Analyzing Program Performance With Sun WorkShop

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

Part No. 806-3562-10
May 2000, Revision A

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

Please
Recycle

Adobe PostScript

Important Note on New Product Names

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

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

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

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

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

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

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

Contents

Figures

Tables

Preface

This manual describes the performance and analysis tools available with Sun WorkShop™. Developing high performance applications requires a combination of compiler features, libraries of optimized routines, and tools to analyze and isolate code. *Analyzing Program Performance With Sun WorkShop* describes the third part of this development strategy, and shows you how to use these tools:

- The Sampling Analyzer and Sampling Collector
- LoopTool and LoopReport
- `prof`, