x`. The `fdtos` and `fstod` instructions convert the value of `x` from a single floating-point value to a double floating-point value and back again, so that 1.0 can be added with the `faddd` instruction.

In `icputime()`, there are only three instructions: a load, an increment, and a store. These instructions take approximately a third of the time of the corresponding set of instructions in `cputime()`, because no conversions are necessary. The value 1 does not need to be loaded into a register—it can be added directly to `x` by a single instruction.

6. **When you have finished the exercise, close the new Analyzer window.**

Extension Exercise for Simple Metric Analysis

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

Metric Attribution and the gprof Fallacy

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

1. **In the Functions tab, select `gpf_work()` and then click Callers-Callees.**

   The Callers-Callees tab is divided into three panes. In the center pane is the selected function. In the pane above are the callers of the selected function, and in the pane below are the functions that are called by the selected function, which are termed callees. This tab is described in “The Callers-Callees Tab” on page 145 and also in “Basic Features of the Performance Analyzer” on page 34 of this chapter.

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

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

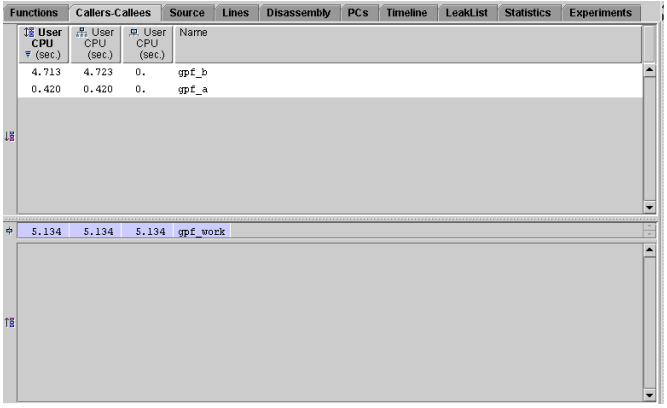


FIGURE 2-4 Callers-Callees Tab With `gpf_work` as the Selected Function

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

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

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

3. **Click the Source tab and scroll down so that you can see the code for both `gpf_a()` and `gpf_b()`.**

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

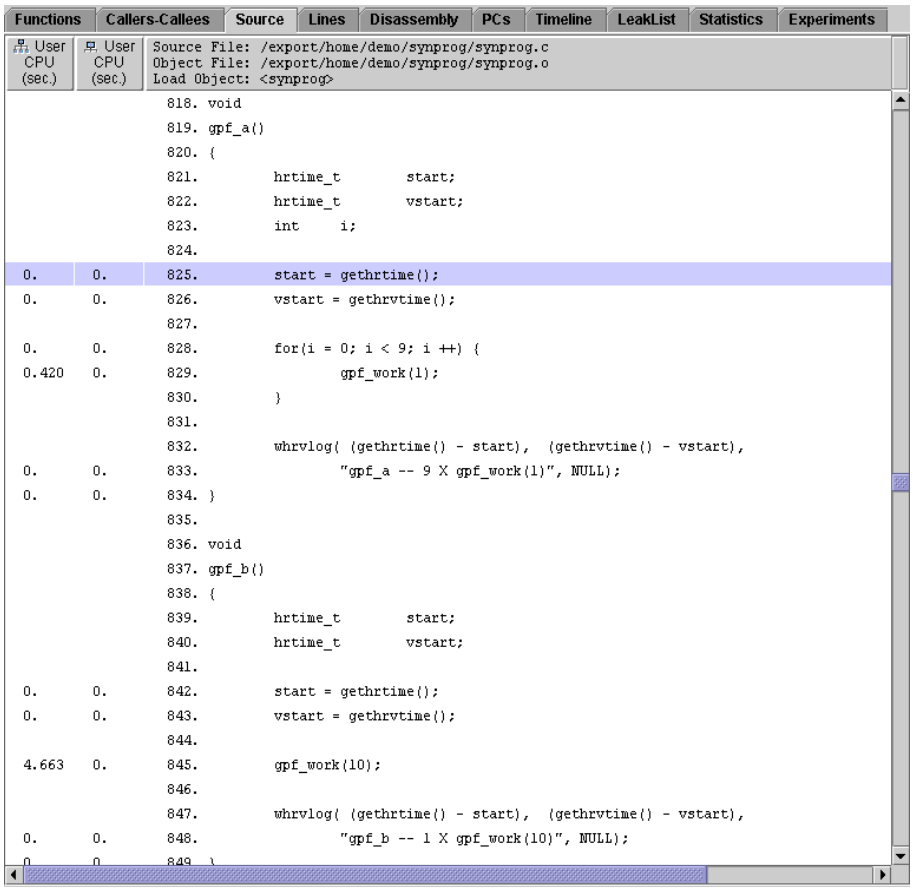


FIGURE 2-5 Source Tab Showing Annotated Source Code for Functions `gpf_a` and `gpf_b`

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

4. **Scroll down to display the code for** `gpf_work()`.

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

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

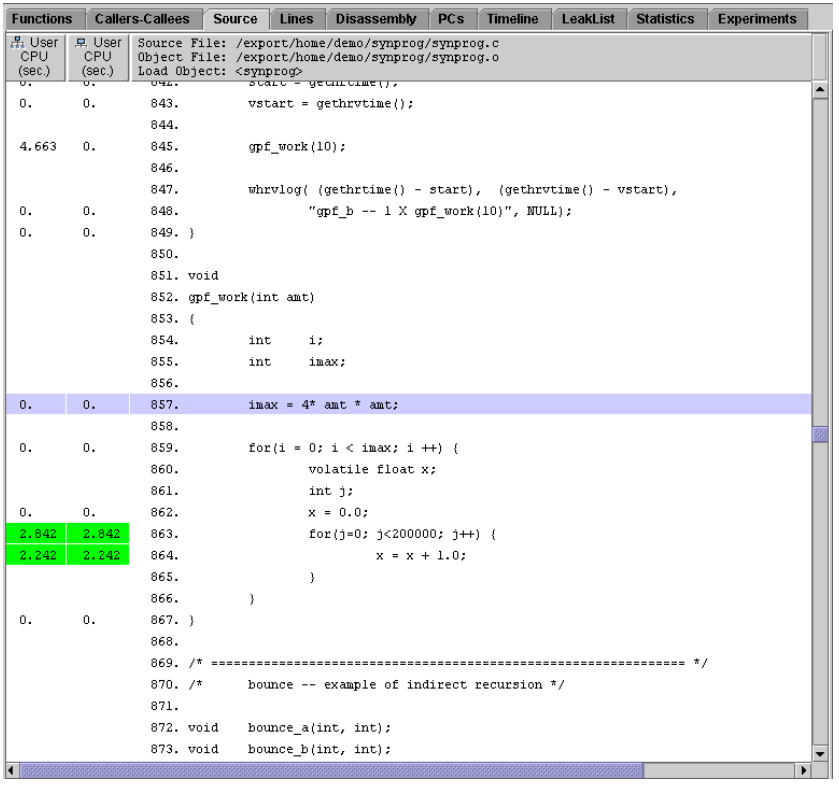


FIGURE 2-6 Source Tab Showing Annotated Source Code for Function `gpf_work`

The Effects of Recursion

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

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

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

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

1. **In the Functions tab, find function `recurse()` and select it.**

   Instead of scrolling the function list you can use the Find tool.

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

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

---

2. **Click the Callers-Callees tab.**

   The selected function, `recurse()`, is shown in the center pane. The function `real_recurse()`, which is called by `recurse()`, is shown in the lower pane. This pane is termed the Callees pane.

3. **Click `real_recurse()`.**

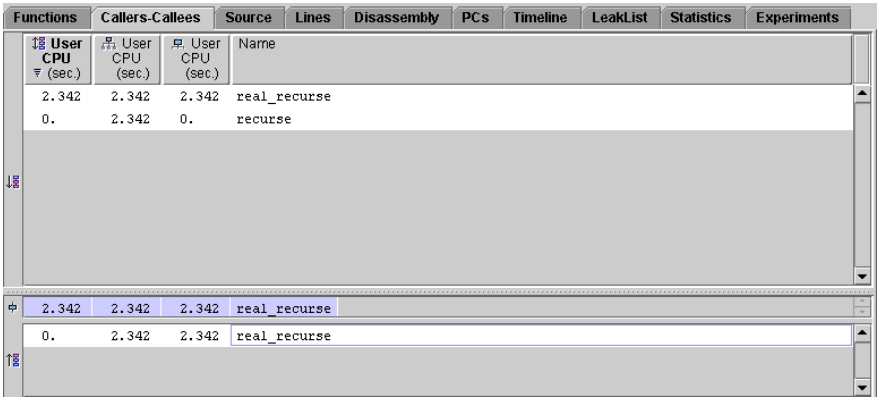


FIGURE 2-7 Callers-Callees Tab With `real_recurse` as the Selected Function

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

- Both `recurse()` and `real_recurse()` appear in the Callers pane (the upper pane) as callers of `real_recurse()`, because after `recurse()` calls `real_recurse()`, `real_recurse()` calls itself recursively.
- `real_recurse()` appears in the Callees pane because it calls itself.
- Exclusive metrics as well as inclusive metrics are displayed for `real_recurse()`, where the actual user CPU time is spent. The exclusive metrics are passed up to `recurse()` as inclusive metrics.
- The callee attributed metrics are affected by the recursive nature of the call. In a non-recursive call sequence, a single callee attributes all of its inclusive metrics to the caller. Here, `real_recurse()` is a leaf function as well as a caller and a callee of itself. To avoid double-counting of attributed metrics, the callee instance shows no attributed time. The appearance of `real_recurse()` as a callee gives information about the call sequence, but not about the attributed time.
- Likewise, the caller attributed metrics are affected by the recursive nature of the call. None of the inclusive time spent in `real_recurse()` is attributed to the caller `recurse()`, which is the ultimate caller of `real_recurse()`. Instead, the time is attributed to the caller of the instance of `real_recurse()` where the exclusive time is spent. To examine attributed metrics for `recurse()`, you should make it the selected function.

Now examine what happens in the indirect recursive sequence.

1. **Find function** `bounce()` **in the Functions tab and select it.**

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

2. **Click the Callers-Callees tab.**

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

3. **Click** bounce_a().

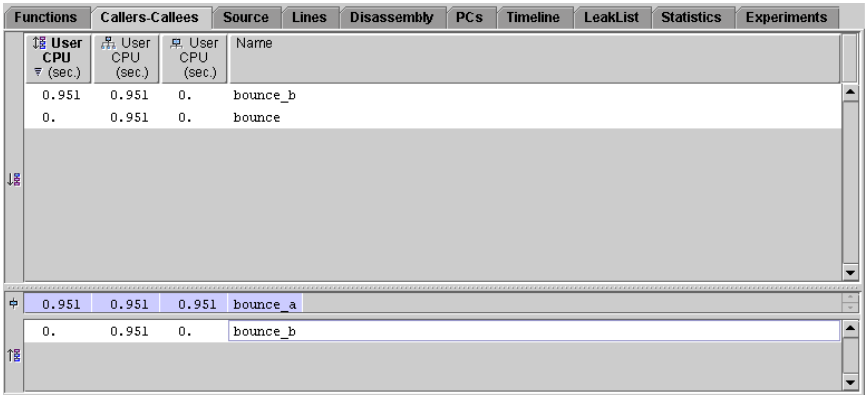


FIGURE 2-8 Callers-Callees Tab With bounce_a as the Selected Function

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

- Both bounce() and bounce_b() appear in the Callers pane as callers of bounce_a().
- In addition, bounce_b() appears in the Callees pane because it is called by bounce_a().
- Exclusive as well as inclusive metrics are displayed for bounce_a(), where the actual user CPU time is spent. These are passed up to the functions that call bounce_a() as inclusive metrics.
- All of the inclusive time in bounce_b() as a caller is attributed to calls to bounce_a(), but none of the inclusive time in bounce() is attributed to calls to bounce_a(). The inclusive time in bounce() and bounce_b() are not separate but are in fact the same time represented twice. To avoid double counting of attributed time, only the direct caller of the function where the time is spent shows the attributed time. This situation is the same as for recurse() and real_recurse().
- Because bounce_a() is the leaf function, bounce_b() as a callee does not attribute any time to bounce_a(), to avoid double-counting of attributed time.

4. **Click** bounce_b().

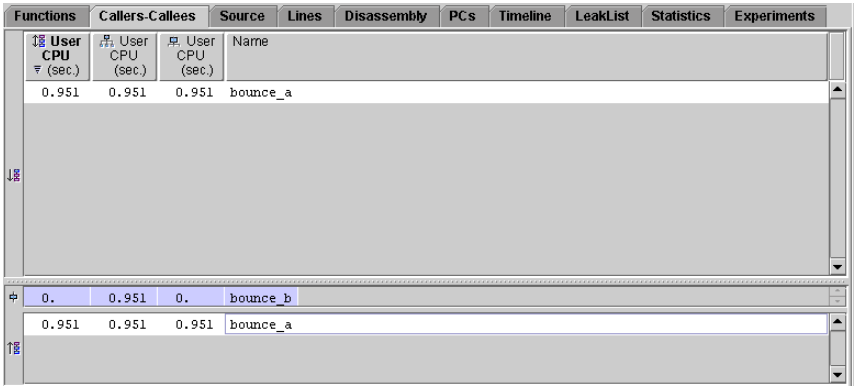


FIGURE 2-9 Callers-Callees Tab With `bounce_b` as the Selected Function

The Callers-Callees tab now displays information for `bounce_b()`. Function `bounce_a()` appears in both the Callers pane and the Callees pane. The callee attributed time is shown correctly, because `bounce_b()` is not a leaf function, and accumulates inclusive time from its call to the instance of `bounce_a()` in which the work is done.

Loading Dynamically Linked Shared Objects

This section demonstrates how the Performance Analyzer displays information for shared objects and how it handles calls to functions that are part of a dynamically linked shared object that can be loaded at different places at different times.

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

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

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

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

1. **Click the Functions tab.**
2. **Choose View → Show/Hide Functions.**

   The Show / Hide Functions dialog box lists all the load objects used by the program when it ran.

3. **Click Clear All, select** `so_syx.so` **and** `so_syn.so`**, then click Apply.**

   The functions for all the load objects except the two selected objects no longer appear in the function list. Their entries are replaced by a single entry for the entire load object.

   The list of load objects in the Functions tab includes only the load objects for which metrics were recorded, so it can be shorter than the list in the Show / Hide Functions dialog box.

4. **In the Functions tab, examine the metrics for** `sx_burncpu()` **and** `so_burncpu()`**.**

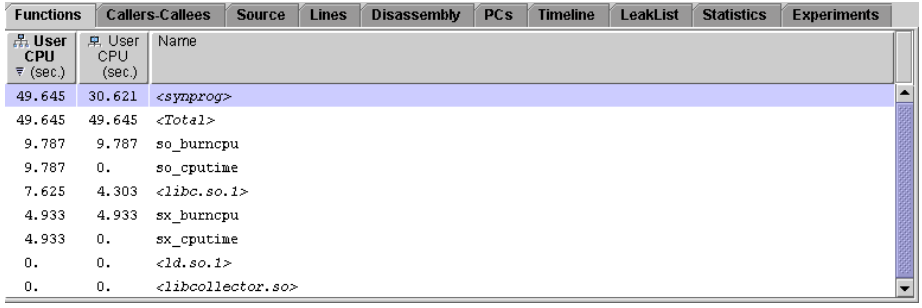


FIGURE 2-10 Functions Tab Showing Functions `so_burncpu` and `sx_burncpu`

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

Descendant Processes

This part of the example illustrates different ways of creating descendant processes and how they are handled, and demonstrates the Timeline display to get an overview of the execution of a program that creates descendant processes. The program forks two descendant processes. The parent process does some work, then calls `popen`, then does some more work. The first descendant does some work and then calls `exec`. The second descendant calls `system`, then calls `fork`. The descendant from this call to `fork` immediately calls `exec`. After doing some work, the descendant calls `exec` again and does some more work.

1. **Collect another experiment and restart the Performance Analyzer.**

```
% cd work-directory/synprog
% collect -F on synprog icpu.popen.cpu so.sx.exec system.forkexec
% analyzer test.2.er &
```

This command loads the founder and all of its descendant experiments, but data is only loaded for the founder experiment. To load data for the descendant experiments, choose the View->Filter Data, then select “Enable All”, then click “OK” or “Apply”. Note that you could also open the experiment `test.2.er` in the existing analyzer and then add the descendant experiments. If you do this you must open the Add Experiment dialog box once for each descendant experiment and type `test.2.er/`*descendant-name* in the text box, then click OK. You cannot navigate to the descendant experiments to select them: you must type in the name. The list of descendant names is: `_f1.er`, `_f1_x1.er`, `_f2.er`, `_f2_f1.er`, `_f2_f1_x1.er`, `_f2_f1_x1_x1.er`. You must add the experiments in this order, otherwise the remaining instructions in this part of the example do not match the experiments you see in the Performance Analyzer.

2. **Click the Timeline tab.**

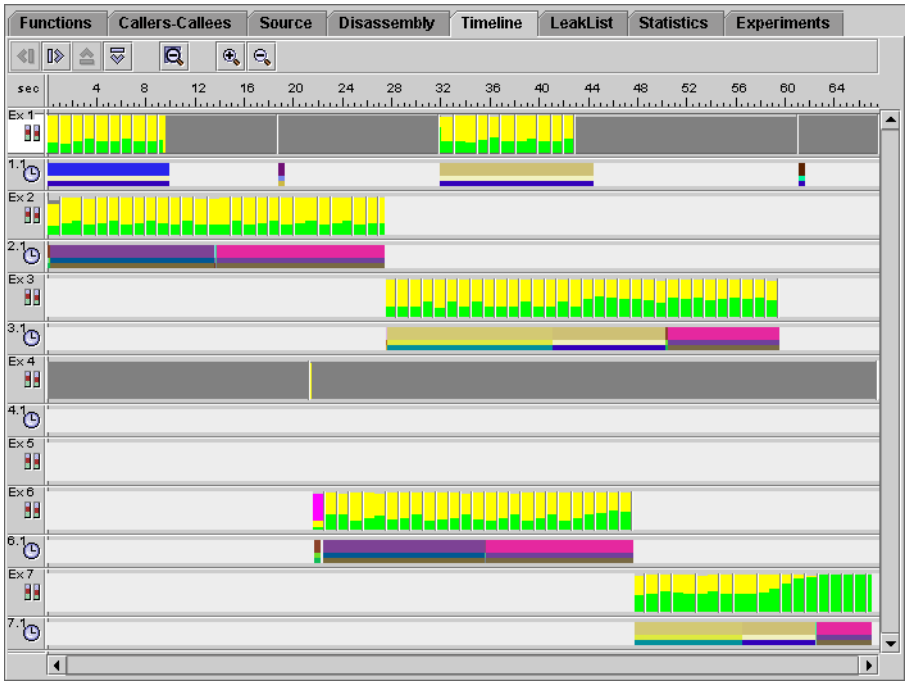


FIGURE 2-11 Timeline Tab Showing the Seven Experiments Recorded for the Parent Process and its Descendant Processes

The topmost bar for each experiment is the samples bar. The next bar contains the clock-based profiling event data.

Some of the samples are colored yellow and green. The green color indicates that the process is running in User CPU mode. The fraction of time spent in User CPU mode is given by the proportion of the sample that is colored green. Because there are three processes running most of the time, only about one-third of each sample is colored green. The rest is colored yellow, which indicates that the process is waiting for the CPU. This kind of display is normal when there are more processes running than there are CPUs to run on. When the parent process (experiment 1) has finished executing and is waiting for its children to finish, the samples for the running processes are half green and half yellow, showing that there are only two processes contending for the CPU. When the process that generates experiment 3 has completed, the remaining process (experiment 7) is able to use the CPU exclusively, and the samples in experiment 7 show all green after that time.

3. **Click the sample bar for experiment 7 in the region that shows half yellow and half green samples.**

4. **Zoom in so that you can see the individual event markers.**

   You can zoom in by dragging through the region you want to zoom in to, or clicking the zoom in button, or choosing Timeline → Zoom In x2, or typing Alt-T, I.

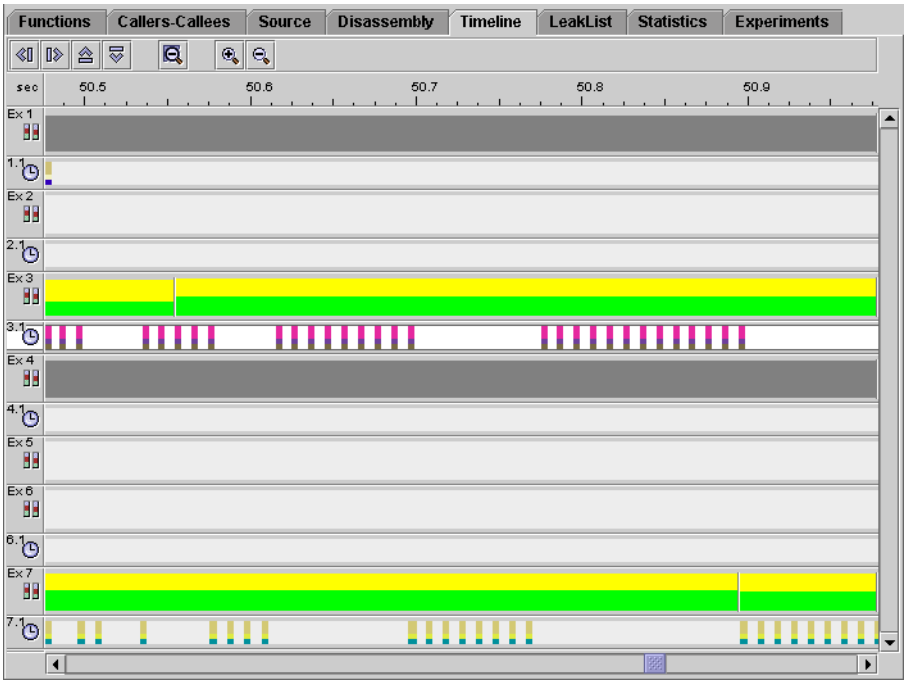


FIGURE 2-12 Timeline Tab at High Zoom Showing Event Markers and Gaps Between Them

There are gaps between the event markers in both experiment 3 and experiment 7, but the gaps in one experiment occur where there are event markers in the other experiment. These gaps show where one process is waiting for the CPU while the other process is executing.

5. **Reset the display to full width.**

   You can reset the display by clicking the Reset Display button, or choosing Timeline → Reset Display, or typing Alt-T, R.

   Some experiments do not extend for the entire length of the run. This situation is indicated by a light gray color for the regions of time where these experiments do not have any data (see FIGURE 2-11). Experiments 3, 5, 6, and 7 are created after their parent processes have done some work. Experiments 2, 5, and 6 are terminated by a successful call to exec. Experiment 3 ends before experiment 7 and its process terminates normally. The points at which `exec` is called show clearly: the data for experiment 3 starts where the data for experiment 2 ends, and the data for experiment 7 starts where the data for experiment 6 ends.

6. **Click the Experiments tab, then click the turner for `test.2.er`.**

   The experiments that are terminated by a successful call to `exec` show up as “bad experiments” in the Experiments tab. The experiment icon has a cross in a red circle superimposed on it.

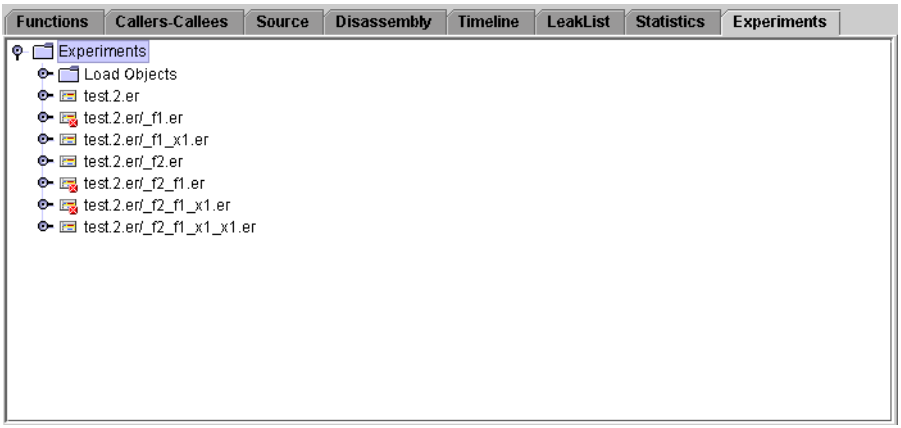


FIGURE 2-13 Experiments Tab Showing Seven Experiments, Three of Which Are Marked as "Bad"

7. Click the turner for `test.2.er/_f1.er`.

At the bottom of the text pane is a warning that the experiment terminated abnormally. Whenever a process successfully calls `exec`, the process image is replaced and the collector library is unloaded. The normal completion of the experiment cannot take place, and is done instead when the experiment is loaded into the Analyzer.

8. Click the Timeline tab.

The dark gray regions in the samples bars indicate time spent waiting, other than waiting for the CPU or for a user lock. The first dark gray region in experiment 1 (the experiment for the founding process) occurs during the call to `popen`. Most of the time is spent waiting, but there are some events recorded during this time. In this region, the process created by `popen` is using CPU time and competing with the other processes, but it is not recorded in an experiment. Similarly, the first dark gray region in experiment 4 occurs during a call to `system`. In this case the calling process waits until the call is complete, and does no work until that time. The process created by the call to `system` is also competing with the other processes for the CPU, and does not record an experiment.

The last gray region in experiment 1 occurs when the process is waiting for its descendants to complete. The process that records experiment 4 calls `fork` after the call to `system` is complete, and then waits until all its descendant processes have completed. This wait time is indicated by the last gray region. In both these cases, the waiting processes do no work and have no descendants that are not recording experiments.

Experiment 4 spends most of its time waiting. As a consequence, it records no profiling data until the very end of the experiment.

Experiment 5 appears to have no data at all. It is created by a call to `fork` that is immediately followed by a call to `exec`.

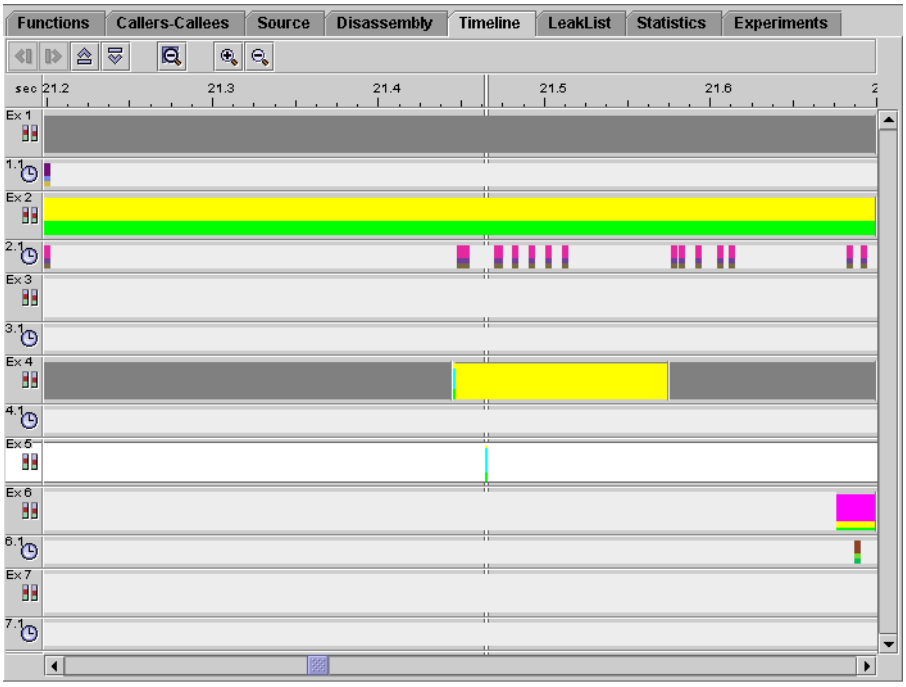


FIGURE 2-14 Timeline Tab at High Zoom, Showing Short Sample for Experiment 5

9. **Zoom in on the boundary between the two gray regions in experiment 4.**

   At sufficiently high zoom, you can see that there is a very small sample in experiment 5.

10. **Click the sample in experiment 5 and look at the Event tab.**

    The experiment recorded an initial sample point and a sample point in the call to `exec`, but did not last long enough to record any profiling data. This is the reason why there is no profiling data bar for experiment 5.

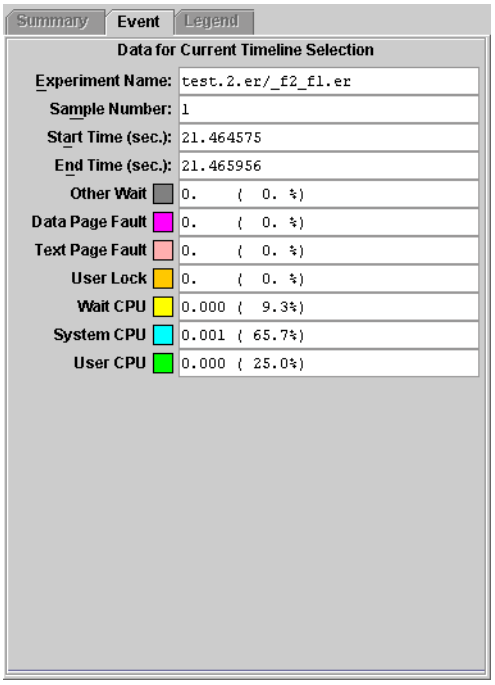


FIGURE 2-15 Event Tab Showing Very Short Duration Sample

Extension Exercise for Descendant Processes

If you have access to a computer with more than one processor, repeat the data collection. What differences do you see in the Timeline display?

Example 2: Analyzing the Performance of a Mixed Java / C++ Application

This example shows how to collect and analyze performance data for an application written in the Java™ programming language that calls C++ methods. It illustrates the timeline of the program execution and how performance data for the program is presented in the Performance Analyzer GUI. The process for collecting / analyzing data as described in this chapter can be used for mixed-language applications and programs written entirely in the Java programming language.

jsynprog Program Structure and Control Flow

The jsynprog experiment is a simple application designed to demonstrate the amount of time it takes to perform various tests, such as vector/array manipulation, recursion, integer/double addition, memory allocation, system calls, and calls to native code using JNI. The experiment consists of the following individual source files:

- `jsynprog.java:` The main entry point, containing the required `public static void main(String[] args)` method.
- `Intface.java:` A simple interface that defines two methods for adding numbers together: `public int add_int(),` and `public double add_double().`
- `Routine.java:` Implements the methods of `Intface` and defines additional methods that test garbage collection, inner classes, recursion, indirect recursion, array operations, vector operations, and system calls.
- `Sub_Routine.java:` Subclass of `Routine` which overrides the `add_int()` method.
- `jsynprog.h:` Header file used by `cloop.cc`
- `cloop.cc:` C++ code for native methods called by jsynprog.

Control flow through the experiment begins and ends with `jsynprog.main`. While in `main`, the program performs the following individual tests, in the following order, by calling methods of other classes such as `Routine.memalloc` and `Sub_Routine.add_double.`

- Test 1: `Routine.memalloc`: Triggers garbage collection in the JVM by creating large memory allocations.
- Test 2: `Routine.add_int`: Uses nested loops to repeatedly add a set of integers.
- Test 3: `Routine.add_double`: Uses nested loops to repeatedly add a set of doubles.
- Test 4: `Sub_Routine.add_int`: Overrides the add_int test to provide different behavior.
- Test 5: `Routine.has_inner_class`: Defines and uses an inner classes, local to the method.
- Test 6: `Routine.recurse`: Demonstrates the use of direct recursion; this method calls itself many times.
- Test 7: `Routine.bounce`: Demonstrates the use of indirect recursion; this method calls another method which in turn calls this method.
- Test 8: `Routine.array_op`: Allocates two large arrays, then performs a copy operation on them.
- Test 9: `Routine.vector_op`: Allocates a large Vector then performs operations to add/remove elements from it.

- Test 10: `Routine.sys_op`: Spends some time in system calls, using `java.lang.System.currentTimeMillis`()
- Test 11: `jsynprog.jni_JavaJavaC`: Demonstrates the use of JNI: A Java method calls another Java method which calls a C function.
- Test 12: `jsynprog.JavaCC`: Demonstrates the use of JNI: A Java method calls a C function which calls another C function.
- Test 13: `jsynprog.JavaCJava`: Demonstrates the use of JNI: A Java method calls a C function which calls a Java method.

Collecting Data for `jsynprog`

Read the instructions in the sections, “Setting Up the Examples for Execution” on page 32 and “Basic Features of the Performance Analyzer” on page 34, if you have not done so. Compile `jsynprog` before you begin this example.

To collect data for `jsynprog` and start the Performance Analyzer, type the following at the command line:

```
% cd work-directory/jsynprog
% make collect
% analyzer test.1.er &
```

Loading the performance data takes a few seconds, and no information will appear in the GUI until all data has finished loading. A progress bar in the upper right corner of the screen will keep you informed of the data currently being loaded.

You are now ready to analyze the `jsynprog` experiment using the procedures presented in the following section.

Analyzing `jsynprog` Program Data

1. **Viewing function data**

   The Functions tab contains a list of all functions for which performance data was recorded, together with metrics that are derived from the performance data. The performance data is aggregated to obtain metrics for each function. The term “functions” includes both Java and C++ methods.

In the default display, the first column of metrics is Exclusive User CPU time: time spent inside a function. The second column of metrics is Inclusive User CPU time: time spent inside a function and in any function that it calls. The list is sorted by the data in the first column.

When all experiment data has finished loading, the functions tab will be selected, showing the most costly routines (ranked in terms of user CPU time), and many of the various tests are shown to take several seconds. At the top of the function list is an artificial function, `<Total>`. This artificial function represents the entire program. In the Java representation, the artificial function `<no Java callstack recorded>` indicates that the Java virtual machine did not report a Java callstack, even though a Java program was running. (The JVM will do so when necessary to avoid deadlocks, or when unwinding the Java stack will cause excessive synchronization).

Click on the column header for Inclusive User CPU time, and select the top function: `jsynprog.main`.

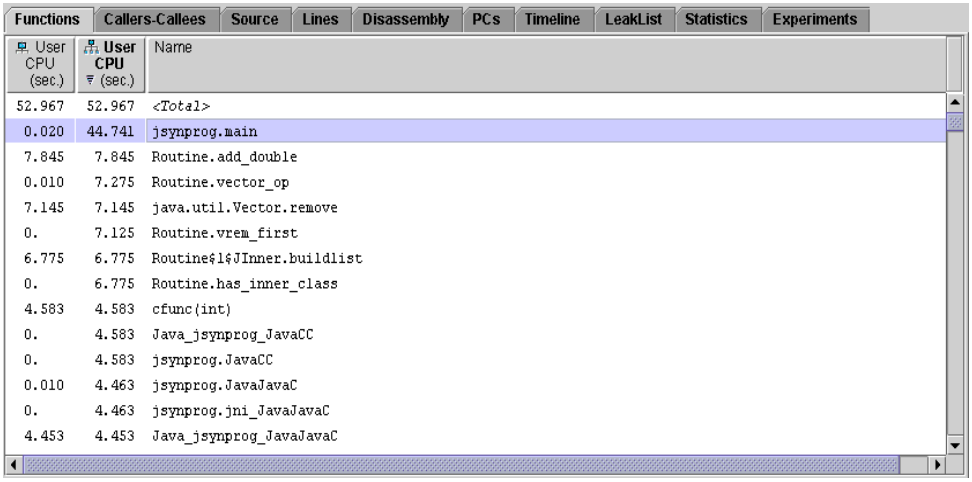


FIGURE 2-16 The Functions Tab, Showing Several `jsynprog` Experiment Methods

The full set of clock profiling metrics and selected object attributes are summarized on the right panel.

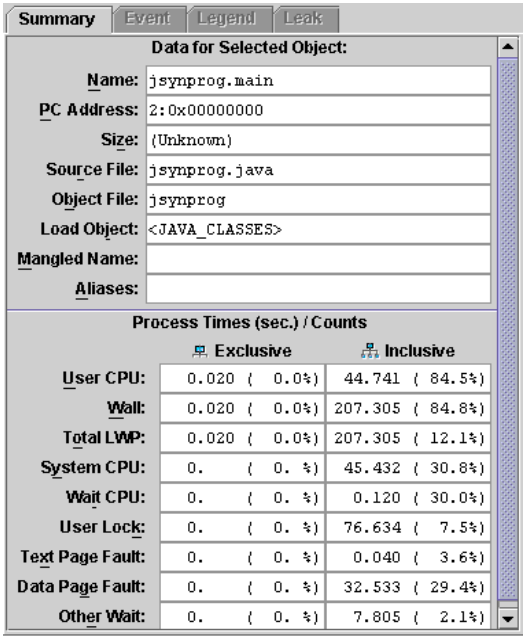


FIGURE 2-17 The Summary Panel

Additional metrics and metric presentations can be added to the main display using the View / Set Data Presentation / Metrics dialog.

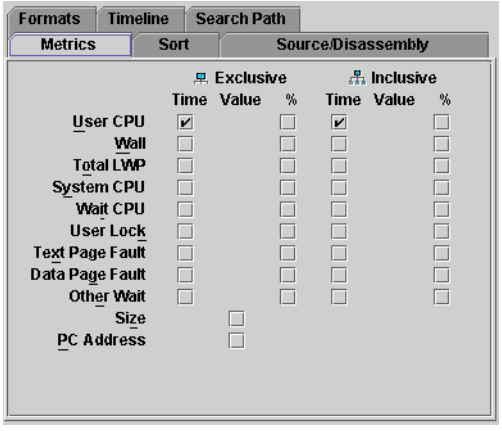


FIGURE 2-18 The Metrics Tab of the Set Data Presentation Dialog

2. Viewing callers-callees data

With the `jsynprog.main` function selected, click on the callers-callees tab. Note how the test functions described in the overview appear in the list of callees.

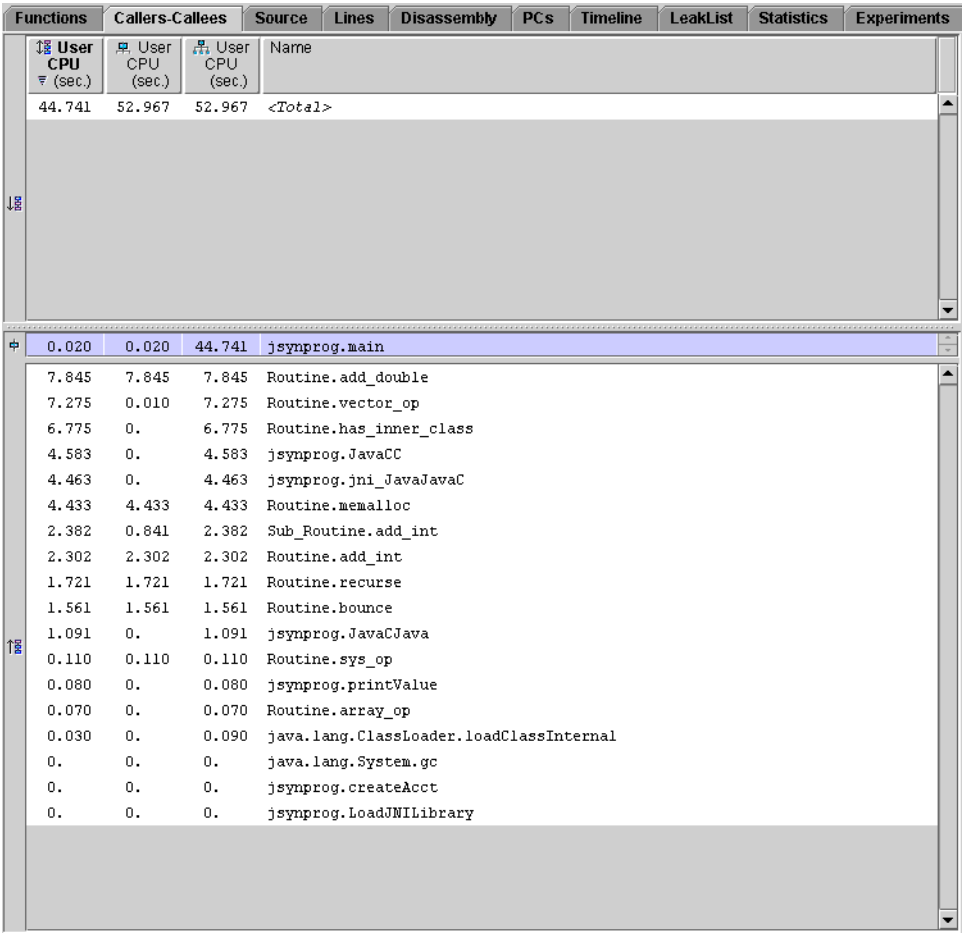


FIGURE 2-19 The Callers-Callees Tab, With `jsynprog.main` Selected

The first three (most expensive) functions listed are `Routine.add_double`, `Routine.vector_op`, and `Routine.has_inner_class`. The callers-callees display features a dynamic call tree that you can navigate through by selecting a function from the top (caller) or bottom (callee) panels. For example, select `Routine.has_inner_class` from the list of callees, and notice how the display redraws centered on it, listing an inner class as its most important callee. To return to the caller of this function, simply select `jsynprog.main` from the top panel. You can use this procedure to navigate through the entire experiment. Take a minute to navigate through the various `Routine` functions, ending back at `jsynprog.main`.

3. Viewing source data

With `jsynprog.main` still selected, click on the source tab. The source code for jsynprog.java appears, with performance numbers for the various lines where jsynprog.main invokes some other method.

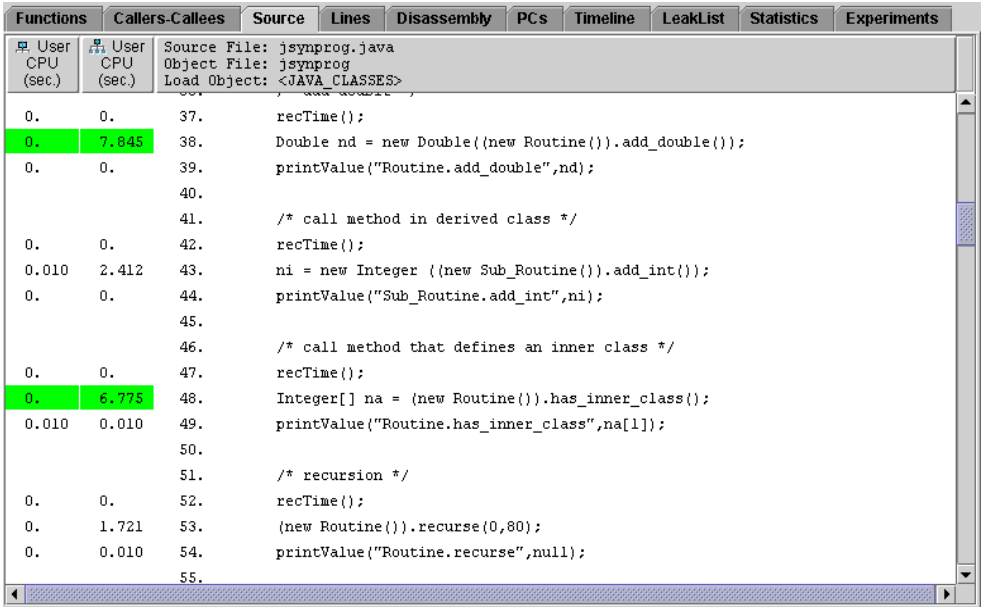


FIGURE 2-20 Source Tab, Listing `jsynprog.java`

This particular screen shot shows two expensive method calls highlighted in green: add_double (7.845 seconds) and has_inner_class (6.775 seconds). Scrolling up and down the source code listing will reveal other expensive calls, also highlighted in green. You may also notice some lines that appear in red italicized text, such as *Routine.add_int <instructions without line numbers>*. This represents HotSpot compiled versions of the named function, since HotSpot does not provide bytecode indices for the PCs from these instructions.

If the analyzer was unable to locate your source files, try setting the path explicitly using the View / Set Data Presentation / Search Path dialog. The path to add is the root directory of your sources.

4. Viewing disassembly data

Click on the disassembly tab to view the annotated bytecode with Java source interleaved.

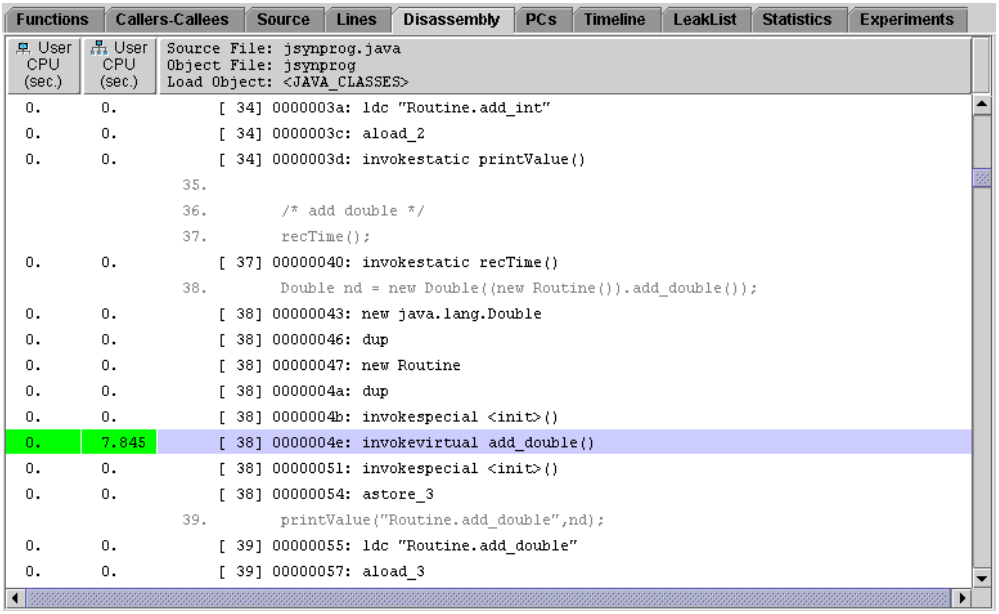


FIGURE 2-21 The Disassembly Tab, Showing Annotated Bytecode

As before, this view shows both Exclusive and Inclusive User CPU times, with the most expensive calls highlighted in green. In this view the Java bytecode appears in black, while its corresponding Java source appears in light gray. You may also notice that, similar to the Source tab, the Disassembly tabs provides lines in red italicized text such as *Routine.add_int <HotSpot-compiled leaf instructions>*. As before, this represents the HotSpot compiled version of the named function.

5. Viewing timeline data

The Performance Analyzer displays a timeline of the events that it records in a graphical format. The progress of the program's execution and the calls made by the program can be tracked using this display.

To display the timeline, click the Timeline tab.

The data is displayed in horizontal bars. The colored rectangles in each bar represent the recorded events. The colored area appears to be continuous when the events are closely spaced, but at high zoom the individual events are resolved.

For each experiment, the global data is displayed in the topmost bar. This bar is labeled with the experiment number (Ex 1) and an icon. The colored rectangles in the global data bar are called *samples*. The global data bar is also called the samples bar. Samples represent timing data for the process as a whole. The timing data includes times for all the LWPs, whether they are displayed in the timeline or not.

The event data is displayed below the global data. The display contains one event bar for each LWP (lightweight process) for each data type. The colored rectangles in the event bars are called *event markers*. Each marker represents a part of the call stack for the event. Each function in the call stack is color coded. The color coding for the functions is displayed in the Legend tab in the right pane.

In the Java representation, the timeline for this experiment appears as follows:

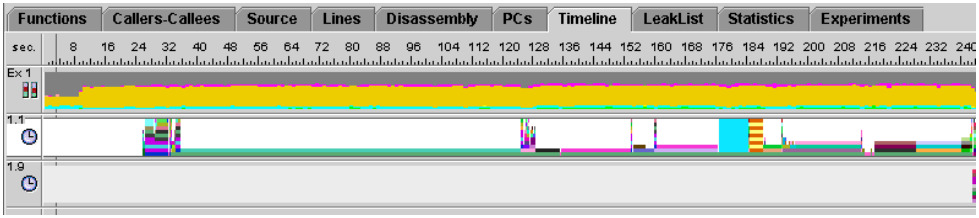


FIGURE 2-22 The Timeline Tab in the Java Representation

The bar labeled 1.1 is the user thread. Click on any part of it to find out which function was executing at that particular point in time. The data will be displayed in the Event tab on the right side of the screen. You can toggle between the Summary, Event, and Legend tabs for information about what was happening at any given point in time. You can use the left / right arrows located at the top of the analyzer screen to step back or forward to the previous / next recorded event. You can also use the up / down arrows to step though the various threads.

Switching to the Expert-Java representation reveals the following additional threads.

FIGURE 2-23 The Timeline Tab in the Expert-Java Representation

In this representation, there are three threads of interest: the user thread, the Garbage Collector (GC) thread, and the HotSpot compiler thread. In the above display, these threads are numbered 1.1, 1.5, and 1.10, respectively. If you compare the user thread from this Expert-Java representation against the previous Java-representation display, you'll notice some extra activity in the first 30 seconds of the run. Clicking anywhere in this area reveals this callstack to be JVM activity. Next, notice the burst of activity in the GC thread between 30 and 120 seconds. The `Routine.memalloc` test repeatedly allocates large amounts of memory, which causes the garbage collector to check periodically for memory that can be reclaimed. Finally notice the shorts bursts of activity that repeatedly appear in the HotSpot compiler thread. This indicates that HotSpot has dynamically compiled the code shortly after the start of each task.

6. Examine the Interpreted vs.Compiled Methods

While still in the Expert-Java representation, return once again to the functions tab. Alphabetically sort the list by selecting "Name" at the top of the screen, then scroll down to the list of `Routine` methods. You may notice that some methods have duplicate entries, as shown in the following figure.

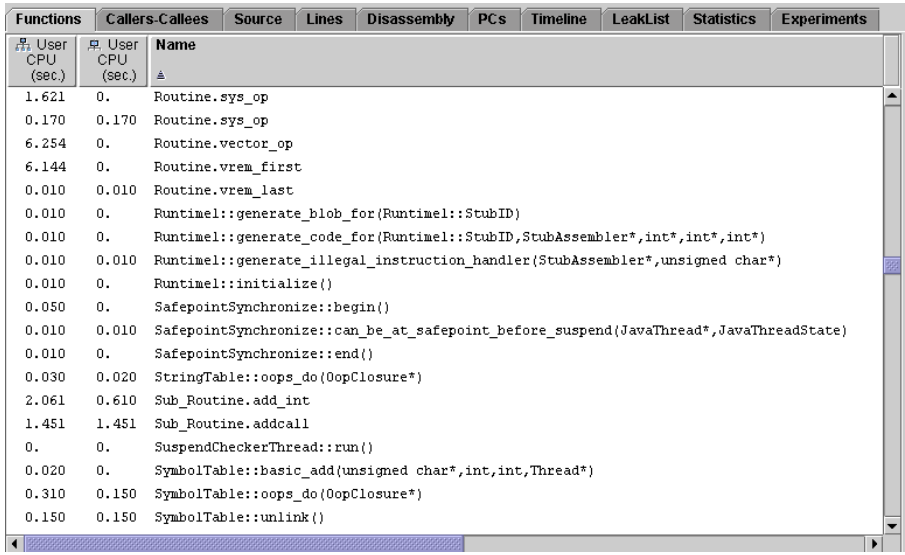


FIGURE 2-24 The Functions Tab, Showing Interpreted and Dynamically-compiled Versions of `Routine.sys_op`

In this example, the method `Routine.sys_op` appears twice: one is the interpreted version, and the other is the version that was dynamically compiled by the HotSpot virtual machine. To determine which is which, select one of the methods and examine the Load Object field of the Summary tab on the right side of the screen. In this example, selecting the first occurrence of `Routine.sys_op` reveals that it comes from the load object `<JAVA_COMPILED_METHODS>`, indicating that this is the dynamically-compiled version. Selecting the second occurrence reveals the load object to be `<JAVA_CLASSES>`, indicating that this is the interpreted version.

The HotSpot virtual machine does not dynamically compile methods that only execute for short periods of time. Therefore, methods listed only once will always be the interpreted versions.

Example 3: OpenMP Parallelization Strategies

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

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

See the *Fortran Programming Guide* for background on parallelization strategies and OpenMP directives. When the compiler identifies an OpenMP directive, it generates special functions and calls to the threads library. These functions appear in the Performance Analyzer display. For more information, see “Parallel Execution and Compiler-Generated Body Functions” on page 210 and “Compiler-Generated Body Functions” on page 219. Messages from the compiler about the actions it has taken appear in the annotated source and disassembly listings.

Collecting Data for `omptest`

Read the instructions in the sections, “Setting Up the Examples for Execution” on page 32 and “Basic Features of the Performance Analyzer” on page 34, if you have not done so. Compile `omptest` before you begin this example.

In this example you generate two experiments: one that is run with 4 CPUs and one that is run with 2 CPUs. The experiments are labeled with the number of CPUs.

To collect data for `omptest`, type the following commands in the C shell.

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

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

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

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

```
$ cd ~/work-directory/omptest
$ make collect
```

To start the Performance Analyzer for both experiments, type:

```
$ analyzer omptest.4.er &
$ analyzer omptest.2.er &
```

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

Comparing Parallel Sections and Parallel Do Strategies

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

To compare the four-CPU run with the two-CPU run, you must have two Analyzer windows, with `omptest.4.er` loaded into one, and `omptest.2.er` loaded into the other.

1. **In the Functions tab of each Performance Analyzer window, find and select `psec_`.**

   You can use the Find tool to find this function. Note that there are other functions that start with psec_ which have been generated by the compiler.

2. **Position the windows so that you can compare the Summary tabs.**

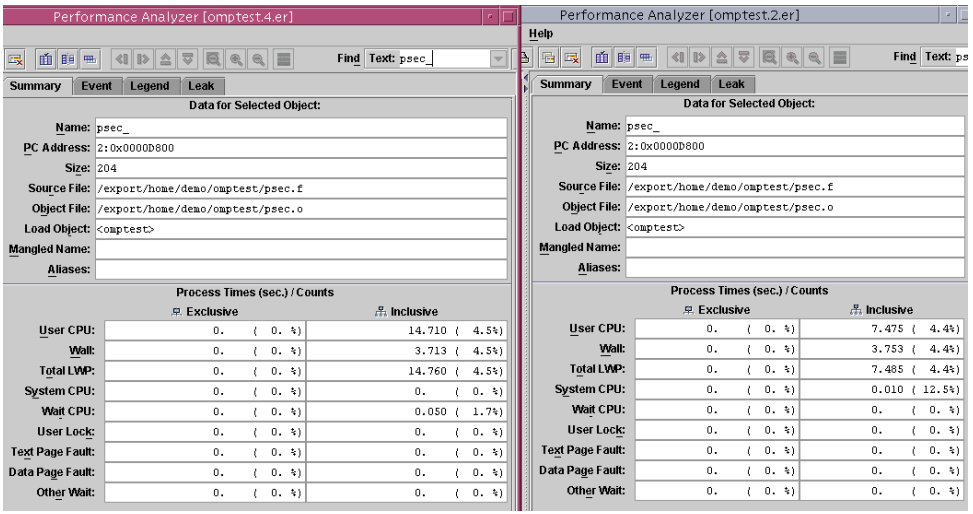


FIGURE 2-25 Summary Tabs for Function `psec_` From the Four-CPU Run (Left) and the Two-CPU Run (Right)

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

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

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

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

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

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

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

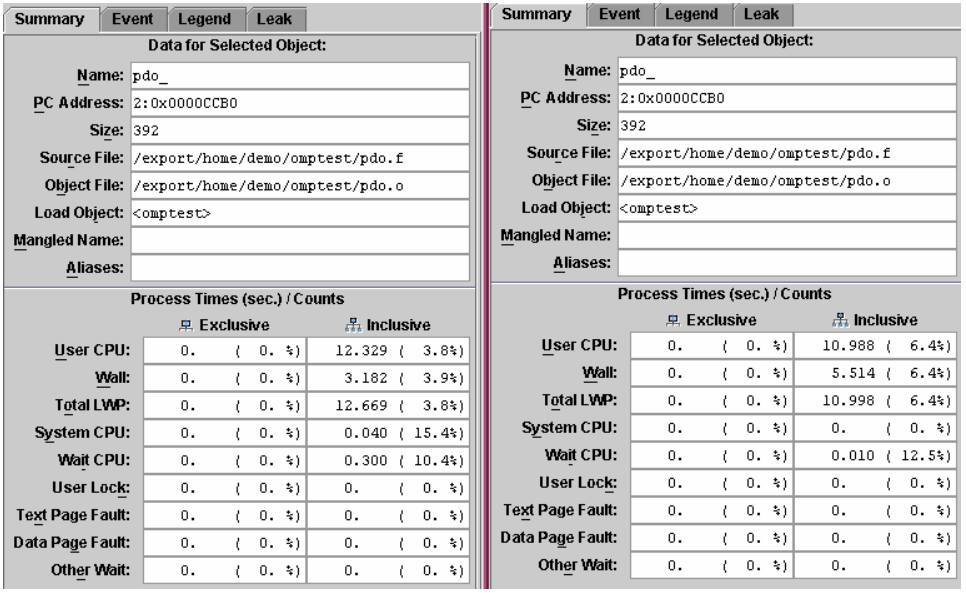


FIGURE 2-26 Summary Tabs for Function `pdo_` From the Four-CPU Run (Left) and the Two-CPU Run (Right)

6. **Close the Analyzer window that is displaying** `omptest.2.er`**.**

Comparing Critical Section and Reduction Strategies

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

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

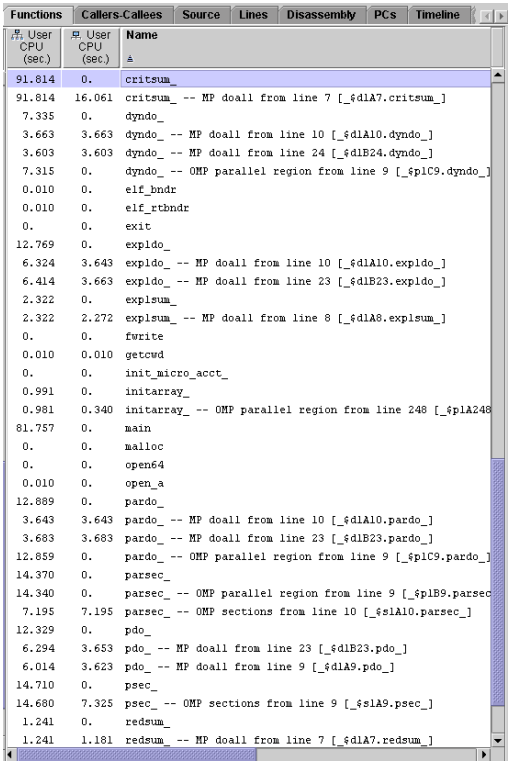


FIGURE 2-27 Functions Tab Showing Entries for `critsum_` and `redsum_`

1. **For the four-CPU experiment, `omptest.4.er`, locate `critsum_` and `redsum_` in the Functions tab.**

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

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

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

Example 4: Locking Strategies in Multithreaded Programs

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

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

Collecting Data for `mttest`

Read the instructions in the sections, “Setting Up the Examples for Execution” on page 32 and “Basic Features of the Performance Analyzer” on page 34, if you have not done so. Compile `mttest` before you begin this example.

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

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

```
% cd work-directory/mttest
% collect -s on -o mttest.4.er mttest
% collect -s on -o mttest.1.er mttest -u
% analyzer mttest.4.er &
% analyzer mttest.1.er &
```

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

```
% cd work-directory/mttest
% make collect
% analyzer mttest.4.er &
% analyzer mttest.1.er &
```

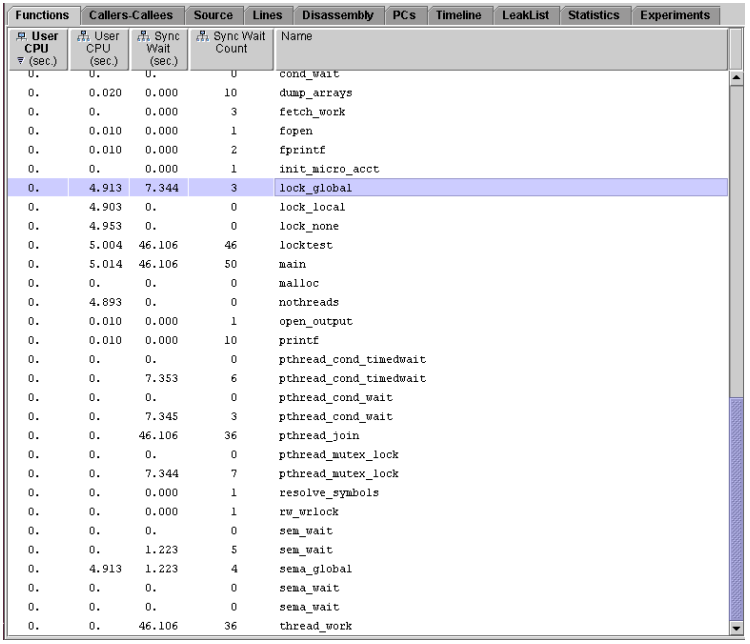
After you have loaded the two experiments, position the two Performance Analyzer windows so that you can see them both.

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

How Locking Strategies Affect Wait Time

1. **Find `lock_local` and `lock_global` in the Functions tab for the four-CPU experiment, `mttest.4.er`.**

   Both functions have approximately the same inclusive user CPU time, so they are doing the same amount of work. However, `lock_global()` has a high synchronization wait time, whereas `lock_local()` has none.



User CPU (sec.)	User CPU (sec.)	Sync Wait (sec.)	Sync Wait Count	Name
0.	0.	0.	0	cond_wait
0.	0.020	0.000	10	dump_arrays
0.	0.	0.000	3	fetch_work
0.	0.010	0.000	1	fopen
0.	0.010	0.000	2	fprintf
0.	0.	0.000	1	init_micro_acct
0.	4.913	7.344	3	lock_global
0.	4.903	0.	0	lock_local
0.	4.953	0.	0	lock_none
0.	5.004	46.106	46	locktest
0.	5.014	46.106	50	main
0.	0.	0.	0	malloc
0.	4.893	0.	0	nothreads
0.	0.010	0.000	1	open_output
0.	0.010	0.000	10	printf
0.	0.	0.	0	pthread_cond_timedwait
0.	0.	7.353	6	pthread_cond_timedwait
0.	0.	0.	0	pthread_cond_wait
0.	0.	7.345	3	pthread_cond_wait
0.	0.	46.106	36	pthread_join
0.	0.	0.	0	pthread_mutex_lock
0.	0.	7.344	7	pthread_mutex_lock
0.	0.	0.000	1	resolve_symbols
0.	0.	0.000	1	rw_wrlock
0.	0.	0.	0	sem_wait
0.	0.	1.223	5	sem_wait
0.	4.913	1.223	4	sema_global
0.	0.	0.	0	sema_wait
0.	0.	0.	0	sema_wait
0.	0.	46.106	36	thread_work

FIGURE 2-28 Functions Tab for the Four-CPU Experiment Showing Data for `lock_local` and `lock_global`

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

2. **Click `lock_global`, then click the Source tab.**

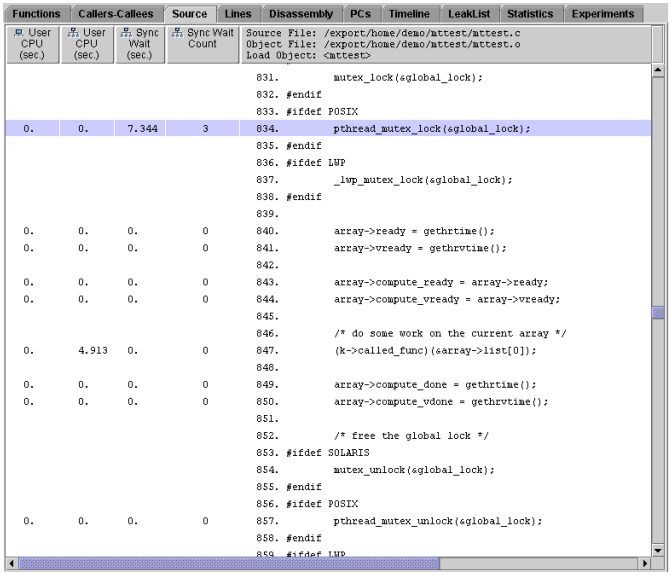


FIGURE 2-29 Source Tab for the Four-CPU Experiment for Function `lock_global`

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

3. **Click `lock_local` in the Functions tab, then click the Source tab.**

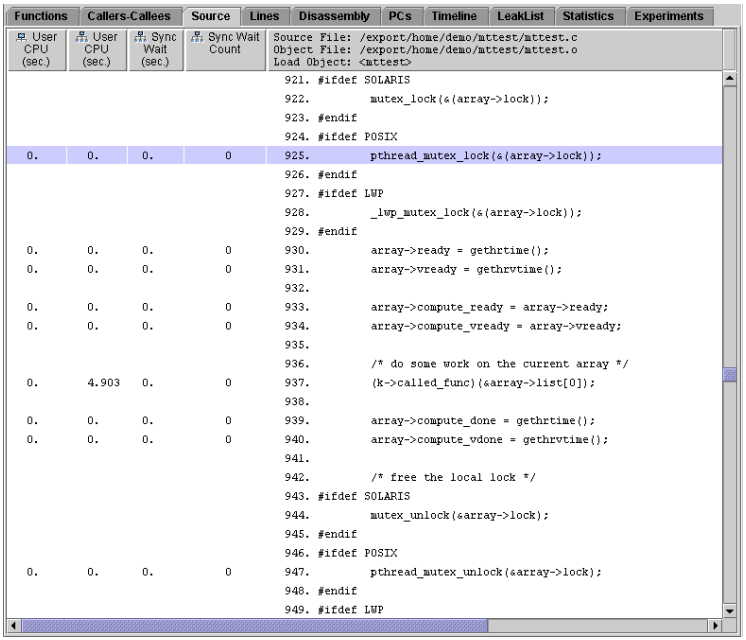


FIGURE 2-30 Source Tab for the Four-CPU Experiment for Function `lock_local`

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

4. **Change the metric selection for the one-CPU experiment,** `mttest.1.er`:

   a. **Choose View → Set Data Presentation.**

   b. **Clear Exclusive User CPU Time and Inclusive Synchronization Wait Counts.**

   c. **Select Inclusive Total LWP Time, Inclusive Wait CPU Time and Inclusive Other Wait Time.**

   d. **Click Apply.**

5. **In the Functions tab for the one-CPU experiment, find** `lock_local` **and** `lock_global`**.**

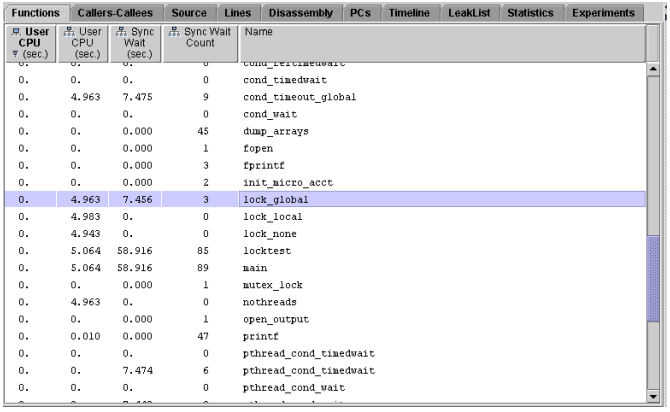


FIGURE 2-31 Functions Tab for the One-CPU Experiment Showing Data for `lock_local` and `lock_global`

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

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

6. **Restore the metric selection to the default values for `mttest.1.er`.**

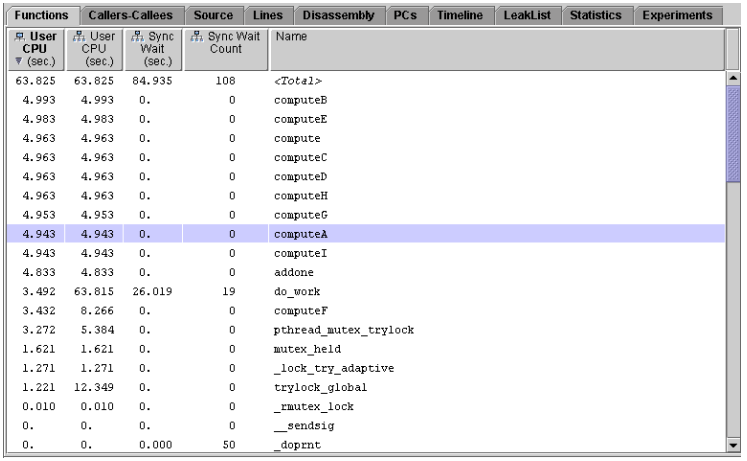
   In the Set Data Presentation dialog box, which should still be open, do the following:

   a. **Select Exclusive User CPU Time and Inclusive Synchronization Wait Counts.**

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

   c. **Click OK.**

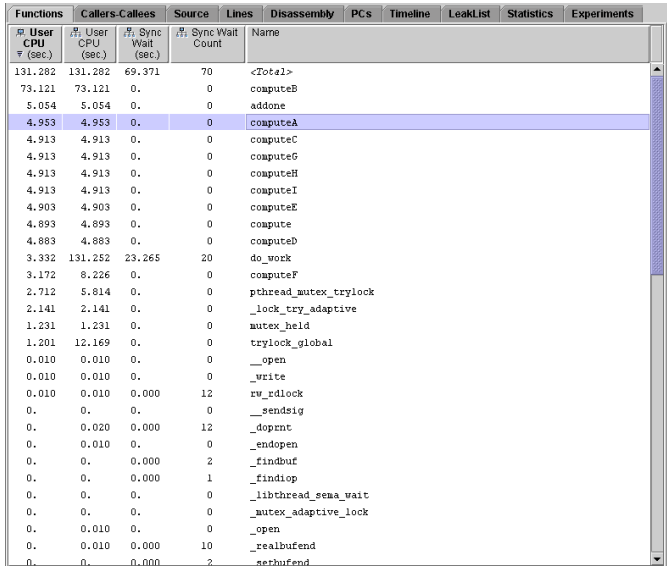
1. **Find `computeA` and `computeB` in the Functions tab of both Performance Analyzer window.**



User CPU (sec.)	User CPU (sec.)	Sync Wait (sec.)	Sync Wait Count	Name
63.825	63.825	84.935	108	<Total>
4.993	4.993	0.	0	computeB
4.983	4.983	0.	0	computeE
4.963	4.963	0.	0	compute
4.963	4.963	0.	0	computeC
4.963	4.963	0.	0	computeD
4.963	4.963	0.	0	computeH
4.953	4.953	0.	0	computeG
4.943	4.943	0.	0	computeA
4.943	4.943	0.	0	computeI
4.833	4.833	0.	0	addone
3.492	63.815	26.019	19	do_work
3.432	8.266	0.	0	computeF
3.272	5.384	0.	0	pthread_mutex_trylock
1.621	1.621	0.	0	mutex_held
1.271	1.271	0.	0	_lock_try_adaptive
1.221	12.349	0.	0	trylock_global
0.010	0.010	0.	0	_rmutex_lock
0.	0.	0.	0	__sendsig
0.	0.	0.000	50	_doprnt

FIGURE 2-32 Functions Tab for the One-CPU Experiment Showing Data for Functions `computeA` and `computeB`

In the one-CPU experiment, `mttest.1.er`, the inclusive user CPU time for `computeA()` is almost the same as for `computeB()`.



Functions | Callers-Callees | Source | Lines | Disassembly | PCs | Timeline | LeakList | Statistics | Experiments

User CPU (sec.)	User CPU (sec.)	Sync Wait (sec.)	Sync Wait Count	Name
131.282	131.282	69.371	70	<Total>
73.121	73.121	0.	0	computeB
5.054	5.054	0.	0	addone
4.953	4.953	0.	0	computeA
4.913	4.913	0.	0	computeC
4.913	4.913	0.	0	computeG
4.913	4.913	0.	0	computeH
4.913	4.913	0.	0	computeI
4.903	4.903	0.	0	computeE
4.893	4.893	0.	0	compute
4.883	4.883	0.	0	computeD
3.332	131.252	23.265	20	do_work
3.172	8.226	0.	0	computeF
2.712	5.814	0.	0	pthread_mutex_trylock
2.141	2.141	0.	0	_lock_try_adaptive
1.231	1.231	0.	0	mutex_held
1.201	12.169	0.	0	trylock_global
0.010	0.010	0.	0	__open
0.010	0.010	0.	0	_write
0.010	0.010	0.000	12	rw_rdlock
0.	0.	0.	0	__sendsig
0.	0.020	0.000	12	_doprnt
0.	0.010	0.	0	_endopen
0.	0.	0.000	2	_findbuf
0.	0.	0.000	1	_findiop
0.	0.	0.	0	_libthread_sema_wait
0.	0.	0.	0	_mutex_adaptive_lock
0.	0.010	0.	0	_open
0.	0.010	0.000	10	_realbufend
0.	0.	0.000	2	_serbufend

FIGURE 2-33 Functions Tab for the Four-CPU Experiment Showing Data for Functions `computeA` and `computeB`

In the four-CPU experiment, `mttest.4.er`, `computeB()` uses much more inclusive user CPU time than `computeA()`.

The remaining instructions apply to the four-CPU experiment, `mttest.4.er`.

2. **Click `computeA`, then click the Source tab. Scroll down so that the source for both `computeA()` and `computeB()` is displayed.**

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

3. **Use the Find tool to find `cache_trash`. Repeat the search until the source code for `cache_trash()` is displayed.**

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

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

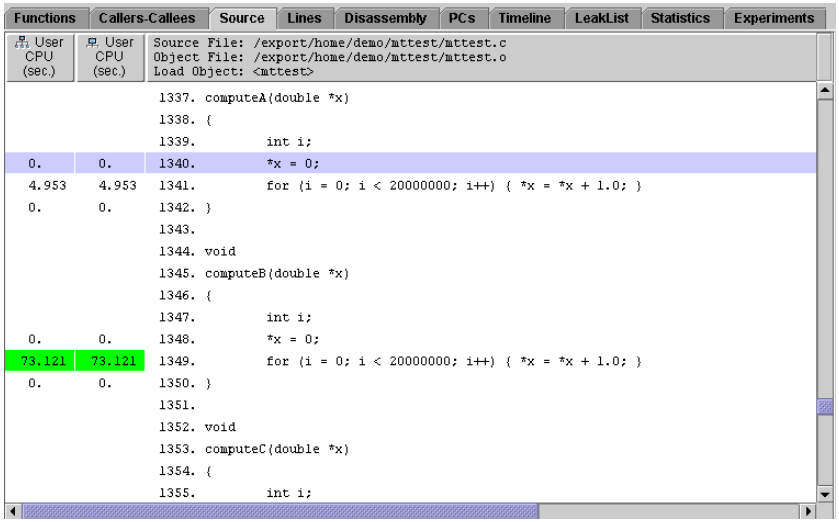


FIGURE 2-34 Source Tab for the Four-CPU Experiment Showing Annotated Source Code for `computeA` and `computeB`

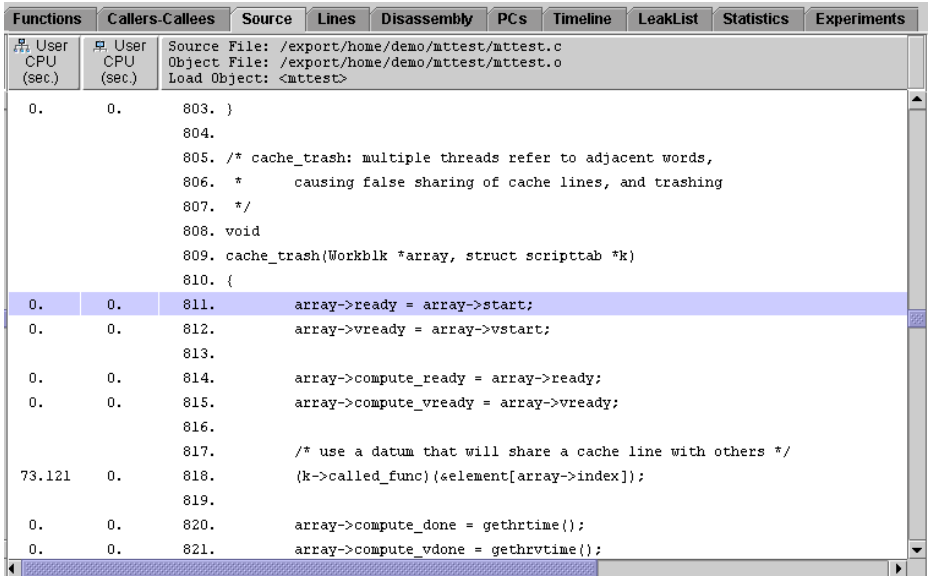


FIGURE 2-35 Source tab for the Four-CPU Experiment Showing Annotated Source Code for `cache_trash`

4. **Compare the calls to `computeA()` and `computeB()`.**

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

   `computeB()`, however, is called with doubles in each thread that occupy successive words in memory (`&element[array->index]`). These words share a cache line. Whenever a thread writes to one of these addresses in memory, any other threads that have that address in their cache must delete the data, which is now invalid. If one of the threads needs the data again later in the program, the data must be copied back into the data cache from memory, even if the data item that is needed has not changed. The resulting cache misses, which are attempts to access data not available in the data cache, waste a lot of CPU time. This explains why `computeB()` uses much more user CPU time than `computeA()` in the four-CPU experiment.

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

Extension Exercises for `mttest`

1. If you are using a computer that has hardware counters, run the four-CPU experiment again and collect data for one of the cache hardware counters, such as cache misses or stall cycles. On UltraSPARC® III hardware you can use the command

```
% collect -p off -h dcstall -o mttest.3.er mttest
```

   You can combine the information from this new experiment with the previous experiment by choosing File → Add. Examine the hardware counter data for `ComputeA` and `ComputeB` in the Functions tab and the Source tab.

2. The makefile contains optional settings for compilation variables that are commented out. Try changing some of these options and see what effect the changes have on program performance. The compilation variable to edit is `OFLAGS`.

Example 5: Cache Behavior and Optimization

This example addresses the issue of efficient data access and optimization. It uses two implementations of a matrix-vector multiplication routine, `dgemv`, which is included in standard BLAS libraries. Three copies of the two routines are included in the program. The first copy is compiled without optimization, to illustrate the effect of the order in which elements of an array are accessed on the performance of the routines. The second copy is compiled with `-O2`, and the third with `-fast`, to illustrate the effect of compiler loop reordering and optimization.

This example illustrates the use of hardware counters and compiler commentary for performance analysis. You must run this example on UltraSPARC III hardware.

Collecting Data for `cachetest`

Read the instructions in the sections, “Setting Up the Examples for Execution” on page 32 and “Basic Features of the Performance Analyzer” on page 34, if you have not done so. Compile `cachetest` before you begin this example.

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

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

```
% cd work-directory/cachetest
% collect -o flops.er -S off -p on -h fpadd,,fpmul cachetest
% collect -o cpi.er -S off -p on -h cycles,,insts cachetest
% collect -o dcstall.er -h dcstall cachetest
```

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

```
% cd work-directory/cachetest
% make collect
```

The Performance Analyzer shows exclusive metrics only. This is different from the default, and has been set in a local defaults file. See “Default-Setting Commands” on page 188 for more information.

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

Execution Speed

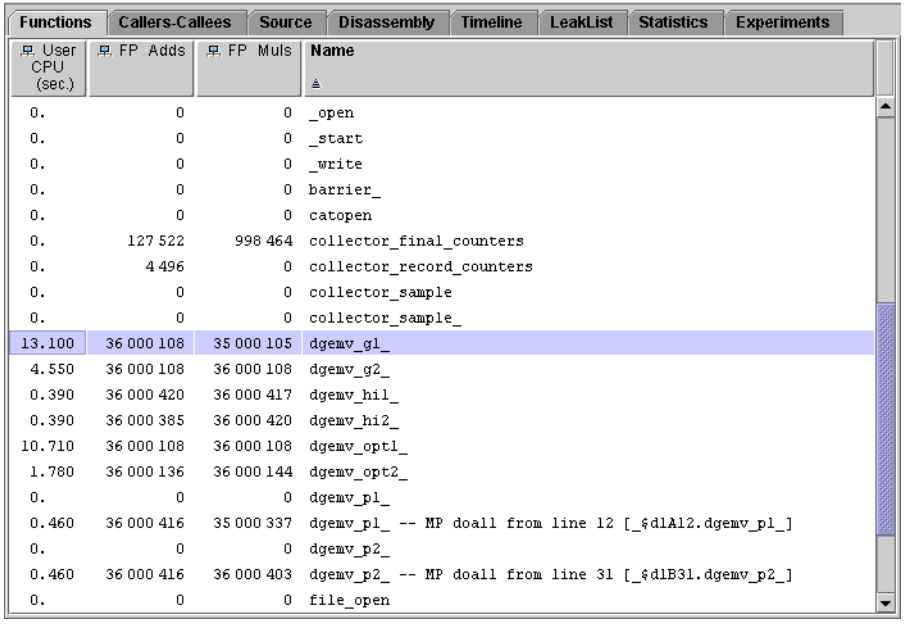
1. **Start the analyzer on the floating point operations experiment.**

```
% cd work-directory/cachetest
% analyzer flops.er &
```

2. **Click the header of the Name column.**

   The functions are sorted by name, and the display is centered on the selected function, which remains the same.

3. **For each of the six functions, `dgemv_g1`, `dgemv_g2`, `dgemv_opt1`, `dgemv_opt2`, `dgemv_hi1`, and `dgemv_hi2`, add the FP Adds and FP Muls counts and divide by the User CPU time and 10^6.**



User CPU (sec.)	FP Adds	FP Muls	Name
0.	0	0	_open
0.	0	0	_start
0.	0	0	_write
0.	0	0	barrier_
0.	0	0	catopen
0.	127 522	998 464	collector_final_counters
0.	4 496	0	collector_record_counters
0.	0	0	collector_sample
0.	0	0	collector_sample_
13.100	36 000 108	35 000 105	dgemv_g1_
4.550	36 000 108	36 000 108	dgemv_g2_
0.390	36 000 420	36 000 417	dgemv_hi1_
0.390	36 000 385	36 000 420	dgemv_hi2_
10.710	36 000 108	36 000 108	dgemv_opt1_
1.780	36 000 136	36 000 144	dgemv_opt2_
0.	0	0	dgemv_p1_
0.460	36 000 416	35 000 337	dgemv_p1_ -- MP doall from line 12 [_$d1A12.dgemv_p1_]
0.	0	0	dgemv_p2_
0.460	36 000 416	36 000 403	dgemv_p2_ -- MP doall from line 31 [_$d1B31.dgemv_p2_]
0.	0	0	file_open

FIGURE 2-36 Functions Tab Showing User CPU, FP Adds and FP Muls for the Six Variants of `dgemv`

The numbers obtained are the MFLOPS counts for each routine. All of the subroutines have the same number of floating-point instructions issued but use different amounts of CPU time. (The variation between the counts is due to counting statistics.) The performance of `dgemv_g2` is better than that of `dgemv_g1`, the performance of `dgemv_opt2` is better than that of `dgemv_opt1`, but the performance of `dgemv_hi2` and `dgemv_hi1` are about the same.

4. **Run `cachetest` without collecting performance data.**

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

   The values computed from the data are lower because of the overhead for the data collection. The MFLOPS values computed by the program under data collection vary because the data collection overhead is different for different data types and for different hardware counters.

Program Structure and Cache Behavior

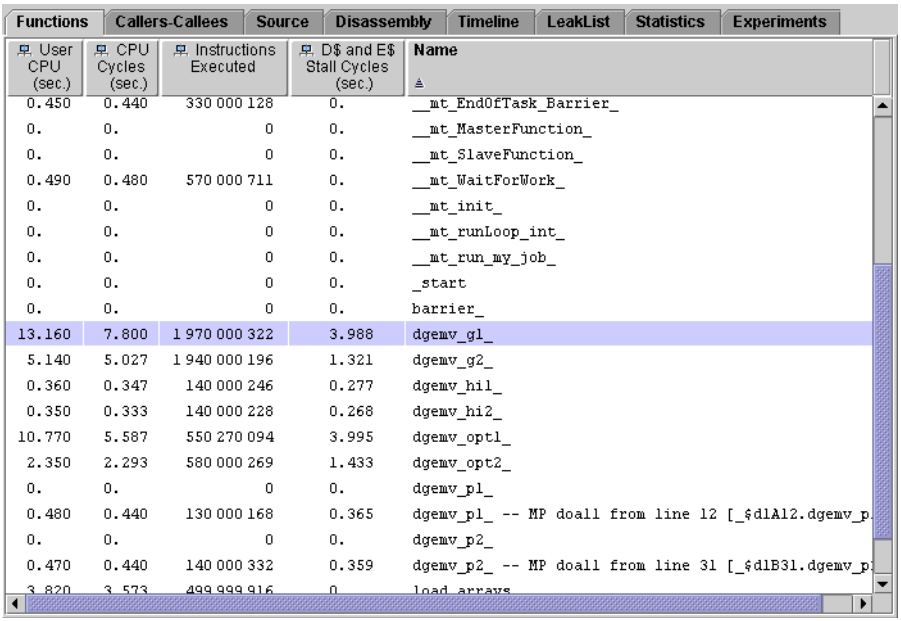
In this section, we examine the reasons why `dgemv_g2` has better performance than `dgemv_g1`. If you already have the Performance Analyzer running, do the following:

1. **Choose File → Open and open `cpi.er`.**

2. **Choose File → Add and add `dcstall.er`.**

   If you do not have the Performance Analyzer running, type the following commands at the prompt:

```
% cd work-directory/cachetest
% analyzer cpi.er dcstall.er &
```



User CPU (sec.)	CPU Cycles (sec.)	Instructions Executed	D$ and E$ Stall Cycles (sec.)	Name
0.450	0.440	330 000 128	0.	__mt_EndOfTask_Barrier_
0.	0.	0	0.	__mt_MasterFunction_
0.	0.	0	0.	__mt_SlaveFunction_
0.490	0.480	570 000 711	0.	__mt_WaitForWork_
0.	0.	0	0.	__mt_init_
0.	0.	0	0.	__mt_runLoop_int_
0.	0.	0	0.	__mt_run_my_job_
0.	0.	0	0.	_start
0.	0.	0	0.	barrier_
13.160	7.800	1 970 000 322	3.988	dgemv_g1_
5.140	5.027	1 940 000 196	1.321	dgemv_g2_
0.360	0.347	140 000 246	0.277	dgemv_hi1_
0.350	0.333	140 000 228	0.268	dgemv_hi2_
10.770	5.587	550 270 094	3.995	dgemv_opt1_
2.350	2.293	580 000 269	1.433	dgemv_opt2_
0.	0.	0	0.	dgemv_p1_
0.480	0.440	130 000 168	0.365	dgemv_p1_ -- MP doall from line 12 [_$d1A12.dgemv_p
0.	0.	0	0.	dgemv_p2_
0.470	0.440	140 000 332	0.359	dgemv_p2_ -- MP doall from line 31 [_$d1B31.dgemv_p
3.820	3.573	499 999 916	0.	load_arrays

FIGURE 2-37 Functions Tab Showing User CPU Time, CPU Cycles, Instructions Executed and D- and E-Cache Stall Cycles for the Six Variants of `dgemv`

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

   There is a difference between these two metrics for `dgemv_g1` because of DTLB (data translation lookaside buffer) misses. The system clock is still running while the CPU is waiting for a DTLB miss to be resolved, but the cycle counter is turned off. The difference for `dgemv_g2` is negligible, indicating that there are few DTLB misses.

2. **Compare the D- and E-cache stall times for `dgemv_g1` and `dgemv_g2`.**

   There is less time spent waiting for the cache to be reloaded in `dgemv_g2` than in `dgemv_g1`, because in `dgemv_g2` the way in which data access occurs makes more efficient use of the cache.

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

3. **Choose View → Set Data Presentation and deselect the metrics for Instructions Executed and CPU Cycles in the Metrics tab.**

4. **Click `dgemv_g1`, then click the Source tab.**

5. **Resize and scroll the display so that you can see the source code for both `dgemv_g1` and `dgemv_g2`.**

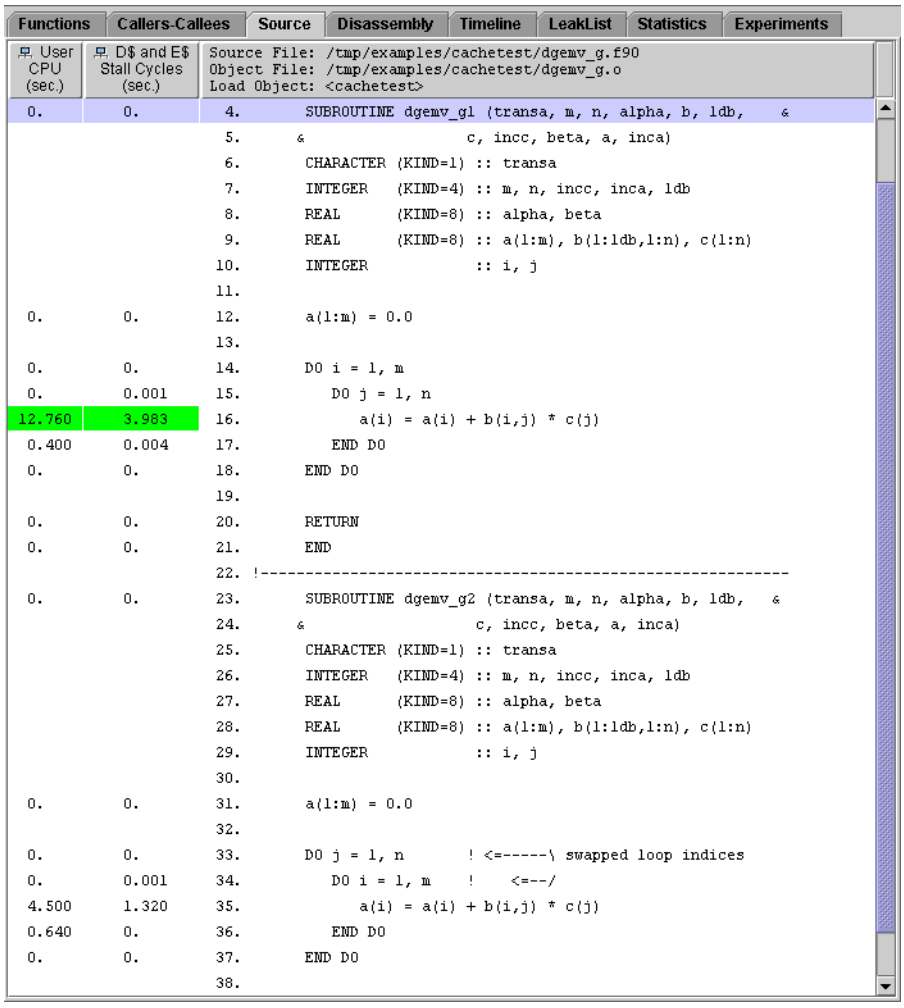


FIGURE 2-38 Source Tab Showing Annotated Source Code for `dgemv_g1` and `dgemv_g2`

The loop structure in the two routines is different. Because the code is not optimized, the data in the array in `dgemv_g1` is accessed by rows, with a large stride (in this case, 6000). This is the cause of the DTLB and cache misses. In `dgemv_g2`, the data is accessed by column, with a unit stride. Since the data for each loop iteration is contiguous, a large segment can be mapped and loaded into cache and there are cache misses only when this segment has been used and another is required.

Program Optimization and Performance

In this section we examine the effect of two different optimization options on the program performance, `-O2` and `-fast`. The transformations that have been made on the code are indicated by compiler commentary messages, which appear in the annotated source code.

1. **Load the experiments `cpi.er` and `dcstall.er` into the Performance Analyzer.**

   If you have just completed the previous section, Choose View → Set Data Presentation and ensure that the metric for CPU Cycles as a time and the metric for Instructions Executed are selected.

   If you do not have the Performance Analyzer running, type the following commands at the prompt:.

   ```
   % cd work-directory/cachetest
   % analyzer cpi.er dcstall.er &
   ```

2. **Click the header of the Name column.**

   The functions are sorted by name, and the display is centered on the selected function, which remains the same.

3. **Compare the metrics for `dgemv_opt1` and `dgemv_opt2` with the metrics for `dgemv_g1` and `dgemv_g2` (see** FIGURE 2-37**).**

   The source code is identical to that in `dgemv_g1` and `dgemv_g2`. The difference is that they have been compiled with the `-O2` compiler option. Both functions show about the same decrease in CPU time, whether measured by User CPU time or by CPU cycles, and about the same decrease in the number of instructions executed, but in neither routine is the cache behavior improved.

4. **In the Functions tab, compare the metrics for `dgemv_opt1` and `dgemv_opt2` with the metrics for `dgemv_hi1` and `dgemv_hi2`.**

   The source code is identical to that in `dgemv_opt1` and `dgemv_opt2`. The difference is that they have been compiled with the `-fast` compiler option. Now both routines have the same CPU time and the same cache performance. Both the CPU time and the cache stall cycle time have decreased compared to `dgemv_opt1` and `dgemv_opt2`. Waiting for the cache to be loaded takes about 80% of the execution time.

5. **Click `dgemv_hi1`, then click the Source tab. Resize and scroll the display so that you can see the source for all of `dgemv_hi1`.**

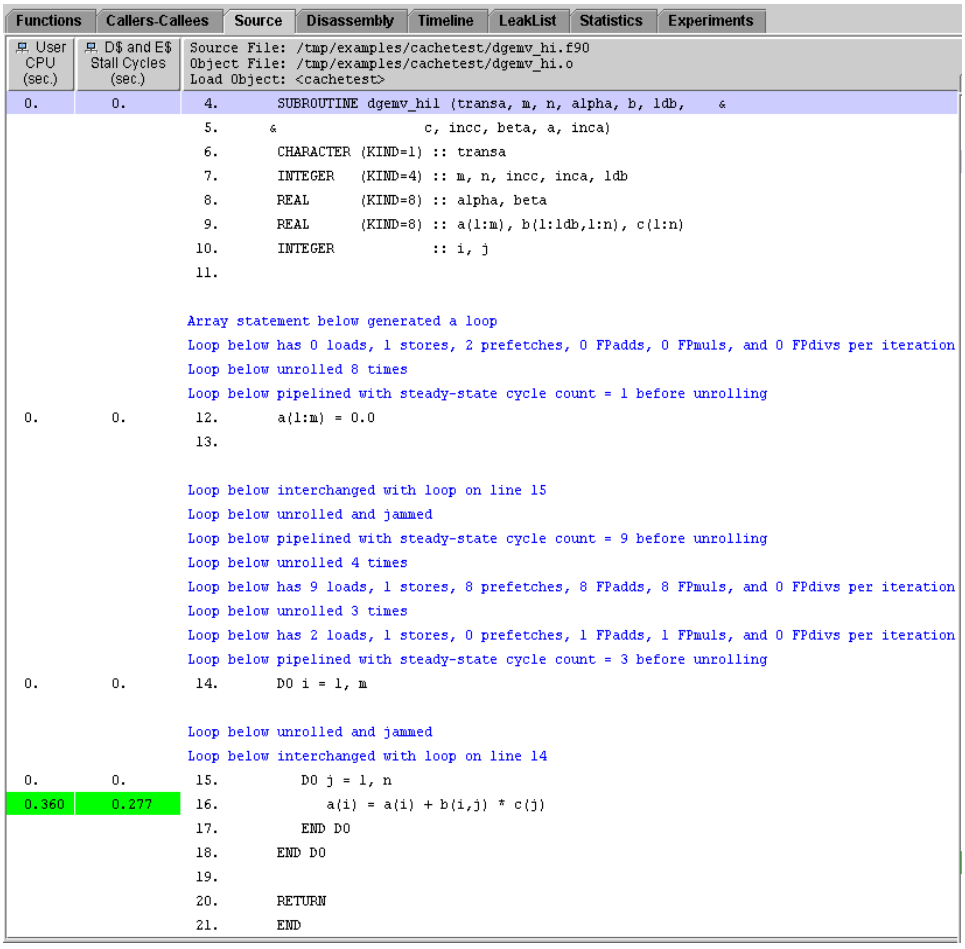


FIGURE 2-39 Source Tab for `dgemv_hi1` Showing Compiler Commentary That Includes Loop Interchange Messages

The compiler has done much more work to optimize this function. It has interchanged the loops that were the cause of the DTLB miss problems. In addition, the compiler has created new loops that have more floating-point add and floating-point multiply operations per loop cycle, and inserted prefetch instructions to improve the cache behavior.

Note that the messages apply to the loop that appears in the source code and any loops that the compiler generates from it.

6. **Scroll down to see the source code for `dgemv_hi2`.**

   The compiler commentary messages are the same as for `dgemv_hi1` except for the loop interchange. The code generated by the compiler for the two versions of the routine is now essentially the same.

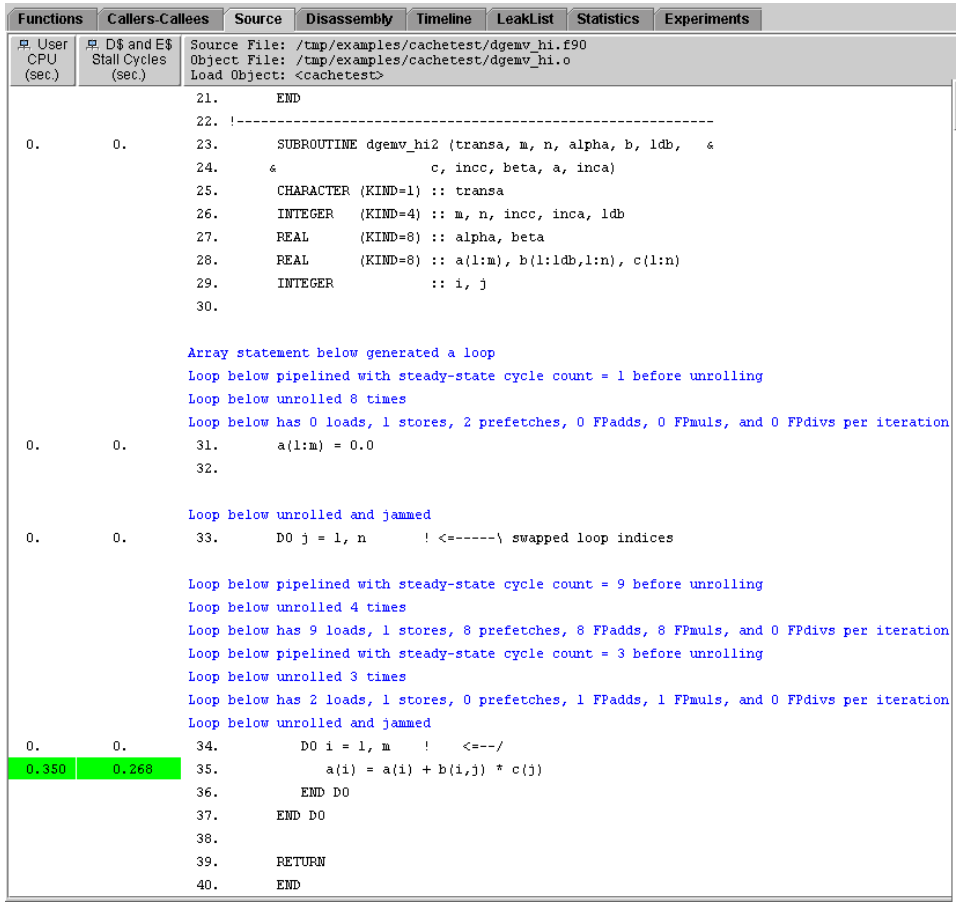


FIGURE 2-40 Source Tab for `dgemv_hi2` Showing Compiler Commentary

7. Click the Disassembly tab.

The line numbers for the instructions in the optimized code are no longer sequential. The compiler has rearranged the code: in particular, the initialization loop is now part of the main loop structure.

Compare the disassembly listing with that for `dgemv_g1` or `dgemv_opt1`. There are many more instructions generated for `dgemv_hi1` than for `dgemv_g1`. The loop unrolling and the insertion of prefetch instructions contribute to the larger number of instructions. However, the number of instructions executed in `dgemv_hi1` is the smallest of the three versions of the subroutine. Optimization can produce more instructions, but the instructions are used more efficiently and executed less often.

CHAPTER 3

Performance Data

The performance tools work by recording data about specific events while a program is running, and converting the data into measures 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. For information on collecting and storing performance data, see Chapter 4. For information on analyzing performance data, see Chapter 5 and Chapter 6.

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

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 ID, where available from the OS-- Solaris 9 or later
- A copy of the 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
- Synchronization wait tracing data
- Heap tracing (memory allocation) data
- MPI tracing data

These five data types, the metrics that are derived from them, and how you might use them, are described in the next five subsections.

Clock Data

In clock-based profiling, the state of each LWP is stored at regular time intervals. This time interval is called the profiling interval. The information is stored in an integer array: one element of the array is used for each of the ten microaccounting states maintained by the kernel.The data collected is converted by the Performance Analyzer into times spent in each state, with a resolution of the profiling interval. The default profiling interval is approximately 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 `collect` with no arguments will print 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 3-1 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 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.
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” on page 167) 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.

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.

The UltraSPARC® III processor family and the IA processor family have two registers that can be used to count events. The Collector can collect data from either or both 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 either register, others are only available on a particular register. Consequently, not all combinations of hardware counters can be chosen in a single experiment.

Hardware-counter 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.

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. For convenience, the performance tools provide aliases for a number of counters that are likely to be in common use. You can obtain a list of available hardware counters on any particular system from the Collector by typing `collect` with no arguments in a terminal window on that system.

The entries in the counter list for aliased counters are formatted as follows:

```
CPU Cycles (cycles = Cycle_cnt/*) 9999991 hi=1000003, lo=100000007
(CPU-cycles) Instructions Executed (insts = Instr_cnt/*) 9999991
hi=1000003, lo=100000007 (Events) D$ Read Misses (dcrm =
DC_rd_miss/1) 100003 hi=10007, lo=1000003 load (Events)
```

In the first line, the first field, "CPU Cycles", is the metric name. The second field, "cycles", gives the alias name that can be used in the -h counter... argument. The third field, "Cycle_cnt/*", gives the internal name as used by `cputrack(1)` and the register number on which that counter can be used. The register number will be either 0 or 1, or *, as in this case, indicating the counter is available on either register. The next field is the overflow interval, the following field is the high-resolution overflow interval, and the last field is the low- resolution overflow interval. The "(CPU-cycles)" indicates that the counter counts in units of CPU-cycles, and can be converted to time. The second line has a "(Events)" at the end of the line, indicating that it counts events, and can not be converted to time. A line may have an additional field, as shown in the third line above, between the low-resolution value and the counter units, indicating whether the counter may be triggered by a load, a store, or either, or if the counter is not program related.

The aliased counters that are available on both UltraSPARC and IA hardware are given in TABLE 3-2. There are other aliases that are available on UltraSPARC hardware.

TABLE 3-2 Aliased Hardware Counters Available on SPARC and IA Hardware

Aliased Counter Name	Metric Name	Description
`cycles`	CPU Cycles	CPU cycles, counted on either register
`insts`	Instructions Executed	Instructions executed, counted on either register

Lines of output for the non-aliased counters are formatted as follows:

```
Cycle_cnt Events (reg. 0) 1000003 hi=100003, lo=9999991 (CPU-
cycles)
Instr_cnt Events (reg. 0) 1000003 hi=100003, lo=9999991 (Events)
DC_rd Events (reg. 0) 1000003 hi=100003, lo=9999991 load (Events)
```

In this line, the first field, "Cycle_cnt", gives the internal name as used by `cputrack(1)` and the register number on which that counter can be used. The string "Cycle_cnt Events" is the metric name for this counter. The remainder of the line is formatted as for an aliased counter.

For both aliased and non-aliased counters that count in cycles, indicated by a "(CPU-cycles)" at the end of the its line, the metrics reported are converted by default to inclusive and exclusive times, but can optionally be shown as event counts. For counters that count in events, indicated by "(Events)" at the end of its line, the metrics reported are inclusive and exclusive event counts.

For hardware counters that relate to memory operations, as indicated by "load", "store", or "load-store" following the counter name, the name of the counter may be preceeded by a "+" sign, to request that the data collection attempt to find the precise instruction and effective address that caused the event on the counter that overflowed.

If a counter is not program related, using it for profiling will generate a warning, and profiling will not record a callstack, but rather will show the time being spent in an artificial function, `"collector_not_program_related"`. Thread and LWP ID's will be recorded, but are meaningless. The string `“not-program-related”` will appear after the name of any counter that counts events unrelated to the program running. Using such a counter reports metrics in the function `“collector_not_program_related”` and will give a warning before collection.

In the counter list, the aliased counters appear first, then all the counters available on register 0, then all the counters available on register 1. The aliased counters appear twice, with and without the alias. In the non-aliased list, these counters can have different overflow values. The default overflow values for the aliased cycle counters have been chosen to produce approximately the same data collection rate as for clock data. Other counters are much more sensitive to the actual behavior of the application.

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 functions in the threads library, `libthread.so`. 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.

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 callstacks 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 3-3 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`. 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 `malloc`, `free`, *etc.* also generates events that are recorded. Those events may come from native code, or from the JVM itself. In the machine representation, memory is allocated and deallocated by the JVM, typically in very large chunks. Memory allocation from the Java code is handled entirely by the JVM and its garbage-collector. Heap tracing will not show JVM allocations, since they are done by mapping memory, rather than calling the normal heap routines, and will not show any information about the Java memory allocation and garbage collection.

Heap tracing data is converted into the following metrics:

TABLE 3-4 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.

There is another definition of memory leaks that is commonly used, such as in the debugging tool, dbx. The definition 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. 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 3-5 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 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 3-6, categorized as MPI send functions, MPI receive functions, MPI send and receive functions, and other MPI functions.

TABLE 3-6 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 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 → New Sample, or click the New Sample button in the Debugging window, or use the `dbx collector sample record` command
- At a call to `collector_sample`, if you have put calls to this routine in your code (see "Controlling Data Collection From Your Program" on page 105)
- When a specified signal is delivered, if you have used the `-l` option with the `collect` command (see "Experiment Control Options" on page 122)
- When collection is initiated and terminated
- 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 on a particular time period. 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 at the bottom of a particular call stack (the “leaf function”), the exclusive and inclusive metrics are the same, because the function makes no calls to other functions.

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

Exclusive and inclusive metrics for a function give information about all 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 Function-Level Metrics: An Example

Exclusive, inclusive and attributed metrics are illustrated in FIGURE 3-1, 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.

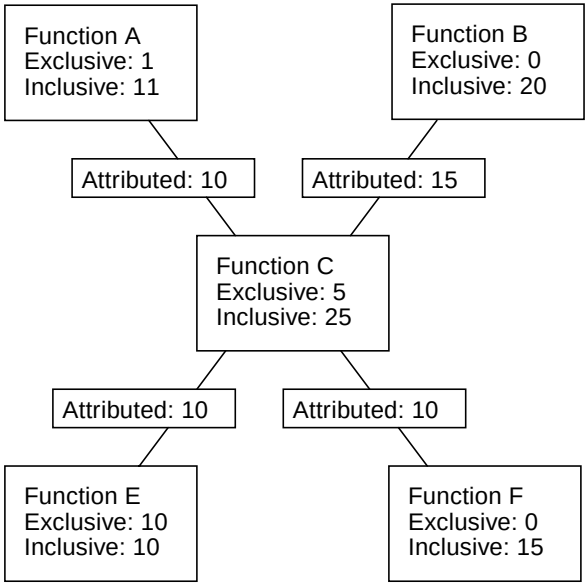


FIGURE 3-1 Call Tree Illustrating Exclusive, Inclusive, and Attributed Metrics

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

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

CHAPTER 4

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 3.

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 Using the `collect` Command
- Collecting Data Using the `dbx collector` Subcommands
- Collecting Data From a Running Process
- Collecting Data From MPI Programs

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, 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 via `"-xdebugformat=(stabs|dwarf)".`

To prepare compilation objects with debug information which allows correction of hardware counter profiles, currently only for the C compiler for SPARC®, compile 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"` to obtain full symbolic information.

Executables and libraries built with DWARF format debugging symbols automatically include a copy of each constituent object (.o) file's debugging symbols, and while this is also true for STABS format debugging symbols if linked with the `"-xs"` option, the default leaves STABS symbols in the various object files. If you need to move or remove the object files for any reason, you can link your program with the `"-xs"` option. 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 before analyzing it, for example.

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. This is 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.

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

Intermediate Files

If you generate intermediate files using the `-E` or `-P` compiler options, the Performance 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.

Compiling Java Programs

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

Preparing Your Program for Data Collection and Analysis

For most programs, you do not need to do anything special to prepare your program 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 loads a module (`.o` file)
- 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.

Use of 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.

- Collection of synchronization wait tracing data. The Collector interposes on functions from the threads library, `libthread.so`.
- Collection of 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`.
- Collection of MPI tracing data. The Collector interposes on functions from the MPI library, `libmpi.so`.
- 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.

There are some circumstances in which 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.

Use of Signal Handlers

The Collector uses two signals to collect profiling data, `SIGPROF` and `SIGEMT`. The Collector installs a signal handler for each of these signals, which intercept and process the signals, but pass on signals they do not use 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. It is the responsibility of the user 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 arrives late.

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 will be different from when the collector library is preloaded.

Use of `setuid`

There are restrictions enforced by the dynamic loader that 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.

Controlling Data Collection From Your Program

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, and a Fortran interface is also provided. Both C and

Fortran interfaces are defined in header files that are provided with the library. For Java programs, similar functionality is provided via the CollectorAPI class, which is described in the following section.

To use the API functions from C or C++, insert the following statement.

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

The 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);
```

To use the API functions from Fortran, insert the following statement:.

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

When you link your program, link with `-lfcollector`.

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

To use the Java API, use the following statement to import the CollectorAPI class. Note however that your application must be invoked with a classpath pointing to `<installation-directory>/lib/collector.jar` where `<installation-directory>` is the directory into which the Sun ONE Studio release was 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 Fortran API, excluding the dynamic function API.

The C include file 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. 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 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 descriptions of the API functions follow.

`collector_sample(char *name)` (C and C++)
`collector_sample(string)` (Fortran)
`CollectorAPI.sample(String)` (Java)

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.

`collector_pause()` (C ,C++, Fortran)

`CollectorAPI.pause()` (Java)

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.

`collector_resume()` (C ,C++, Fortran)

`CollectorAPI.resume()` (Java)

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.

`collector_thread_pause(unsigned int t)` (C and C++ only)

`CollectorAPI.threadPause(Thread)` (Java)

Stop writing event-specific data from the thread specified in the argument list to the experiment. The argument t is the POSIX thread identifier. 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.

`collector_thread_resume(unsigned int t)` (C and C++ only)

`CollectorAPI.threadResume(Thread)` (Java)

Resume writing event-specific data from the thread specified in the argument list to the experiment. The argument `t` is the POSIX thread identifier. 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.

`collector_terminate_expt()` (C ,C++, Fortran) `CollectorAPI.terminate` (Java)

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 or dynamically loads modules (`.o` files) 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);
void collector_module_load(char *modulename, void *vaddr);
void collector_module_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 four 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 4-1 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.
`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.

`collector_module_load()`

Used to inform the collector that the module `modulename` has been loaded into the address space at address `vaddr` by the program. The module is read to determine its functions and the source and line number mappings for these functions.

`collector_module_unload()`

Inform the collector that the module that was loaded 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 environment, 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.

In the Solaris 7 operating environment and earlier versions of the Solaris 8 operating environment, the system clock is used for profiling. 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 operating environment and later versions of the Solaris 8 operating environment, it is not necessary to enable the high-resolution system clock for high-resolution profiling.

Runtime Distortion and Dilation with Clock-profiling

Clock profiling records data when a SIGPROF signal is delivered to the target. It will cause dilation to process that signal, and unwind the callstack. The deeper the callstack, and the more frequent the signals, the greater the dilation. To a limited extent, clock-profiling will show 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`, has been preloaded. See “Collecting Data From a Running Process” on page 133 for more information.

Runtime Distortion and Dilation with Tracing

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

Limitations on Hardware-Counter Overflow Profiling

There are several limitations on hardware counter overflow profiling:

- You can only collect hardware-counter overflow data on processors that 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 operating environment that precede the Solaris™ 8 release.
- You can record data for at most two hardware counters in an experiment. To record data for more than two hardware counters or for counters that use the same register you must run separate experiments.
- You cannot collect hardware-counter overflow data 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.
- You cannot use the hardware counters in your own code via 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 is corrupted.

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

Runtime Distortion and Dilation With HWC Overflow Profiling

HW counter profiling records data when a SIGEMT is delivered to the target. It will cause dilation to process that signal, and unwind the callstack. Unlike clock profiling, for some HW counters, different parts of the program may generate events more rapidly than other parts, and will show dilation in that part of the code. Any part of the program that generates such events very rapidly may be significantly distorted. Similarly, some events may 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` option.
- You can collect data automatically for calls to `fork` and its variants and `exec` and its variants. Calls to `system`, `popen`, and `sh` are not followed by the Collector.
- 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 133 for more information.
- If you want to collect data for individual descendant processes, or those created by `system`, `popen`, `sh`, etc., you must use a separate `dbx` to attach to each process and enable the collector.

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 no earlier than 1.4.2. The path to the Java virtual machine[1] should be specified in one of the following four environment variables: `JDK_1_4_HOME`, `JDK_HOME`, `JAVA_PATH`, `PATH`. 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.
- If you want to use the 64 bit JVM™, it must either be the default, or you must specify the path to it when you collect data. Do not use `java -d64` to collect data using the 64 bit JVM. If you do, no data is collected.

Using JVM versions earlier than 1.4.2 will compromise the data as follows:

- **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. A significant amount of time will be shown in the `<Unknown>` function, since some of the code regions created by the JVM are not named. In addition, there are various bugs in JVM 1.4.1 that may cause the program being profiled to crash.

1. The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java platform.

- **JVM 1.4.0**: No Java representation is possible, and a significant amount of time is shown in <Unknown>. HotSpot-compiled functions are shown by name in the machine representation.
- **JVMs earlier than 1.4.0**: Profiling Java applications with JVMs earlier than 1.4.0 is not supported.

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

Java profiling uses the JVMPI interface, which can cause some distortion and dilation of the run. For clock- and hwc-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 estimated to be less than 10%.

In addition, 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 will lead 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 JVMPI events, which will cause 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 change 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 135 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 and x for an exec. 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.

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

In this section some guidelines are given 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 will be two callstacks of interest: the Java callstack and the machine callstack, which therefore will 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. There is currently no way to specify the size of these buffers. If the Collector runs out of memory, you should 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, you can consider collecting data for part of the run rather than the whole run. You can do this 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 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 90. See also “Limitations on Hardware-Counter Overflow Profiling” on page 113.

Data Collection Options

These options control the types of data that are collected. See “What Data the Collector Collects” on page 87 for a description of the data types.

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

If clock-based profiling is explicitly disabled, and neither 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 10 milliseconds.
- `lo[w]` – Turn on clock-based profiling with the low-resolution profiling interval of 100 milliseconds.
- `hi[gh]` – Turn on clock-based profiling with the high-resolution profiling interval of 1 millisecond. In the Solaris 7 operating environment and earlier versions of the Solaris 8 operating environment, high-resolution profiling must be explicitly enabled. See "Limitations on Clock-based Profiling" on page 111 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*[,*value*[,*counter2*[,*value2*]]]

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

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

If two counters are specified, they must use different registers. If they do not use different registers, the `collect` command prints an error message and exits. 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 91.

If the hardware counter counts events that relate to memory access, the counter name can be prefixed 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.

The overflow value is the number of events counted at which the hardware counter overflows and the overflow event is recorded. The overflow values can be specified using *value* and *value2*, which can be set to one of the following:

- `hi[gh]` – The high-resolution value for the chosen counter is used. The abbreviation `h` is also supported for compatibility with previous software releases.
- `lo[w]` – The low-resolution value for the chosen counter is used.
- *number* – The overflow value. Must be a positive integer.
- `on`, or a null string – The default overflow value is used.

The default is the normal threshold, which is predefined for each counter and which appears in the counter list. See also "Limitations on Hardware-Counter Overflow Profiling" on page 113.

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.

Heap tracing data is not recorded for Java memory allocations.

-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 95 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 on all descendant processes that are followed by the Collector.
- off – Do not record experiments on descendant processes.

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. The Collector does not follow processes created by calls to system(3C), system(3F), sh(3F), and popen(3C).

`-j` *option*

Enable Java profiling for a nonstandard Java installation, or choose whether to collect data on methods compiled by the Java HotSpot virtual machine. The allowed values of *option* are:

- `on` – Recognize methods compiled by the Java HotSpot virtual machine, and attempt to record Java stacks.
- `off` – Do not attempt to recognize methods compiled by the Java HotSpot virtual machine.

This 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 one of the following environment variables: `JDK_1_4_HOME`, `JDK_HOME`, `JAVA_PATH`, or `PATH`. 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. The `collect` command does not verify that *program* is a JVM machine, and collection can fail if it is not. However it does verify that *program* is an ELF executable, and 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, 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.

`-l` *signal*

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

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

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

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, you should 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

-d *directory-name*

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

-g *group-name*

Make the experiment part of experiment group *group-name*. If *group-name* does not end in `.erg`, the `collect` command prints an error message and exits. If the group exists, the experiment is added to it. 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.

-o *experiment-name*

Use *experiment-name* as the name of the experiment to be recorded. If *experiment-name* does not end in `.er`, the `collect` command prints an error message and exits. See "Experiment Names" on page 116 for more information on experiment names and how the Collector handles them.

-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, you should 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`.

Other Options

-n

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

-R

Display the text version of the performance tools readme in the terminal window. If the readme is not found, a warning is printed.

-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 10 milliseconds.
  - `lo[w]` – Use the low-resolution profiling interval of 100 milliseconds.
  - `hi[gh]` – Use the high-resolution profiling interval of 1 millisecond. In the Solaris 7 operating environment and earlier versions of the Solaris 8 operating environment, high-resolution profiling must be explicitly enabled. See “Limitations on Clock-based Profiling” on page 111 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 10 ms.

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 91 for a description of the list. If your system does not support hardware-counter overflow profiling, `dbx` returns a warning message.
- `counter` *name value* `[` *name2 value2* `]` – Selects the hardware counter *name*, and sets its overflow value to *value*; optionally selects a second hardware counter *name2* and sets its overflow value to *value2*. The overflow value can be one of the following.
  - `hi[gh]` – The high-resolution value for the chosen counter is used. The abbreviation `h` is also supported.
  - `lo[w]` – The low-resolution value for the chosen counter is used.
  - *number* – The overflow value. Must be a positive integer.
  - `on` – The default overflow value is used.

  The two counters 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, the counter name can be prefixed 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 113.

`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 disabled, it is ignored with a warning. If no process is running, it enables data collection for subsequent runs.

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

`pause`

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

`resume`

Resumes data collection after a `pause` has been issued. This subcommand is ignored if data 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 115 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 Program Performance Analysis Tools Readme, which is available through the documentation index at `file:/opt/SUNWspro/docs/index.html`. If the Sun ONE Studio 8 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 to Solaris Process and select the process using the dialog box. Use the online help for instructions.
   - From dbx, type the following.

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

   If dbx is not already running, type the following.

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

See the 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 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 "Use of System Libraries" on page 104 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 4-2 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 ONE 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 will be halted and restarted at a different time. The time difference could change the interaction between the MPI processes and affect the performance data you collect. To minimize this problem, one solution is to use pstop(1) to halt all the processes. However, once you attach dbx to the processes, you must restart them from dbx, and there will be a timing delay in restarting the processes, which can affect the synchronization of the MPI processes. See also “Collecting Data From MPI Programs” on page 135.

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 of the ClusterTools software if possible, which is 4.0, 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 subsequent subsections.

Storing MPI Experiments

Because multiprocessing environments can be complex, there are some issues about storing MPI experiments you should be aware of when you collect performance data from MPI programs. These issues concern the efficiency of data collection and storage, and the naming of experiments. See “Where the Data Is Stored” on page 115 for information on naming experiments, including MPI experiments.

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

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

If you store the experiments on a local file system and specify an experiment name in the standard format, the names are not unique. For example, suppose you ran an MPI job on a machine with 4 single-processor nodes labelled `node0`, `node1`, `node2`

and node3. Each node has a local disk called /scratch, and you store the experiments in directory *username* on this disk. The experiments created by the MPI job have the following full path names.

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

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

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

For large MPI jobs, you might want to move the experiments to a common location using a script. Do not use the Unix commands cp or mv; see "Manipulating Experiments" on page 235 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 will not have to rename the experiments when you copy them and reconstruct any experiment group file.

If you do not know which local file systems are available to you, use the df -lk command or ask your system administrator. You should always make sure that the experiments are stored in a directory that already exists, that is uniquely defined and that is not in use for any other experiment. You should also make sure that the file system has enough space for the experiments. See "Estimating Storage Requirements" on page 117 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 115 for information on where and how to store the experiments.

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

Collecting Data by Starting `dbx` Under MPI

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

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

Here, *n* is the number of processes to be created by MPI and *collection-script* is a `dbx` script that contains the commands necessary to set up and start data collection. This procedure creates *n* separate instances of `dbx`, each of which records an experiment on one of the MPI processes. If you do not define the experiment name, the experiment will be labelled with the MPI rank. Read the section "Storing MPI Experiments" on page 136 for information on where and how to store the experiments.

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

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

In a Fortran program you would insert the following line.

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

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

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

CHAPTER 5

The Performance Analyzer Graphical User Interface

The Performance Analyzer analyzes the program performance data that is collected by the Sampling Collector. This chapter provides a brief description of the Performance Analyzer GUI, its capabilities, and how to use it. The online help system of the Performance Analyzer provides information on new features, the GUI displays, how to use the GUI, interpreting performance data, finding performance problems, troubleshooting, a quick reference, keyboard shortcuts and mnemonics, and a tutorial.

This chapter covers the following topics.

- Running the Performance Analyzer
- The Performance Analyzer Displays
- Using the Performance Analyzer

For an introduction to the Performance Analyzer in tutorial format, see Chapter 2.

For a more detailed description of how the Performance Analyzer analyzes data and relates it to program structure, see Chapter 7.

Running the Performance Analyzer

Starting the Analyzer from the Command Line

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

```
% analyzer [-h] [-j jvm-path] [-J jvm-options] [-V] [-v] [experiment-list]
```

Here, *experiment-list* is a list of experiment names or experiment group names. See “Where the Data Is Stored” on page 115 for information on experiment names. If you omit the experiment name, the Open Experiment dialog box is displayed when the Performance Analyzer starts. If you give more than one experiment name, the data for all experiments are added in the Performance Analyzer.

The options for the `analyzer` command are described in TABLE 5-1.

TABLE 5-1 Options for the `analyzer` Command

`-h`	Prints a usage message for the `analyzer` command
`-j` *jvm-path*	Specify the path to the Java™ virtual machine used to run the Performance Analyzer
`-J` *jvm-options*	Specify options to the JVM™ machine used to run the Performance Analyzer
`-v`	Print information while the Performance Analyzer is starting
`-V`	Prints the version number of the Performance Analyzer to `stdout`

To exit the Performance Analyzer, choose File → Exit.

Starting the Analyzer from the IDE

For information on starting the Performance Analyzer from the IDE, see the Program Performance Analysis Tools Readme, which is available through the documentation index at `file:/opt/SUNWspro/docs/index.html`. If the Sun ONE Studio 8, software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

The Performance Analyzer Displays

The Performance Analyzer window contains a menu bar, a tool bar, and a split pane for data display. Each pane of the split pane contains several tab panes that are used for the displays of the Performance Analyzer. The Performance Analyzer window is shown in FIGURE 5-1.

The Menu Bar

The menu bar contains a File menu, a View menu, a Timeline menu and a Help menu. In the center of the menu bar, the selected function or load object is displayed in a text box. This function or load object can be selected from any of the tabs that display information for functions. From the File menu you can open new Performance Analyzer windows that use the same experiment data. From each window, whether new or the original, you can close the window or close all windows.

The Toolbar

The toolbar contains a number of buttons grouped according to menu.

The first group contains buttons related to the File menu: 

- Open Experiment
- Add Experiment
- Drop Experiment
- Create Mapfile
- Print
- Create New Window
- Close

The second group contains buttons related to the View menu: 

- Set Data Presentation
- Filter Data
- Show / Hide Functions

The third group contains buttons related to the Timeline menu:

- Back One Event
- Forward One Event
- Up One Bar
- Down One Bar
- Reset Display
- Zoom in x2
- Zoom Out x2
- Show Function Color Chooser

Additionally, the toolbar contains a Find text box, with buttons to Find Previous and Find Next. 

The following subsections describe what is displayed in each of the tabs.

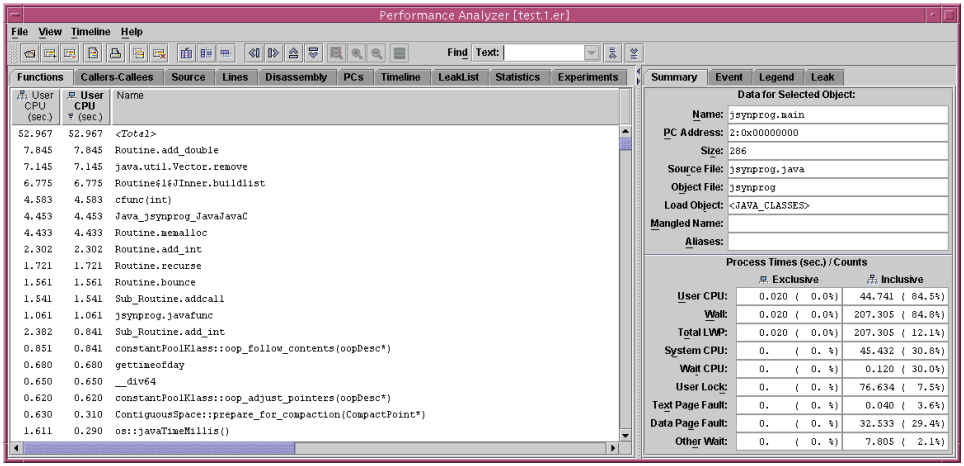


FIGURE 5-1 The Performance Analyzer Window

The Functions Tab

The Functions tab shows a list of functions and load objects and their metrics. Only the functions that have non-zero metrics are listed. The term *functions* includes Fortran functions and subroutines, C functions, C++ functions and methods, and Java™ methods. The function list in the Java representation shows metrics against the interpreted Java methods, and any native methods called. The Expert-Java representation additionally lists methods that were dynamically compiled by the HotSpot virtual machine. In the machine representation, multiple HotSpot compilations of a given method will be shown as completely independent functions, although the functions will all have the same name. All functions from the JVM will be shown as such.

The Functions tab can display inclusive metrics and exclusive metrics. The metrics initially shown are based on the data collected and on the default settings. The function list is sorted by the data in one of the columns. This allows you to easily identify which functions have high metric values. The sort column header text is displayed in bold face and a triangle appears in the lower left corner of the column header. Changing the sort metric in the Functions tab changes the sort metric in the Callers-Callees tab unless the sort metric in the Callers-Callees tab is an attributed metric.

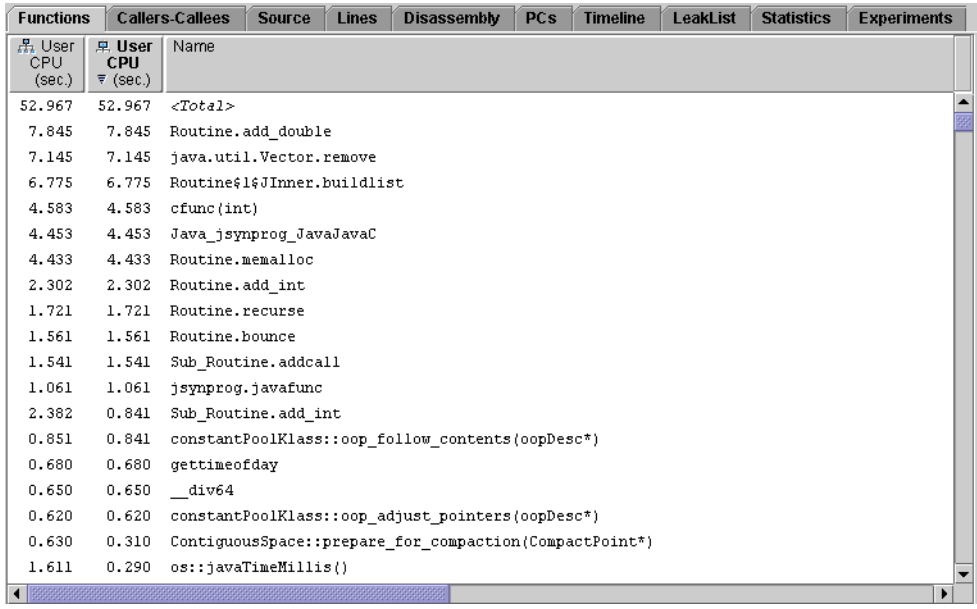


FIGURE 5-2 The Functions Tab

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 it, and callees of that function in a pane below it. Functions that appear in the Functions tab can appear in the Callers-Callees tab.

In addition to showing exclusive and inclusive metric values for each function, the tab also shows attributed metrics. If either an inclusive or an exclusive metric is shown, the corresponding attributed metric is also shown. The default metrics shown are derived from the metrics shown in the Function List display.

The percentages given for attributed metrics are the percentages that the attributed metrics contribute to the selected function's inclusive metric. For exclusive and inclusive metrics, the percentages are percentages of the total program metrics.

You can navigate through the structure of your program, searching for high metric values, by selecting a function from the callers or the callees pane. Whenever a new function is selected in any tab, the Callers-Callees tab is updated to center it on the selected function.

The callers list and the callees list are sorted by the data in one of the columns. This allows you to easily identify which functions have high metric values. The sort column header text is displayed in bold face. Changing the sort metric in the Callers-Callees tab changes the sort metric in the Functions tab.

In the machine representation for applications written in the Java programming language, the caller-callee relationships will show all overhead frames, and all frames representing the transitions between interpreted, compiled, and native methods.

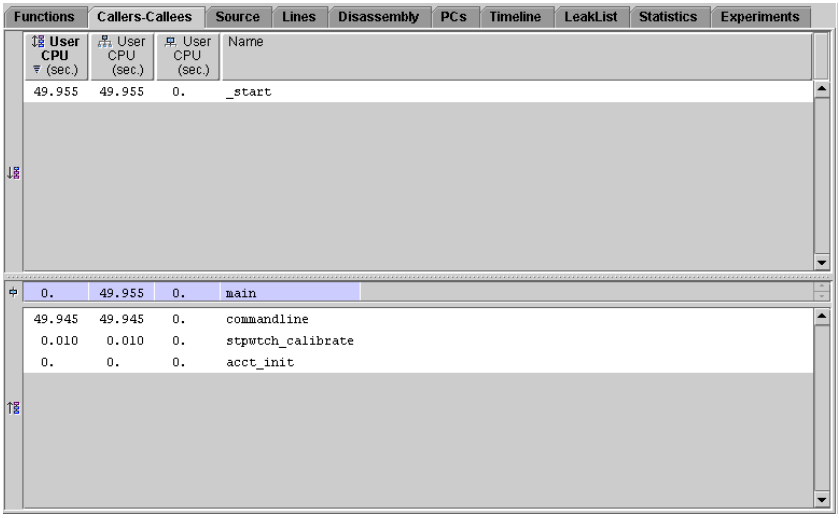


FIGURE 5-3 The Callers-Callees Tab

The Source Tab

The Source tab shows the source file that contains the selected function. Each line in the source file for which instructions have been generated is annotated with performance metrics. If compiler commentary is available, it appears above the source line to which it refers.

Lines with high metric values have the metrics highlighted. A high metric value is one that exceeds a threshold percentage of the maximum value of that metric on any line in the file. The entry point for the function you selected is also highlighted.

The choice of performance metrics, compiler commentary and highlighting threshold can be changed in the Set Data Presentation dialog box. The default choices can be set in a defaults file. See “Default-Setting Commands” on page 188 for more information on setting defaults.

You can view annotated source code for a C or C++ function that was dynamically compiled if you provide information on the function using the collector API, but you only see non-zero metrics for the selected function, even if there are more functions in the source file.

The 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. In the machine representation, the source from compiled methods will be shown against the Java source; the data will represent the specific instance of the compiled-method selected.

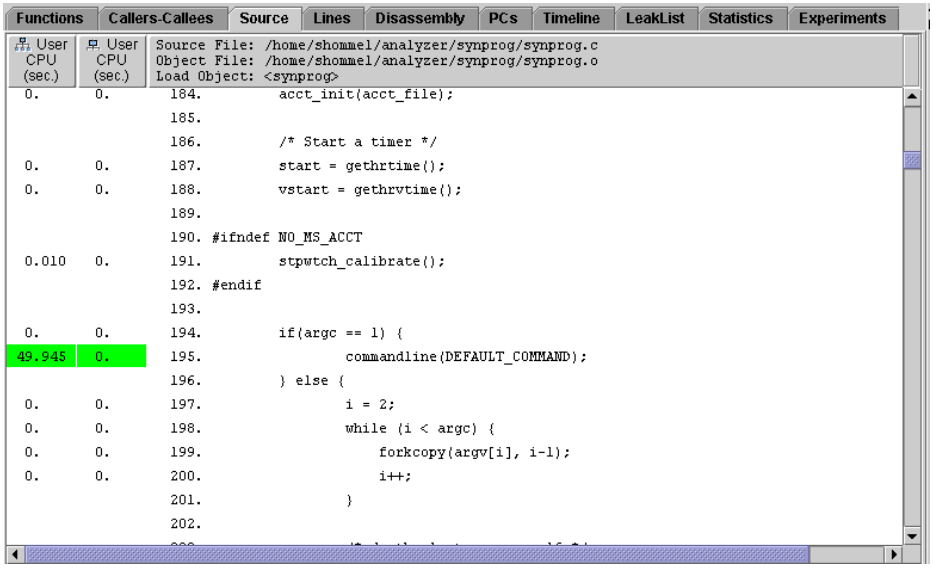


FIGURE 5-4 The Source Tab

The Lines Tab

The Lines tab shows a list of source lines and their metrics. The source lines are represented by the function name followed by the line number and the source file name.

The source lines are ordered by the data in one of the columns. This allows you to easily identify which lines have high metric values. The sort column header text is displayed in bold face and a triangle appears in the lower left corner of the column header. You can select the sort metric column by clicking its column header.

If you select a source line, this line becomes the selected object, and is displayed in the Selected Object text box. When you click the Source tab, the source code from which the line came is displayed with the source line selected. When you click the Functions tab or Callers-Callees tab, the function from which the line came is the selected function.

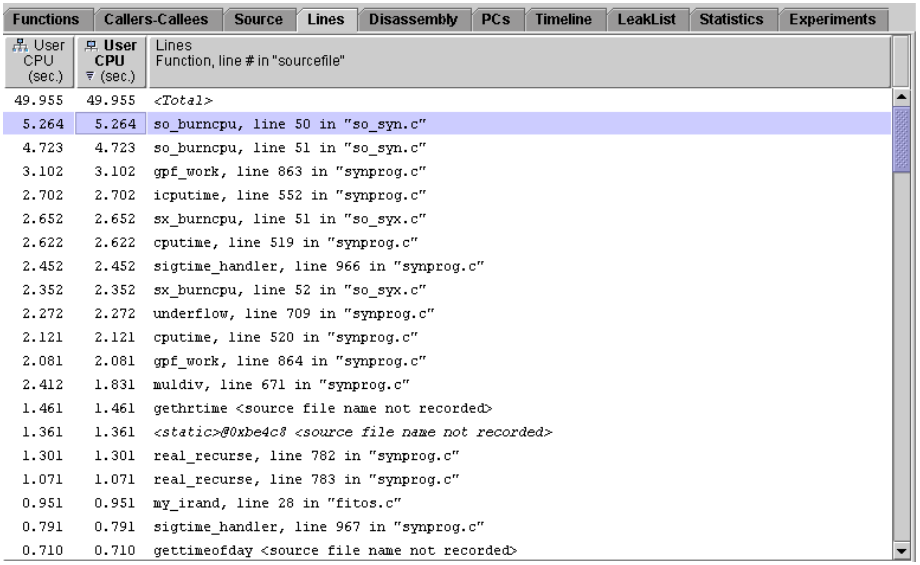


FIGURE 5-5 The Lines Tab

The Disassembly Tab

The Disassembly tab shows a disassembly listing for the object file that contains the selected function, annotated with performance metrics for each instruction. The instructions can also be displayed in hexadecimal. Instructions that are marked with an asterisk are synthetic instructions. These instructions are generated for hardware counters that count memory access events if the search for the PC that triggered the event is unsuccessful.

If the compilation object was compiled with debugging information, and the source code is available, it is inserted into the listing. Each source line is placed above the first instruction that it generates. Source lines can appear in blocks when compiler optimizations of the code rearrange the order of the instructions. If compiler commentary is available it is inserted with the source code. The source code can also be annotated with performance metrics.

If the compilation object was compiled with support for hardware counter profiling (see “Source Code Information” on page 102) control transfer targets are distinguished from the (immediately following) instruction at the (same) address

with the label "<branch target>" and by marking their address with an asterisk. Hardware counter events corresponding to memory operations which were collected with backtracking enabled (see "`-h` counter`[`,value`[`,counter2`[`,value2`]]]`" on page 120 and "`hwprofile` option" on page 128) may be associated with these synthetic instructions whenever they prevent the causal instruction from being determined.

If the compilation object was compiled with both debugging information and support for hardware counter profiling, memory referencing instructions may be annotated with the referenced dataobject descriptor, which constitutes the basis for program data-oriented analyses (see "The Data Objects Tab" on page 151).

Lines with high metric values have the metric highlighted. A high metric value is one that exceeds a threshold percentage of the maximum value of that metric on any line in the file.

If the selected function was dynamically compiled, you only see instructions for that function. If you provided information on the function using the Collector API (see "Dynamic Functions and Modules" on page 109), you only see non-zero source metrics for the specified function, even if there are more functions in the source file. You can see instructions for Java compiled methods without using the Collector API. The disassembly of any Java method shows the bytecode generated for it, with metrics against each bytecode, and interleaved Java source, where available. In the machine representation, disassembly for compiled methods will show the generated machine assembler code, not the Java bytecode.

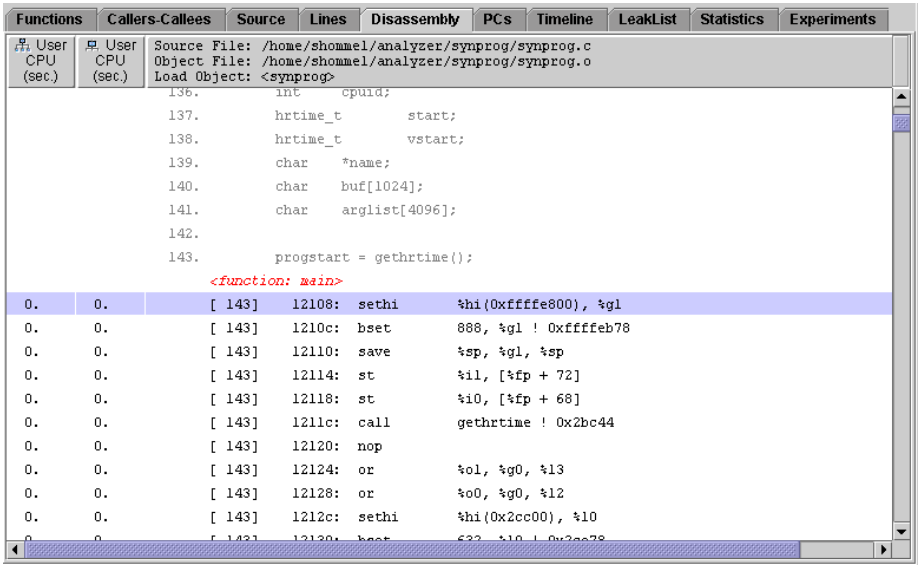


FIGURE 5-6 The Disassembly Tab

The choice of performance metrics, compiler commentary, highlighting threshold, source annotation and hexadecimal display can be changed in the Set Data Presentation dialog box. The default choices can be set in a defaults file. See “Default-Setting Commands” on page 188 for more information on setting defaults.

The PCs Tab

The PCs tab shows a list of program counter addresses and the metrics for the corresponding instructions. The PCs are represented by the function name and the offset relative to the start of the function.

For hardware counter experiments profiling events corresponding to memory operations which were collected with backtracking enabled (see “`-h` counter`[,`value`[,`counter2`[,`value2`]]]” on page 120 and “`hwprofile` option” on page 128), a PC may have been adjusted to that of the most likely memory-referencing instruction, or backtracking may have been blocked, e.g., by an intervening control transfer target. Where backtracking is blocked, a synthetic PC is created to distinguish it from the actual instruction PC: such synthetic PCs are visibly distinguished with an appended asterisk character (e.g., `"main + 0x0000ABC4*"` represents the synthetic control transfer target with the same address as the instruction `"main + 0x0000ABC4"`).

The list of PCs is ordered by the data in one of the columns. This allows you to easily identify which PCs have high metric values. The sort column header text is displayed in bold face and a triangle appears in the lower left corner of the column header. You can select the sort metric column by clicking its column header.

If you select a PC from the list this PC becomes the selected object. When you click the Disassembly tab, the disassembly listing for the function from which the PC came is displayed with the PC selected. When you click the Source tab, the source listing for the function from which the PC came is displayed with the line containing the PC selected. When you click the Functions tab or Callers-Callees tab, the function from which the PC came is the selected function.

The choice of performance metrics and sort metric can be changed in the Set Data Presentation dialog box. The default choices can be set in a defaults file. See “Default-Setting Commands” on page 188 for more information on setting defaults.

For applications written in the Java programming language, a PC for a method (in the Java representation) corresponds to the method-id and a bytecode index into that method; a PC for a native function corresponds to a machine PC. The callstack for a Java thread may have a mixture of Java PCs and machine PCs. It will not have any frames corresponding to Java housekeeping code, which does not have a Java representation.

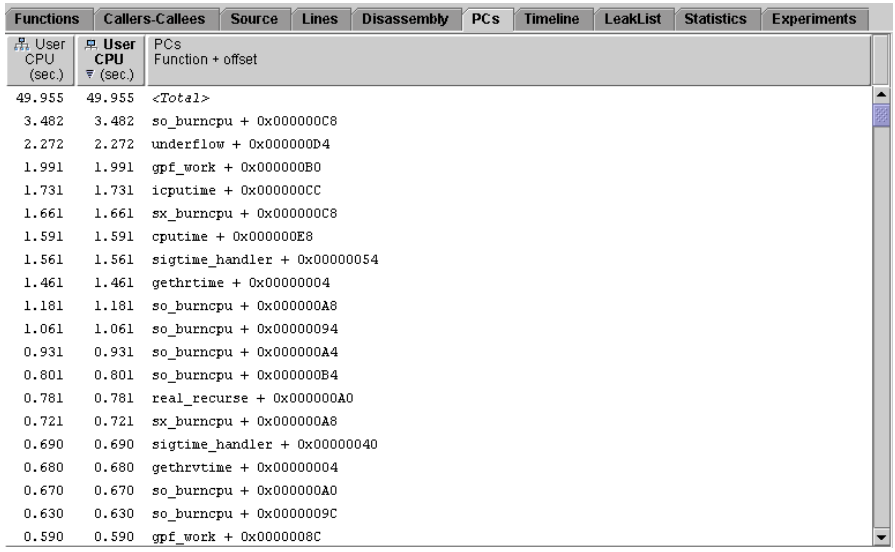


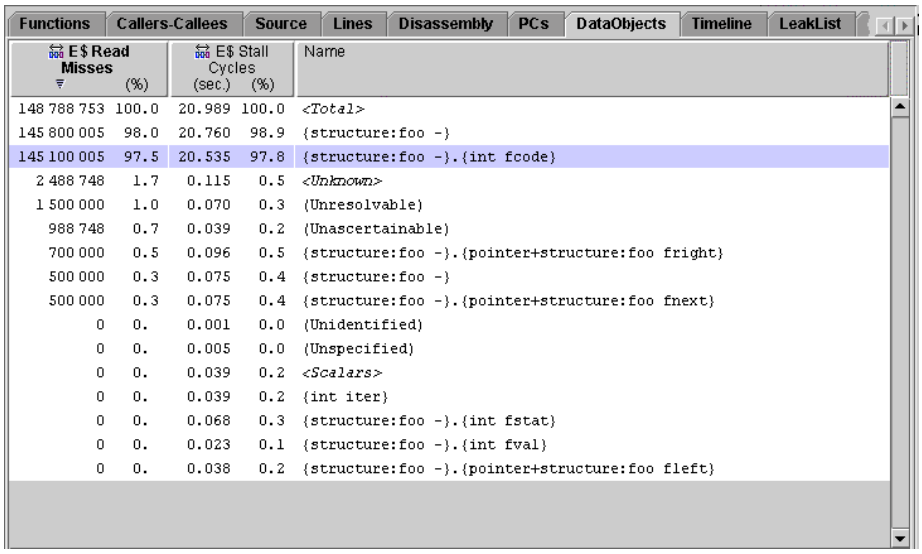
FIGURE 5-7 The PCs Tab

The Data Objects Tab

The Data Objects tab is only presented when Data Space Display Mode has been enabled (see “The Formats Tab” on page 163 and “`datamode { on| off }`” on page 191). The Data Objects tab shows a list of dataobjects and their metrics. Only dataobjects that have non-zero metrics are listed. The term “dataobjects” includes program constants, variables, arrays and aggregates such as structures and unions, along with distinct aggregate elements. Various synthetic dataobjects are also defined as required (see “The <Unknown> Dataobject” on page 224).

The Data Objects tab shows only data-derived metrics from hardware counter events for memory operations collected with backtracking enabled for compilation objects built with associated hardware profiling support. The metrics initially shown are based on the data collected and the data presentation settings for inclusive and exclusive (code) metrics. The dataobject list is sorted by the data in one of the columns. This allows you to easily identify which dataobjects have high metric values. The sort column header text is displayed in bold face and a triangle appears in the lower left corner of the column header. The initial sort metric is based on the corresponding inclusive or exclusive (code) metric, if a data-derived metric variant is appropriate.

Data-derived metrics apply only to dataobjects, and are similar to inclusive (code) metrics: the metric value for an element of an aggregate is also included in the metric value for the aggregate.



E$ Read Misses	(%)	E$ Stall Cycles (sec.)	(%)	Name
148 788 753	100.0	20.989	100.0	*<Total>*
145 800 005	98.0	20.760	98.9	{structure:foo -}
145 100 005	97.5	20.535	97.8	{structure:foo -}.{int fcode}
2 488 748	1.7	0.115	0.5	*<Unknown>*
1 500 000	1.0	0.070	0.3	(Unresolvable)
988 748	0.7	0.039	0.2	(Unascertainable)
700 000	0.5	0.096	0.5	{structure:foo -}.{pointer+structure:foo fright}
500 000	0.3	0.075	0.4	{structure:foo -}
500 000	0.3	0.075	0.4	{structure:foo -}.{pointer+structure:foo fnext}
0	0.	0.001	0.0	(Unidentified)
0	0.	0.005	0.0	(Unspecified)
0	0.	0.039	0.2	*<Scalars>*
0	0.	0.039	0.2	{int iter}
0	0.	0.068	0.3	{structure:foo -}.{int fstat}
0	0.	0.023	0.1	{structure:foo -}.{int fval}
0	0.	0.038	0.2	{structure:foo -}.{pointer+structure:foo fleft}

FIGURE 5-8 Data Objects Tab

The Timeline Tab

The Timeline tab shows a chart of events as a function of time. The event and sample data for each experiment and each LWP (or thread or CPU) is displayed separately, rather than being aggregated. The Timeline display allows you to examine individual events recorded by the Collector.

Data is displayed in horizontal bars. The display for each experiment consists of a number of bars. By default, the top bar shows sample information, and is followed by a set of bars for each LWP, one bar for each data type (clock-based profiling, hardware counter profiling, synchronization tracing, heap tracing), showing the events recorded. The bar label for each data type contains an icon that identifies the data type and a number in the format *n.m* that identifies the experiment (*n*) and the LWP (*m*). LWPs that are created in multithreaded programs to execute system threads are not displayed in the Timeline tab, but their numbering is included in the LWP index. See “Parallel Execution and Compiler-Generated Body Functions” on page 210 for more information. You can choose to display data for threads or for CPUs (if recorded in an experiment) rather than for LWPs, using the Timeline Options dialog box. The index *m* is then the index of the thread or the CPU.

The sample bar shows a color-coded representation of the process times, which are aggregated in the same way as the timing metrics. Each sample is represented by a rectangle, colored according to the proportion of time spent in each microstate. Clicking a sample displays the data for that sample in the Event tab. When you click a sample, the Legend and Summary tabs are dimmed.

The event markers in the other bars consist of a color-coded representation of part of the call stack. Each function in the call stack is represented by a small colored rectangle. These rectangles are aligned vertically. By default, the leaf function is at the top. The call stack can be aligned on the leaf function or the root function using the Timeline Options dialog box. The color coding of the functions in the call stack is displayed in the Legend tab and can be changed using the Timeline Color Chooser dialog box.

Selecting a sample bar or event marker results in the corresponding horizontal data channel being highlighted, along with a vertical cursor showing the duration of the sample or event. This will be a line, 1 pixel wide, if the event is instantaneous or of short duration.

Clicking a colored rectangle in an event marker selects the corresponding function and PC from the call stack and displays the data for that event and that function in the Event tab. The selected function is highlighted in both the Event tab and the Legend tab and the PC address for the event is displayed in the menu bar as a function with an offset from the function. Clicking the Disassembly tab displays the annotated disassembly code for the function with the line for the PC selected.

FIGURE 5-9 The Timeline Tab

The Event tab is displayed by default in the right pane when the Timeline tab is selected. The Legend tab, also in the right pane, shows color-coding information for functions.

The default choice of data type, display type (by LWP, CPU or thread), call stack alignment and maximum depth can be set in a defaults file. See “Default-Setting Commands” on page 188 for more information on setting defaults.

In the Java representation, each Java thread's event callstack is shown with its Java methods. In the machine representation, the timeline will show bars for all threads LWPs or CPUs, and the callstack in each will be the machine-representation callstack.

The LeakList Tab

The LeakList tab shows all leak and allocation events that occurred in the program. The tab is divided into two panels: a top panel, showing leak events, and a bottom panel, showing allocation events. Timeline information appears at the top of the tab, and call stacks for the events appear in both panels.

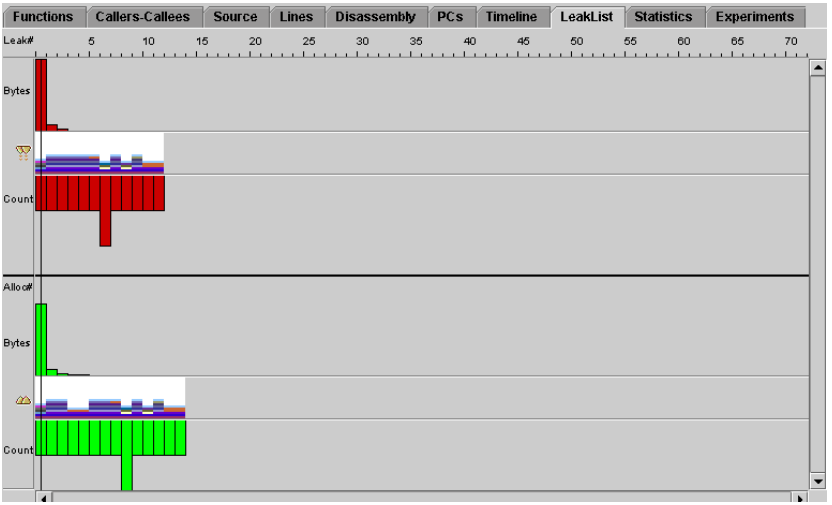


FIGURE 5-10 The LeakList Tab

The leak and allocation event panels are each subdivided into three sections: the number of bytes leaked / allocated, the call stack for the selected event, and the number of times the leak or allocation has occurred. To select an individual leak or allocation event, single-click on any data portion of the display. Data for the selected event will be displayed in the Leak tab on the right side of the main analyzer display. Pressing the arrow buttons in the toolbar will step event selection back and forth between the displayed events in each bar. The up and down buttons are disabled because, unlike on the timeline, there is no correlation between leaks and allocations.

The Statistics Tab

The Statistics tab shows totals for various system statistics summed over the selected experiments and samples, followed by the statistics for the selected samples of each experiment. The process times are summed over the microaccounting states in the same way that metrics are summed. See “Clock Data” on page 88 for more information.

The statistics displayed in the Statistics tab should in general match the timing metrics displayed for the `<Total>` function in the Functions tab. The values displayed in the Statistics tab are more accurate than the microstate accounting values for `<Total>`. But in addition, the values displayed in the Statistics tab include other contributions that account for the difference between the timing metric values for `<Total>` and the timing values in the Statistics tab. 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 7 and 8 operating environments 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.

For information on the definitions and meanings of the execution statistics that are presented, see the `getrusage`(3C) and `proc`(4) man pages.

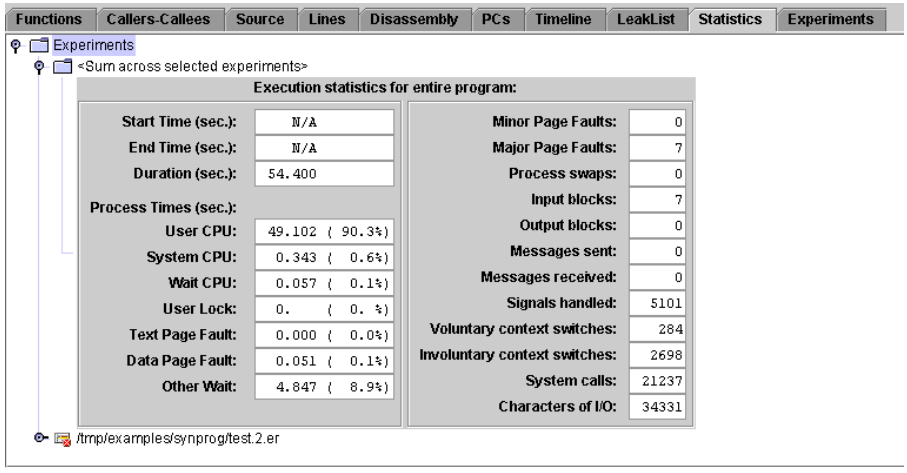


FIGURE 5-11 The Statistics Tab

The Experiments Tab

The Experiments tab is divided into two panes.

The top pane contains a tree that shows information on the experiments collected and on the load objects accessed by the collection target. The information includes any error messages or warning messages generated during the processing of the experiment or the load objects. Experiments that were incomplete but otherwise readable by the Analyzer have a cross in a red circle superimposed on the experiment icon.

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

FIGURE 5-12 The Experiments Tab

Data in the Experiments tab is organized into a hierarchical tree, with “Experiments” shown as its root node. Beneath that is the “Load Objects” branch node, with additional nodes representing each currently loaded experiment. When expanded, the “Load Objects” node will list all loadobjects in the experiments, with any errors or warnings recorded at the time of archiving, and a message about the process that did the archiving. In addition, if `"-A copy"` was specified to `collect`, or the `"collector archive copy"` command was given to `dbx`, or the `"-A"` flag was given during an explicit invocation of `er_archive`, a copy of each load object (the `a.out` and any shared objects referenced) will be copied into the archive subdirectory. Expanding an experiment node reveals information about how the experiment was collected, such as the target command, collector version, host name, data collection parameters, and warning messages.

The archive subdirectory

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

The Summary Tab

The upper section of the Summary tab shows information about the selected object. When the selected object is a loadobject, a function, a source line or a PC, the information displayed includes the name, address and size, and for functions, source lines and PCs, the name of the source file, object file and load object. Selected functions are shown with their mangled names and also any aliases which may have been defined. In Dataspace Display mode (see “The Formats Tab” on page 163 and “`datamode { on| off }`” on page 191), the alias for a selected PC is its descriptor (when it is ascertainable from the available debugging information). When the selected object is a dataobject, the information displayed includes the name, scope, type and size. If the selected dataobject is an aggregate (such as a structure or union), a list is shown with its elements and their sizes and offsets in the aggregate. If the selected dataobject is a member of an aggregate, the aggregate's name is shown along with the member's offset in the aggregate.

The lower section of the Summary tab shows all the available metrics for the selected object. For loadobjects, functions, source lines and PCs, both exclusive and inclusive metrics are shown as values and percentages, with an additional line for hardware counters that count in cycles. For dataobjects, only data-derived metrics are shown.

The information in the Summary tab is not affected by metric selection. The Summary tab is updated whenever a new object is selected.

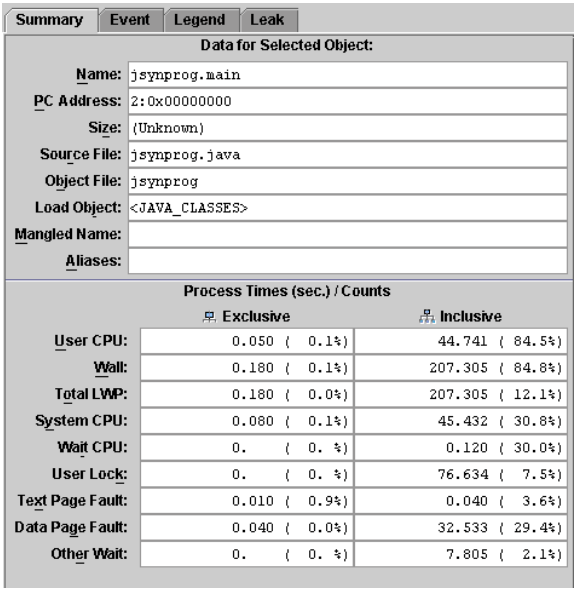


FIGURE 5-13 The Summary Tab

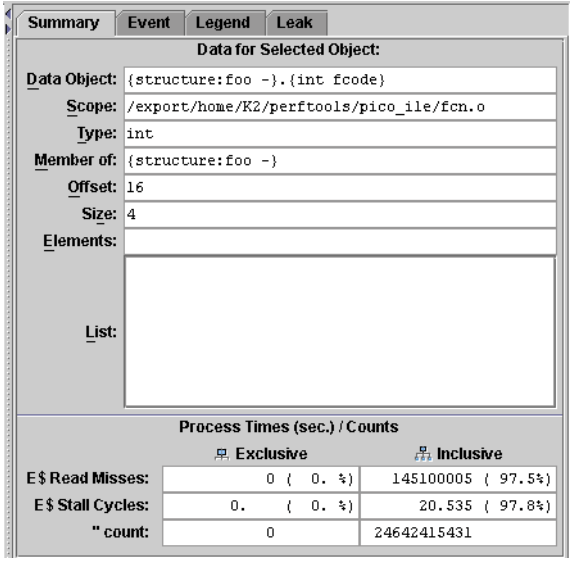


FIGURE 5-14 Data Objects Summary

The Event Tab

The Event tab shows the available data for the selected event, including the experiment name, event type, leaf function, timestamp, LWP ID, thread ID, CPU ID, Duration, and Micro State information. Below the data panel the call stack is displayed with the color coding that is used in the event markers for each function in the stack. Clicking a function in the call stack makes it the selected function.

For hardware counter events corresponding to memory operations which were collected with backtracking enabled (see "`-h` counter`[,`value`[,`counter2`[,`value2`]]]`" on page 120 and "`hwprofile` option" on page 128), corresponding data address information is also shown where determinable and verifiable.

When a sample is selected, the Event tab shows the sample number, the start and end time of the sample, and a list of timing metrics. For each timing metric the amount of time spent and the color coding is shown. The timing information in a sample is more accurate than the timing information recorded in clock profiling.

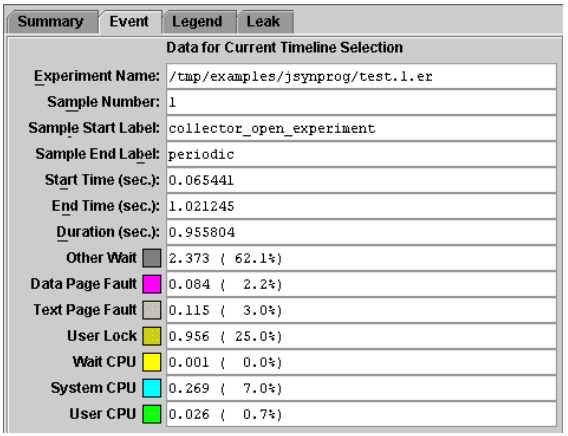


FIGURE 5-15 The Event Tab, Showing Event Data

The Legend Tab

The Legend tab shows the mapping of colors to functions for the display of events in the Timeline tab. The Legend tab is only enabled when an event is selected in the Timeline tab. It is dimmed when a sample is selected in the Timeline tab. The color coding can be changed using the color chooser in the Timeline menu.

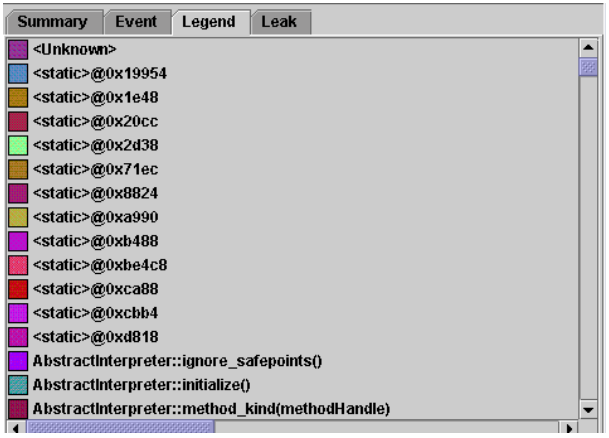


FIGURE 5-16 The Legend Tab

The Leak Tab

The leak tab shows detailed data for the selected leak or allocation in the LeakList tab. It is divided into two panels: a top panel, showing data for the LeakList, and a bottom panel, showing the call stack for the selected event.

The top panel displays the event type, leak/allocation number, number of bytes leaked or allocated, and the instances count. In the bottom panel, clicking on a function in the call stack makes it the selected function.

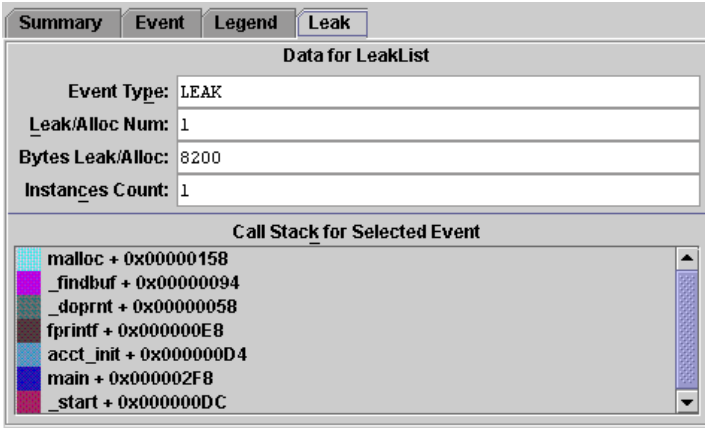


FIGURE 5-17 The Leak Tab

Using the Performance Analyzer

This section describes some of the capabilities of the Performance Analyzer and how its displays can be configured.

Comparing Metrics

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

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

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

Selecting Experiments

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

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

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

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

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

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

Selecting the Data to Be Displayed

Once you have experiment data loaded into the Performance Analyzer, there are various ways for you to select what is displayed.

The Set Data Presentation Dialog

You can open the Set Data Presentation dialog box using the following toolbar button, or by selecting View->Set Data Presentation... from the menu.

The Set Data Presentation dialog contains the following individual tabs: Metrics, Sort, Source / Disassembly, Formats, Timeline, and Search Path.

The Metrics Tab

You can select the metrics that are displayed and the sort metric using the Metrics and Sort tabs of the Set Data Presentation dialog box. The choice of metrics applies to all tabs. The Callers-Callees tab adds attributed metrics for any metric that is chosen for display.

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

The Sort Tab

The Sort tab allows you to set the sort metric and order in which the metric columns are displayed. To change the sort metric, double-click the metric or its radio button. To change the order of the metrics, click the metric then use the Move Up or Move Down buttons to move the metric.

The sort tab allows the data to be sorted by any of the following:

- Inclusive or Exclusive User CPU
- Inclusive or Exclusive Wall
- Inclusive or Exclusive Total LWP
- Inclusive or Exclusive System CPU
- Inclusive or Exclusive Wait CPU
- Inclusive or Exclusive User Lock
- Inclusive or Exclusive Text Page Fault
- Inclusive or Exclusive Data Page Fault
- Inclusive or Exclusive Other Wait
- Size
- PC Address
- Name

The visible metrics appear in **bold** text.

The Source/Disassembly Tab

You can select the threshold for highlighting high metric values, select the classes of compiler commentary and choose whether to display metrics on annotated source code and whether to display the hexadecimal code for the instructions in the annotated disassembly listing from the Source / Disassembly tab of the Set Data Presentation dialog box.

The Formats Tab

The formats tab allows you to specify whether you want C++ function names to be displayed in short or long form. The long form is the full, demangled name including parameters; the short form does not include the parameters. The formats tab also allows you to set the Java representation (to on, expert, or off), and lets you enable or disable the data space display.

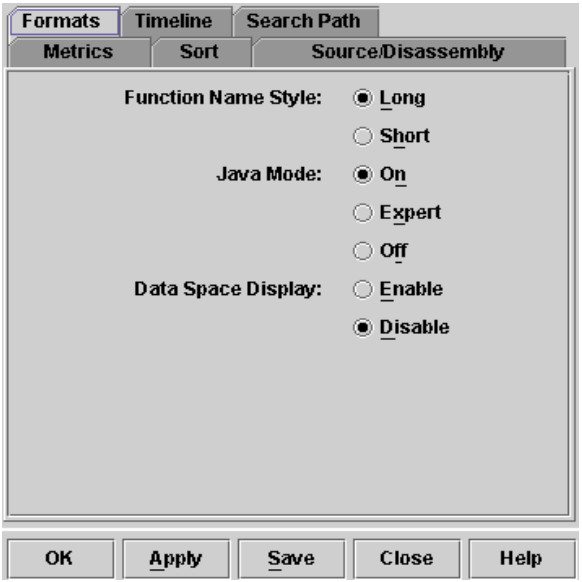


FIGURE 5-18 The Formats Tab

The Timeline Tab

You can choose to display event data for LWPs, for threads or for CPUs in the Timeline tab, choose the number of levels of the call stack to display, choose the alignment of the call stacks in the event markers, and select the data types to display.

The Search Path Tab

Sets the path used for finding source, object etc. files. The search path is also used to locate the `.jar` files for the Java™ Runtime Environment (JRE) on your system. The special directory name `$expts` refers to the set of current experiments, in the order in which they were loaded. To change the search order, single-click on an entry and press the Move Up / Move Down buttons. The compiled-in full pathname will be used if a file is not found in searching the current path setting.

The Filter Data Dialog

Filtering by Experiment, Sample, Thread, LWP and CPU. You can control the information in the Performance Analyzer displays by specifying only certain experiments, samples, threads, LWPs, and CPUs for which to display metrics. You make the selection using the Filter Data dialog box. Selection by thread, by sample, and by CPU does not apply to the Timeline display.

You can open the Filter Data dialog box using the following toolbar button:

The Show / Hide Functions Dialog

Showing and Hiding Functions. For each load object, you can choose whether to show metrics for each function separately or to show metrics for the load object as a whole, using the Show / Hide Functions dialog box. You can open the Show / Hide Functions dialog box using the following toolbar button, or by selecting View->Show / Hide Functions... from the menu.

Setting Defaults

The settings for all the data displays are initially determined by a defaults file, which you can edit to set your own defaults.

The default metrics are read from a defaults file. In the absence of any user defaults files, the system defaults file is read. A defaults file can be stored in a user's home directory, where it will be read each time the Performance Analyzer is started, or in any other directory, where it will be read when the Performance Analyzer is started from that directory. The user defaults files, which must be named `.er.rc`, can contain selected `er_print` commands. See "Default-Setting Commands" on

page 188 for more details. The selection of metrics to be displayed, the order of the metrics and the sort metric can be specified in the defaults file. The following table summarizes the system default settings for metrics.

TABLE 5-2 Default Metrics Displayed in the Functions Tab

Data Type	Default Metrics
clock-based profiling	inclusive and exclusive User CPU time
hardware-counter overflow profiling	inclusive and exclusive times (for counters that count in cycles) or event counts (for other counters)
synchronization delay tracing	inclusive synchronization wait count and inclusive synchronization delay time
heap tracing	inclusive leaks and inclusive bytes leaked
MPI tracing	inclusive MPI Time, inclusive MPI Bytes Sent, inclusive MPI Sends, inclusive MPI Bytes Received, inclusive MPI Receives, and inclusive MPI Other

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

For C++ programs, you can display the long or the short form of a function name. The default is long. This choice can also be set up in the defaults file.

You can save any settings you make in the Set Data Presentation dialog box in a defaults file.

See “Default-Setting Commands” on page 188 for more information about defaults files and the commands that you can use in them.

Searching for Names or Metric Values

Find tool. The Performance Analyzer includes a Find tool in the toolbar that you can use to locate text in the Name column of the Functions tab and the Callers-Callees tab, and in the code column of the Source tab and the Disassembly tab. You can also use the Find tool to locate a high metric value in the Source tab and the Disassembly tab. High metric values are highlighted if they exceed a given threshold of the maximum value in a source file. See “Selecting the Data to Be Displayed” on page 162 for information on selecting the highlighting threshold.

Generating and Using a Mapfile

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

To create the mapfile, you must compile your program with the `-g` option or the `-xF` option. Both of these options ensure that the required symbol table information is inserted into the object files.

The order of the functions in the mapfile is determined by the metric sort order. If you want to use a particular metric to order the functions, you must collect the corresponding performance data. Choose the metric carefully: the default metric is not always the best choice, and if you record heap tracing data, the default metric is likely to be a very poor choice.

To use the mapfile to reorder your program, you must ensure that your program is compiled using the `-xF` option, which causes the compiler to generate functions that can be relocated independently, and link your program with the `-M` option.

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

Here, *compiler* is one of `f95`, `cc` or `CC`.

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

This chapter covers the following topics.

- `er_print` Syntax
- Metric Lists
- Commands Controlling the Function List
- Command Controlling the Callers-Callees List
- Commands Controlling the Leak and Allocation Lists
- Commands Controlling the Source and Disassembly Listings
- Commands Controlling the Data Space List
- Commands Listing Experiments, Samples, Threads, and LWPs
- Commands Controlling Selections
- Commands Controlling Load Object Selection
- Commands That List Metrics
- Commands That Control Output
- Commands That Print Other Displays
- Default-Setting Commands
- Default-Setting Commands Affecting Only the Performance Analyzer
- Miscellaneous Commands

For a description of the data collected by the Collector, see Chapter 3.

For instructions on how to use the Performance Analyzer to display information in a graphical format, see Chapter 5.

er_print Syntax

The command-line syntax for er_print is as follows.

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

The options for er_print are listed in TABLE 6-1.

TABLE 6-1 Options for the er_print Command

Option	Description
-	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 er_print are listed in the following sections. You can abbreviate any command with a shorter string as long as the command is unambiguous.

Metric Lists

Many of the er_print commands use a list of metric keywords. The syntax of the list is as follows.

```
metric-keyword-1[:metric-keyword2...]
```

Except for the `size`, `address`, and `name` keywords, a metric keyword consists of three parts: a metric type string, a metric visibility string, and a metric name string. These are joined with no spaces, as follows.

```
<type><visibility><name>
```

The metric type and metric visibility strings are composed of type and visibility characters.

The allowed metric type characters are given in TABLE 6-2. A metric keyword that contains more than one type character is expanded into a list of metric keywords. For example, `ie.user` is expanded into `i.user:e.user`.

TABLE 6-2 Metric Type Characters

Character	Description
`e`	Show exclusive metric value
`i`	Show inclusive metric value
`a`	Show attributed metric value (only for callers-callees metrics)

The allowed metric visibility characters are given in TABLE 6-3. The order of the visibility characters in the visibility string does not matter: it does not affect the order in which the corresponding metrics are displayed. For example, both `i%.user` and `i.%user` are interpreted as `i.user:i%user`.

Metrics that differ only in the visibility are always displayed together in the standard order. If two metric keywords that differ only in the visibility are separated by some other keywords, the metrics appear in the standard order at the position of the first of the two metrics.

TABLE 6-3 Metric Visibility Characters

Character	Description
`.`	Show metric as a time. Applies to timing metrics and hardware counter metrics that measure cycle counts. Interpreted as "+" for other metrics.
`%`	Show metric as a percentage of the total program metric. For attributed metrics in the callers-callees list, show metric as a percentage of the inclusive metric for the selected function.
`+`	Show metric as an absolute value. For hardware counters, this value is the event count. Interpreted as a "`.`" for timing metrics.
`!`	Do not show any metric value. Cannot be used in combination with other visibility characters.

When both type and visibility strings have more than one character, the type is expanded first. Thus `ie.%user` is expanded to `i.%user:e.%user`, which is then interpreted as `i.user:i%user:e.user:e%user`.

The visibility characters "`.`", "`+`" and "`%`" are equivalent for the purposes of defining the sort order. Thus `sort i%user`, `sort i.user`, and `sort i+user` all mean "sort by inclusive user CPU time if it is visible in any form", and `sort i!user` means "sort by inclusive user CPU time, whether or not it is visible".

TABLE 6-4 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 90 for more information on hardware counters.

TABLE 6-4 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 6-4 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 6-4, there are two name strings that can only be used in default metrics lists. These are `hwc`, which matches any hardware counter name, and `any`, which matches any metric name string. Also note that `cycles` and `insts` are common to SPARC® and Intel, but other flavors also exist that are architecture-specific. To list all available counters, use the `collect` command with no arguments.

Commands Controlling the Function List

The following commands control the display of function information.

`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.

The number of lines written can be limited by using the `limit` command (see “Commands That Control Output” on page 186).

The default metrics printed are exclusive and inclusive user CPU time, in both seconds and percentage of total program metric. You can change the current metrics displayed with the `metrics` command. This must be done before you issue the `functions` command. You can also change the defaults with the `dmetrics` command.

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. Setting the Java mode to “expert” shows HotSpot-compiled methods separately from the interpreted version of the method. Setting the mode to “off” shows functions from the JVM itself, rather than from the Java application being interpreted by the JVM, along with any compiled methods and native methods.

`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 `er_print` to display the following metrics:

- Inclusive user CPU time in seconds
- Inclusive user CPU time percentage
- Exclusive user CPU time in seconds
- Exclusive user CPU time percentage

When the `metrics` command is finished, a message is printed showing the current metric selection. For the preceding example the message is as follows.

```
current: i.user:i%user:e.user:e%user:name
```

For information on the syntax of metric lists, see “Metric Lists” on page 170. 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 170, as shown in this example.

```
% sort i.user
```

This command tells `er_print` to sort the function list by inclusive user CPU time. If the metric is not in the experiments that have been loaded, a warning is printed and the command is ignored. When the command is finished, the sort metric is printed.

`fsummary`

Write a summary metrics panel for each function in the function list. The number of panels written can be limited by using the `limit` command (see "Commands That Control Output" on page 186).

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.

Command Controlling the Callers-Callees List

The following commands control the display of caller and callee information.

callers-callees

Print the callers-callees panel for each of the functions, in the order in which they are sorted. The number of panels written can be limited by using the `limit` command (see "Commands That Control Output" on page 186). 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 170. 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 170, as shown in this example.

```
% csort a.user
```

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

Commands Controlling the Leak and Allocation Lists

This section describes commands relating 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 Controlling the Source and Disassembly Listings

The following commands control the display of annotated source and disassembly code.

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, `er_print` prints a list of possible object-file names; if the name you gave was a function, the name of the function is appended to the object-file name, and the number that represents the value of *N* for that object file is also printed.

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

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

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

`scc` *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 6-5 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.
`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 a `.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. The following options can be added to the class list.

TABLE 6-6 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, it should be escaped with a backslash. The special directory name, `$expts`, refers to the set of current experiments, in the order in which they were loaded; it may be abbreviated with a single `$` character.

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

`setpath` with no argument will print the current path.

`addpath` *path_list*

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

Commands Controlling the Data Space List

`data_objects`

Write the list of data objects with their metrics. Applicable only to HW counter experiments where aggressive backtracking was specified, and for objects in files that were compiled with `-xhwcprof`. (Available on SPARC for C only). See the C compiler manual 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 HW counter experiments where aggressive backtracking was specified, and for objects in files that were compiled with `-xhwcprof`. (Available on SPARC for C only). See the C compiler manual for further information.

Commands Listing Experiments, Samples, Threads, and LWPs

This section describes commands that are used to 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 Controlling 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 Controlling Load Object Selection

object_list

Display the list of load objects. The name of each load object is preceded by a “+” if its functions are shown in the function list, and by a “-” if its functions are not shown in the function list.

The following example is an example of a load object list.

```
(er_print) object_list
Sel Load Object
=== ================================================
yes /tmp/var/synprog/synprog
yes /opt/SUNWspro/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. If you specify a dash (-) instead of *filename*, output is written to standard output.

limit *n*

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

name { long | short }

Specify whether to use the long or the short form of function names (C++ only).

javamode { on | expert | off }

Set the mode for Java experiments to on (show the Java model), expert (show the Java model, but show HotSpot-compiled methods independently from interpreted methods), 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.

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 186).

`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

and 8 operating environments 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.

Default-Setting Commands

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

A defaults file can be included in your home directory, to set defaults for all experiments, or in any other directory, to set defaults locally. When `er_print`, `er_src` or the Performance Analyzer is started, the current directory and your home directory are scanned for defaults files, which are read if they are present, and the system defaults file is also read. Defaults from the `.er.rc` file in your home directory override the system defaults, and defaults from the `.er.rc` file in the current directory override both home and system defaults.

Note – To ensure that you read the defaults file from the directory where your experiment is stored, you must start the Performance Analyzer or the `er_print` utility from that directory.

The defaults file can also include the `scc`, `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 170. 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 "!", 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 170.

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

`gdemangle` *library.so*

Set the path to the shared object that supports an API to demangle C++ function names. The shared object must export the C function `cplus_demangle()`, conforming to the GNU standard `libiberty.so` interface.

Default-Setting Commands Affecting Only 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 6-7 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 6-8 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 6-8 Timeline Display Data Types

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 `er_print`.

`quit`

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

`help`

Print a list of `er_print` commands.

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 in some detail 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
- Annotated Code Listings

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, what versions of various components, and a record of various events during the life of the target.
- A loadobjects 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." 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-profiling	profile
HWC-profiling	hwcounters
Synchronization-tracing	synctrace
Heap-tracing	heaptrace
MPI-tracing	mpitrace

For clock-profiling, or HW-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 `LD_PRELOAD` 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.

The archive subdirectory

Each experiment has an archive subdirectory, which contains binary files describing each loadobject referenced in the loadobjects file. These files are produced by `er_archive`, which runs at the end of data collection. If the process terminates abnormally, `er_archive` may not be invoked, in which case, the archive files are written by `er_print` or Analyzer when firt 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 and “x” for exec), 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 execs 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

Experiments where the target creates dynamic functions have 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

Java experiments also have additional records in the loadobjects file, both for dynamic functions created by the JVM for its internal purposes, and for dynamically-compiled (HotSpot) versions of the target Java methods.

Java experiments also have additional records in the loadobjects file, both for dynamic functions created by the JVM for its internal purposes, and for dynamically-compiled (HotSpot) versions of the target Java methods.

In addition, Java experiments have a `JAVA_CLASSES` file, containing information about all of the user's Java classes invoked.

Java heap- and synchronization tracing data is recorded using a JVMPI agent, which is part of libcollector.so. It 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

There are three different ways to record an experiment, with the collect command, with dbx creating a process, and with dbx creating an experiment from a running process.

collect Experiments

When collect is used to record an experiment, the collect program itself creates the experiment directory, sets LD_PRELOAD to ensure that libcollector.so is preloaded into the target's address space, and sets LD_AUDIT to ensure that collaudit.so is preloaded into the target's address space, and will be invoked by ld.so in processing all shared object loads and unloads. It then sets environment variables to inform libcollector about the experiment name, and data collection options, and exec's the target on top of itself.

In collect experiments, libcollector.so is responsible for writing all experiment files, other than the loadobjects file. collaudit.so is responsible for writing the shared-object records in the loadobjects file, while libcollector.so writes the dynamic function and compiled-method records.

dbx Experiments, Creating the Process

When dbx is used to launch a process with data collection enabled, it also creates the experiment directory, and ensures preloading of libcollector.so. It stops the process at a breakpoint before its first instruction, and then calls an initialization routine in libcollector to start the data collection. dbx is responsible for writing the loadobjects records that are written by collaudit.so in a collect experiment, but libcollector.so otherwise behaves as it does in a collect experiment.

Java experiments can not be collected by dbx, since dbx uses a JVMDI agent for debugging, and that agent can not coexist with the 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 can not use LD_PRELOAD. It makes an interactive function call into the target to dlopen libcollector.so, and then calls libcollector.so's initialization routine, just as it does when creating the process. Data is written by libcollector.so just as in a collect experiment.

dbx is responsible for writing the loadobjects records that are written by collaudit.so in a collect experiment, but libcollector.so otherwise writes loadobjects records as it does 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), may not work. In general, the symbols will 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 will not work properly for experiments created by dbx on a running process.

If the user has explicitly LD_PRELOAD'ed libcollector.so before starting the process with dbx, or before using dbx to attach to the running process, tracing data may be collected.

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 for each of the ten microstates maintained by the kernel for each LWP. At the end of the profiling interval, the count for the microstate of each LWP is

incremented by 1, and a profiling signal is scheduled. The array is only recorded and reset when the LWP is in user mode in the CPU. If the LWP is in user mode when the profiling signal is scheduled, the array element for the User-CPU state is 1, and the array elements for all the other states are 0. If the LWP is not in user mode, the data is recorded when the LWP next enters user mode, and the array can contain an accumulation of counts for various states.

The call stack is recorded at the same time as the data. If the LWP is not in user mode at the end of the profiling interval, the call stack cannot change until the LWP enters user mode again. Thus the call stack always accurately records the position of the program counter at the end of each profiling interval.

The metrics to which each of the microstates contributes are shown in TABLE 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. You should 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, there are specific errors that arise from the way the data is collected and attributed and the way the program progresses through the system. Some of the circumstances in which inaccuracies or distortions can appear in the timing metrics are described in what follows.

- When an LWP is created, the time it has spent before the first profile packet is recorded is less than the profiling interval, but the entire profiling interval is ascribed to the microstate recorded in the first profile packet. If there are many LWPs created the error can be many times the profiling interval.
- When an LWP is destroyed, some time is spent after the last profile packet is recorded. If there are many LWPs destroyed the error can be many times the profiling interval.
- LWP rescheduling 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 LWPs to run than there are processors to run them.
- It is possible for a program to behave in a way which is correlated with the system clock. In this case, the profiling interval always expires when the LWP is in a state which might represent a small fraction of the time spent, and the call stacks recorded for a particular part of the program are overrepresented. On a multiprocessor system, it is possible for the profiling signal to induce a correlation: processors that are interrupted by the profiling signal while they are running LWPs for the program are likely to be in the Trap-CPU microstate when the microstate is recorded.
- The kernel records the microstate value when the profiling interval expires. When the system is under heavy load, that value might not represent the true state of the process. This situation is likely to result in overaccounting of the Trap-CPU or Wait-CPU microstate.
- The threads library sometimes discards profiling signals when it is in a critical section, resulting in an underaccounting of timing metrics.
- 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 profiling signal. (This is another instance of correlation.) The user CPU time spent in the recording process is distributed over whatever microstates are recorded. The result is an underaccounting of the User CPU Time metric and an overaccounting of other metrics. The amount of time spent recording data is typically less than one percent of the CPU time for the default profiling interval.

Comparisons of Timing Metrics

If you compare timing metrics obtained from the profiling done in a clock-based experiment with times obtained by other means, you should be aware of the following issues.

For a single-threaded application, the total LWP time recorded for a process is usually accurate to a few tenths of a percent, compared with the values returned by `gethrtime`(3C) for the same process. The CPU time can vary by several percentage points from the values returned by `gethrvtime`(3C) for the same process. Under heavy load, the variation might be even more pronounced. However, the CPU time differences do not represent a systematic distortion, and the relative times reported for different functions, source-lines, and such are not substantially distorted.

For multithreaded applications using unbound threads, differences in values returned by `gethrvtime()` could be meaningless. This is because `gethrvtime()` returns values for an LWP, and a thread can change from one LWP to another.

The LWP times that are reported in the Performance Analyzer can differ substantially from the times that are reported by `vmstat`, because `vmstat` reports times that are summed over CPUs. If the target process has more LWPs than the system on which it is running has CPUs, the Performance Analyzer shows more wait time than `vmstat` reports.

The microstate timings that appear in the Statistics tab of the Performance Analyzer and the `er_print` statistics display are based on process file system usage reports, for which the times spent in the microstates are recorded to high accuracy. See the `proc`(4) man page for more information. You can compare these timings with the metrics for the `<Total>` function, which represents the program as a whole, to gain an indication of the accuracy of the aggregated timing metrics. 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™ 7 and 8 operating environments 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

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. The fraction of the wait time spent in overhead can be minimized by increasing the threshold for recording events.

Hardware-Counter Overflow Profiling

Hardware-counter overflow profiling data includes a counter ID and the overflow value. The value can be larger than the value at which the counter is set to overflow, because the processor executes some instructions between the overflow and the recording of the event. This is especially true of cycle and instruction counters, which are incremented much more frequently than counters such as floating-point operations or cache misses. The delay in recording the event also means that the program counter address recorded with call stack does not correspond exactly to the overflow event. See “Attribution of Hardware Counter Overflows” on page 232 for more information.

The amount of data collected depends on the overflow value. Choosing a value that is too small can have the following consequences.

- The amount of time spent 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.

MPI Tracing

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”.

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 215 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 it in turn is shown as calling the user's signal handler. Inclusive metrics from both `sigacthandler()` and any user signal handler, and any other functions they call, appear as inclusive metrics for the interrupted 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 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 which are initiated from an instruction and trap into the kernel. All system calls are implemented using trap instructions, for example. Some examples of hardware traps are those issued from the floating point unit when it is unable to complete an instruction (such as the `fitos` instruction on the UltraSPARC® III platform), or when the instruction is not implemented in the hardware.

When a trap is issued, the LWP enters system mode. The microstate is usually switched from User 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 which 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 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.

There is one case in which the trap handler switches back to user mode, and that is the misaligned memory reference trap for an 8-byte integer which is aligned on a 4-byte boundary in Fortran. A frame for the trap handler appears on the stack, and a call to the handler can appear in the Performance Analyzer, attributed to the integer load or store instruction.

When an instruction traps into the kernel, the instruction following the trapping instruction appears to take a long time, because it cannot start until the kernel has finished executing the trapping instruction.

Tail-Call Optimization

The compiler can do one particular optimization whenever the last thing a particular 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 (light-weight process). 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 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 operating environment or of the alternate threads library in the Solaris 8 operating environment.

  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 7 and Solaris 8 operating environments creates some threads 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 operating environment and the alternate threads library in the Solaris 8 operating environment do not create these extra threads.

Overview of Java Technology-Based Software Execution

To the typical developer, a Java™ technology-based application runs just like any other program. It 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 [1]Java virtual machine (JVM). 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 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 will be an interpreted version, and there may also be one or more compiled versions. Code written in the Java programming language may also call directly into native-compiled code, either C, C++, 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 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. It also has 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

1. The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java™ platform.

more complex relationship. But for any version of the library, at any instant a thread may be unscheduled, or scheduled onto an LWP. Data for a thread is not collected while that thread is not scheduled onto an LWP.

The Sun ONE™ Studio performance tools collect their data by recording events in the life of the each LWP of the process, along with the callstack at the time of the event. At any point in the execution of any application, Java technology-based or otherwise, the callstack represents where the program is in its execution, and how it got there. One important way that mixed-model Java technology-based 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 callstack, and a machine callstack. Both stacks are collected and correlated with each other.

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. By default, where the data supports it, the Java representation is shown. 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 may 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 mode is selected in the analyzer by default.

The Expert-Java Representation

The Expert-Java Representation is similar to the Java Representation, except that HotSpot-compiled methods are shown independently of the interpreted version of the method. Some details of the JVM internals that are suppressed in the Java Representation are exposed in the Expert-Java Representation.

The Machine Representation

The Machine Representation shows functions from the JVM itself, rather than from the application being interpreted by the JVM. It also shows all compiled and native methods. The machine representation looks the same as that of applications written in traditional languages. The callstack 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.

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™ ONE Studio compilers. 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 ONE 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.

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 7 and Solaris 8 operating environments 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 operating environment and the alternate threads library in the Solaris 8 operating environment 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_()`. In this function there is another call to `__mt_run_my_job_()`, which calls the worksharing body function, either directly in the case of a worksharing section, or 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.

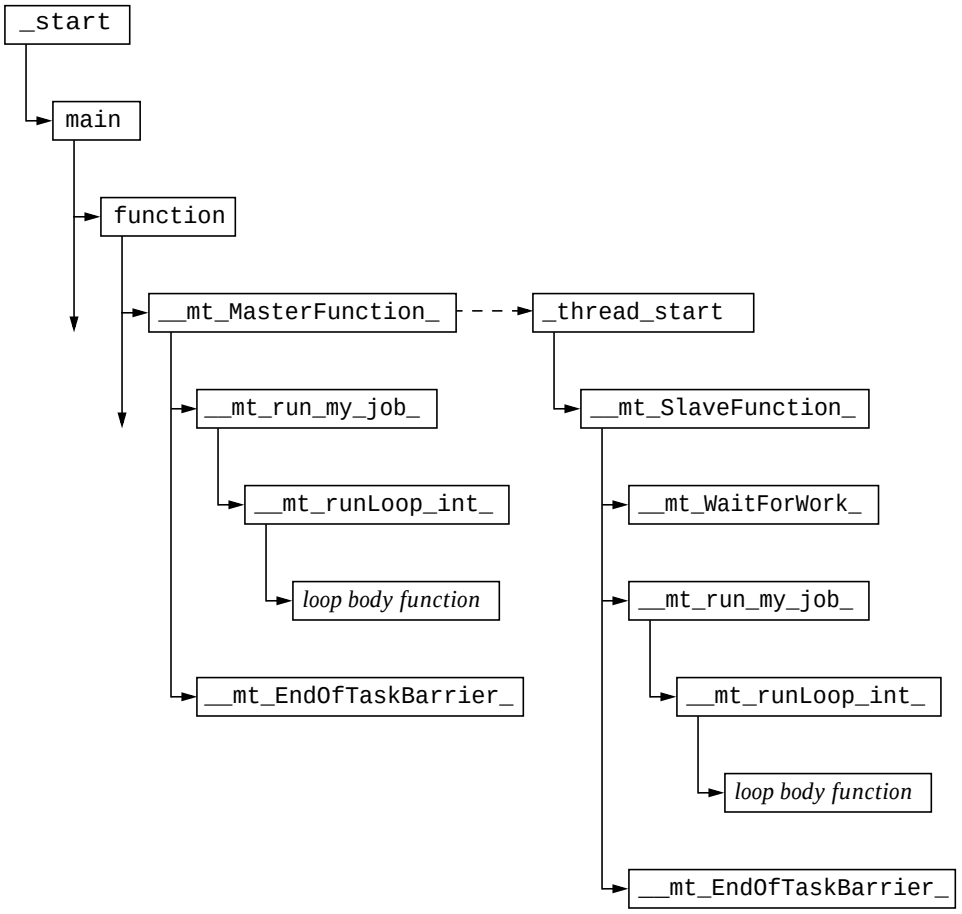


FIGURE 7-1 Schematic Call Tree for a Multithreaded Program That Contains a Parallel Do or Parallel For Construct

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.

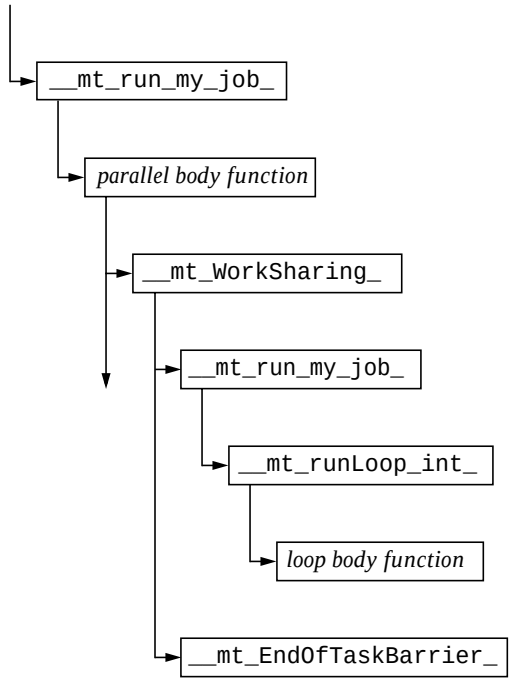


FIGURE 7-2 Schematic Call Tree for a Parallel Region With a Worksharing Do or Worksharing For Construct

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 Performance 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 100).

Worker threads typically use CPU time while they are in `__mt_WaitForWork_()` in order to reduce latency when new work arrives, that is, when the main thread reaches a new parallel construct. This is known as a busy-wait. However, you can set an environment variable to specify a sleep wait, which shows up in the Performance Analyzer as Other Wait time instead of User CPU time. There are generally two situations where the worker threads spend time waiting for work, where you might want to redesign your program to reduce the waiting:

- When the main thread is executing in a serial region and there is nothing for the worker threads to do
- When the work load is unbalanced, and some threads have finished and are waiting while others are still executing

By default, the microtasking library uses threads that are bound to LWPs. You can override this default in the Solaris 7 and 8 operating environments by setting the environment variable `MT_BIND_LWP` to `FALSE`.

Note – The multiprocessing dispatch process is implementation-dependent and might change from release to release.

Incomplete Stack Unwinds

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, and `<Total>` appears as the caller of the last function whose PC was recorded.

Mapping Addresses to Program Structure

Once a call stack is processed into PC values, the Performance Analyzer maps those PCs to shared objects, functions, source lines, and disassembly lines (instructions) in the program. This section describes those mappings.

The Process Image

When a program is run, a process is instantiated from the executable for that program. The process has a number of regions in its address space, some of which are text and represent executable instructions, and some of which are data which is not normally executed. PCs as recorded in the call stack normally correspond to addresses within one of the text segments of the program.

The first text section in a process derives from the executable itself. Others correspond to shared objects that are loaded with the executable, either at the time the process is started, or dynamically loaded by the process. The PCs in a call stack are resolved based on the executable and shared objects loaded at the time the call stack was recorded. Executables and shared objects are very similar, and are collectively referred to as load objects.

Because shared objects can be loaded and unloaded in the course of program execution, any given PC might correspond to different functions at different times during the run. In addition, different PCs might correspond to the same function, when a shared object is unloaded and then reloaded at a different address.

Load Objects and Functions

Each load object, whether an executable or a shared object, contains a text section with the instructions generated by the compiler, a data section for data, and various symbol tables. All load objects must contain an ELF symbol table, which gives the names and addresses of all the globally-known functions in that object. Load objects compiled with the `-g` option contain additional symbolic information, which can augment the ELF symbol table and provide information about functions that are not global, additional information about object modules from which the functions came, and line number information relating addresses to source lines.

The term *function* is used to describe a set of instructions that represent a high-level operation described in the source code. The term covers subroutines as used in Fortran, methods as used in C++ and Java, 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 Performance Analyzer. The name chosen is the last symbol at the given address in alphabetic order. This choice most often corresponds to the name that the user would use. In the Summary 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, there are circumstances under which multiple pieces of code have the same name:

- Sometimes, for reasons of modularity, functions are defined as static, meaning that their names are known only in some parts of the program (usually a single compiled object module). In such cases, several functions of the same name referring to quite different parts of the program appear in the Performance Analyzer. In the Summary 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 Performance Analyzer function list. Such functions come from different shared objects and different object modules, and can be distinguished from each other in that way. The Collector wraps some library functions, and both the wrapper function and the real function can appear in the Performance Analyzer.

Static Functions From Stripped Shared Libraries

Static functions are often used within libraries, so that the name used internally in a library does not conflict with a name that the user might use. When libraries are stripped, the names of static functions are deleted from the symbol table. In such cases, the Performance Analyzer generates an artificial name for each text region in the library containing stripped static functions. The name is of the form `<static>@0x12345`, where the string following the `@` sign is the offset of the text region within the library. The Performance Analyzer cannot distinguish between contiguous stripped static functions and a single such function, so two or more such functions can appear with their metrics coalesced.

Stripped static functions are shown as called from the correct caller, except when the PC from the static function is a leaf PC that appears after the save instruction in the static function. Without the symbolic information, the Performance Analyzer does not know the save address, and cannot tell whether to use the return register as the caller. It always ignores the return register. Since several functions can be coalesced into a single `<static>@0x12345` function, the real caller or callee might not be distinguished from the adjacent 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 Performance Analyzer. Call stacks representing time in Fortran subroutines with alternate entry points usually have PCs in the main body of the subroutine, rather than the prologue, and only the name associated with the main body will appear as a callee. Likewise, all calls from the subroutine are shown as being made from the name associated with the main body of the subroutine.

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 Performance Analyzer.

- C++ inline function definitions. The rationale for inlining in this case is that the cost of calling a function is much greater than the work done by the inlined function, so it is better to simply insert the code for the function at the call site, instead of setting up a call. Typically, access functions are defined to be inlined, because they often only require one instruction. When you compile with the `-g` option, inlining of functions is disabled; compilation with `-g0` permits inlining of functions.

- Explicit or automatic inlining performed by the compiler at high optimization levels (4 and 5). Explicit and automatic inlining is performed even when `-g` is turned on. The rationale for this type of inlining can be to save the cost of a function call, but more often it is to provide more instructions for which register usage and instruction scheduling can be optimized.

Both kinds of inlining have the same effect on the display of metrics. Functions that appear in the source code but have been inlined do not show up in the function list, nor do they appear as callees of the 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 210.

The Performance Analyzer shows these functions as normal functions, and assigns a name to them based on the function from which they were extracted, in addition to the compiler-generated name. Their exclusive and inclusive metrics represent the time spent in the body function. In addition, the function from which the construct was extracted shows inclusive metrics from each of the body functions. The means by which this is achieved is described in “Parallel Execution Sequence” on page 211.

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 optimization. They represent code that is not normally expected to be executed. Specifically, it is code that is not executed during the “training run” used to generate the feedback. To improve paging and instruction-cache behavior, such code is moved elsewhere in the address space, and is made into a 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 Performance 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 Performance 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.

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 109 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 Java™ virtual machine that executes the methods. In Java mode, all methods are merged with the interpreted version. In Expert-Java mode, the HotSpot-compiled and interpreted versions of each method are both shown separately. In the machine mode, 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 109 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 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 of this is an unwind failure, where the value recorded as a PC is not a PC at all, but rather some other word. If the PC is the return register, and it does not seem to be within any known load object, it is ignored, rather than attributed to the <Unknown> function.
- When a PC maps to an internal part of the Java™ virtual machine 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 <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 Java virtual machine 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 may refuse to report a callstack either to avoid deadlock, or when unwinding the Java stack would cause excessive synchronization.

The <Total> Function

The <Total> function is an artificial construct used to represent the program as a whole. All performance metrics, in addition to being attributed to the functions on the call stack, are attributed to the special function <Total>. It appears at the top of the function list and its data can be used to give perspective on the data for other functions. In the Callers-Callees list, it is shown as the nominal caller of _start() in the main thread of execution of any program, and also as the nominal caller of _thread_start() for created threads. If the stack unwind was incomplete, the <Total> function can appear as the caller of other functions.

Functions Related to HW Counter Profiling

The following functions are related to HW counter 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 (i.e., higher resolution configuration) is recommended.
- collector_record_counters: The counts accumulated while handling and recording hardware counter events, partially accounting for hardware counter profiling overhead. If this corresponds to a significant fraction of the <Total> count, a larger overflow interval (i.e., 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 Performance 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 *dataobject* 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, dataobject types and their sizes will vary. Many dataobjects are explicitly named in source programs, while others may be unnamed. Some dataobjects are derived or aggregated from other (simpler) dataobjects, resulting in a rich, often complex, set of dataobjects.

Each dataobject has an associated scope, the region of the source program where it is defined and can be referenced, which may be global (i.e., a loadobject), a particular compilation unit (i.e., object file), or function. Identical dataobjects may be defined with different scopes, or particular dataobjects 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 dataobject and propagate to any aggregates containing the dataobject and the artificial `<Total>` which is considered to contain all dataobjects (including `<Unknown>` and `<Scalars>`). The different subtypes of `<Unknown>` propagate up to the `<Unknown>` aggregate. The following section describes the `<Total>`, `<Scalars>`, and `<Unknown>` dataobjects.

Dataobject Descriptors

Dataobjects are fully described by a combination of their declared type and name. A simple scalar dataobject "`{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`". Types containing spaces have them replaced with "`_`" (underscore), and unnamed dataobjects are represented with a name of "`-`" (dash), e.g., "`{double_precision_complex -}`".

An entire aggregate is similarly represented "`{structure:foo_t foo}`" for a structure of type "`foo_t`" called "`foo`". A single element of an aggregate requires the additional specification of its container, e.g., "`{structure:foo_t foo}.{int i}`"

for a member "`i`" of type "`int`" of the previous structure "`foo`". Aggregates may 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> Dataobject

The <Total> dataobject is an artificial construct used to represent the program's dataobjects as a whole. All performance metrics, in addition to being attributed to a distinct dataobject (and any aggregate to which it belongs), are attributed to the special dataobject <Total>. It appears at the top of the dataobject list and its data can be used to give perspective to the data for other dataobjects.

The <Scalars> Dataobject

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> dataobject.

The <Unknown> Dataobject

Under various circumstances, event data can't be mapped to a particular dataobject. In such cases, the data is mapped to the special dataobject named <Unknown>.

The following circumstances result in data mapped to <Unknown>, as well as the particular subcategory:

(Unascertainable): One or more compilation objects were compiled without hardware profiling support, such that it is not possible to ascertain dataobjects associated with memory-referencing instructions or validate backtracking.

(Unverifiable): The hardware profiling support information provided in the compilation object was insufficient to verify the validity of backtracking.

(Unresolvable): Event backtracking encountered a control transfer target and, being unable to resolve whether the control transfer actually occurred or not, was unable to determine the likely causal memory-referencing instruction (and its associated dataobject).

(Unspecified): Backtracking determined the likely causal memory-referencing instruction, but its associated dataobject was not specified by the compiler.

(Unidentified): Backtracking determined the likely causal memory-referencing instruction, but it was not identified by the compiler and associated dataobject determination is therefore not possible. Compiler temporaries are generally unidentified.

Annotated Code Listings

Annotated source code and annotated disassembly code are useful for determining which source lines or instructions within a function are responsible for poor performance. This section describes the annotation process and some of the issues involved in interpreting the annotated code.

Annotated Source Code

Annotated source code shows the resource consumption of an application at the source-line level. It is produced by taking the PCs that are recorded in the application's call stack, and mapping each PC to a source line. To produce an annotated source file, the Performance Analyzer first determines all of the functions that are generated in a particular object module (`.o` file) or load object, then scans the data for all PCs from each function. In order to produce annotated source, the Performance Analyzer must be able to find and read the object module or load object to determine the mapping from PCs to source lines, and it must be able to read the source file to produce an annotated copy, which is displayed. The 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.

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 7-3.

TABLE 7-3 Annotated Source-Code Metrics

Metric	Significance
(Blank)	No PC in the program corresponds to this line of code. This case should always apply to comment lines, and applies to apparent code lines in the following circumstances: • All the instructions from the apparent piece of code have been eliminated during optimization. • The code is repeated elsewhere, and the compiler performed common subexpression recognition and tagged all the instructions with the lines for the other copy. • The compiler tagged an instruction with an incorrect line number.
`0.`	Some PCs in the program were tagged as derived from this line, but there was no data that referred to those PCs: they were never in a call stack that was sampled statistically or traced for thread-synchronization data. The `0.` metric does not indicate that the line was not executed, only that it did not show up statistically in a profiling data packet or a 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.

Compiler Commentary

Various parts of the compiler can incorporate commentary into the executable. Each comment is associated with a specific line of source code. When the annotated source is written, the compiler commentary for any source line appears immediately preceding the source line.

The compiler commentary describes many of the transformations which have been made to the source code to optimize it. These transformations include loop optimizations, parallelization, inlining and pipelining.

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.

Parallelization Directives

When the compiler generates body functions from code that contains parallelization directives, inclusive metrics for the parallel loop or section are attributed to the parallelization directive, because this line is the call site for the compiler-generated body function. Inclusive and exclusive metrics also appear on the code in the loops or sections. These metrics sum to the inclusive metrics on the parallelization directives.

Special Lines in the Source

Whenever the source line for a PC cannot be determined, the metrics for that PC are attributed to a special source line that is inserted at the top of the annotated source file. High metrics on that line indicates that part of the code from the given object module does not have line-mappings. Annotated disassembly can help you determine the instructions that do not have mappings. The special lines are:

- *Function* `<instructions without line numbers>`
  Where *function* is the name of the function from which the instruction came.
- *Function* `<source file name not recorded>`
  Where *function* is the name of the function from which the instruction came.

Annotated Disassembly Code

Annotated disassembly provides an assembly-code listing of the instructions of a function or object module, with the performance metrics associated with each instruction. Annotated disassembly can be displayed in several ways, determined by whether line-number mappings and the source file are available, and whether the object module for the function whose annotated disassembly is being requested is known:

- If the object module is not known, the Performance Analyzer disassembles the instructions for just the specified function, and does not show any source lines in the disassembly.
- If the object module is known, the disassembly covers all functions within the object module.
- If the source file is available, and line number data is recorded, the Performance Analyzer 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 7-3.

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 Performance Analyzer's algorithm for interleaving is that whenever an instruction is shown as coming from line *N*, all source lines up to and including line *N* are written before the instruction. 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 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. This means that some instructions might be underrepresented because they are always issued in the same clock cycle as the previous instruction, so they never represent the next instruction to be executed. It also means that when the call stack is recorded, there might be several instructions which could be considered the "next" instruction to execute.

Instruction issue 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 `er_print`. 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 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 "+".

Special Lines in the Disassembly and PCs Tabs

There are a number of special lines that may appear in the Disassembly and PCs tabs. This section describes such lines using examples as they might appear in the analyzer.

- `<static>@0x1a4400 + 0x000032B8`
  Represents a static function and its offset
- `<`*`library`*`.so> -- no functions found + 0x0000F870`
  As above, with offset
- *Method_Name* `<HotSpot-compiled leaf instructions>`
  Represents the HotSpot-compiled version of the named Java method
- *Method_Name* `<Java native method>`
  Represents instructions generated by a Java native method
- `<no Java callstack recorded> + 0x00000000`
  The offset is an error code from the JVM, usually 0 indicating nothing -- but no unwind
- `<branch target>`
  In disassembly, where the backtracking for event PC and its effective address fails because it runs into a branch target, and so can not back up past it.

Program Linkage Table (PLT) Instructions

When a function in one load object calls a function in a different shared object, the actual call transfers first to a three-instruction sequence in the PLT, and then to the real destination. The analyzer removes PCs that correspond to the PLT, and assigns the metrics for these PCs to the call instruction. Therefore, if a call instruction has an unexpectedly high metric, it could be due to the PLT instructions rather than the call instructions. See also “Function Calls Between Shared Objects” on page 204.

CHAPTER 8

Manipulating Experiments and Viewing Annotated Code Listings

This chapter describes the utilities which are available for use with the Collector and Performance Analyzer.

This chapter covers the following topics:

- Manipulating Experiments
- Viewing Annotated Code Listings With `er_src`
- Other Utilities

Manipulating Experiments

Experiments are stored in a 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. This is not true 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.

`er_cp [-V]` *experiment1 experiment2*

`er_cp [-V]` *experiment-list directory*

The first form of the `er_cp` command copies *experiment1* to *experiment2*. If *experiment2* exists, `er_cp` exits with an error message. The second form copies a blank-separated list of experiments to a directory. If the directory already contains an experiment with the same name as one of the experiments being copied, `er_mv` exits with an error message. The `-V` option prints the version of `er_cp`. This command does not copy experiments created with software releases earlier than the Forte Developer 7 release.

`er_mv [-V]` *experiment1 experiment2*

`er_mv [-V]` *experiment-list directory*

The first form of the `er_mv` command moves *experiment1* to *experiment2*. If *experiment2* exists, `er_mv` exits with an error message. The second form moves a blank-separated list of experiments to a directory. If the directory already contains an experiment with the same name as one of the experiments being moved, `er_mv` exits with an error message. The `-V` option prints the version of `er_mv`. This command does not move experiments created with software releases earlier than the Forte Developer 7 release.

`er_rm [-f] [-V]` *experiment-list*

Removes a list of experiments or experiment groups. When experiment groups are removed, each experiment in the group is removed then the group file is removed. The `-f` option suppresses error messages and ensures successful completion, whether or not the experiments are found. The `-V` option prints the version of `er_rm`. This command removes experiments created with software releases earlier than the Forte Developer 7 release.

Viewing Annotated Code Listings With `er_src`

Annotated source code and annotated disassembly code can be viewed using the `er_src` utility, without running an experiment. The display is generated in the same way as in the Performance Analyzer, except that it does not display any metrics. The syntax of the `er_src` command is

```
er_src [ options ] object item tag
```

object is the name of an executable, a shared object, or an object file (`.o` file).

item is the name of a function or of a source or object file used to build the executable or shared object; it can be omitted when an object file is specified.

tag is an index used to determine which *item* is being referred to when multiple functions have the same name. If it is not needed, it can be omitted. If it is needed and is omitted, a message listing the possible choices is printed.

The following sections describe the options accepted by the `er_src` utility.

`-c` *commentary-classes*

Define the compiler commentary classes to be shown. *commentary-classes* is a list of classes separated by colons. See "Commands Controlling the Source and Disassembly Listings" on page 178 for a description of these classes.

The commentary classes can be specified in a defaults file. The system wide `er.rc` defaults file is read first, then a `.er.rc` file in the user's home directory, if present, then a `.er.rc` file in the current directory. Defaults from the `.er.rc` file in your home directory override the system defaults, and defaults from the `.er.rc` file in the current directory override both home and system defaults. These files are also used by the Performance Analyzer and `er_print`, but only the settings for source and disassembly compiler commentary are used by `er_src`.

See "Default-Setting Commands" on page 188 for a description of the defaults files. Commands in a defaults file other than `scc` and `dcc` are ignored by `er_src`.

`-d`

Include the disassembly in the listing. The default listing does not include the disassembly. If there is no source available, a listing of the disassembly without compiler commentary is produced.

`-o` *filename*

Open the file *filename* for output of the listing. By default, output is written to `stdout`.

`-V`

Print the current release version.

Other Utilities

There are some other utilities that should not need to be used in normal circumstances. They are documented here for completeness, with a description of the circumstances in which it might be necessary to use them.

The `er_archive` Utility

The syntax of the er_archive command is as follows.

```
er_archive [–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` command 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, `er_archive` 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 `er_archive` 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 `er_archive` is run, or if it has a time stamp differing from that recorded in the experiment, or if `er_archive` is run on a different machine from that on which the experiment was recorded, the archive file contains a warning. Warnings are also written to `stderr` whenever `er_archive` is run manually (without the `-q` flag).

The following sections describe the options accepted by the `er_archive` utility.

-q

Do not write any warnings to `stderr`. Warnings are incorporated into the archive file, and shown in the Performance Analyzer or `er_print` output.

-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 `er_archive` 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™ operating environment. `tcov` is a code coverage tool provided with the Sun™ ONE Studio product.

Note – If you want to track how many times a function is called or how often a line of source code is executed, use the traditional profiling tools. If you want a detailed analysis of where your program is spending time, you can get more accurate information using the Collector and Performance Analyzer. See Chapter 4 and Chapter 5 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 244. In this particular case, improving the algorithm also reduces the number of calls.

Note – For Solaris 7 and 8 platforms, the profile of CPU time is accurate for programs that use multiple CPUs, but the fact that the counts are not locked may affect the accuracy of the counts for functions.

Using `gprof` to Generate a Call Graph Profile

While the flat profile from `prof` can provide valuable data for performance improvements, a more detailed analysis can be obtained by using a call graph profile to display a list identifying which modules are called by other modules, and which modules call other modules. Sometimes removing calls altogether can result in performance improvements.

Note – `gprof` attributes the time spent within a function to the callers in proportion to the number of times that each arc is traversed. Because all calls are not equivalent in performance, this behavior might lead to incorrect assumptions. See "Metric Attribution and the `gprof` Fallacy" on page 40 for an example.

Like `prof`, `gprof` generates a statistical profile of the CPU time that is used by a program and it counts the number of times that each function is entered. `gprof` also counts the number of times that each arc in the program's call graph is traversed. An *arc* is a caller-callee pair.

Note – For Solaris 7 and 8 platforms, the profile of CPU time is accurate for programs that use multiple CPUs, but the fact that the counts are not locked may affect the accuracy of the counts for functions.

To generate a profile report using `gprof`:

1. **Compile your program with the appropriate compiler option.**
   - For C programs, use the `-xpg` option.
   - For Fortran programs, use the `-pg` option.

2. **Run your program.**

   Profiling data is sent to a profile file called `gmon.out`. This file is overwritten each time you run the program.

3. **Run `gprof` to generate a profile report.**

   The syntax of the `prof` command is as follows.

```
% gprof program-name
```

   Here, *program-name* is the name of the executable. The profile report is written to `stdout`, and can be large. The report consists of two major items:

   - The full call graph profile, which shows information about the callers and callees of each 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 255).

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

D

E

F

G

H

I

J

K

L

M

R

S

T

U

V

W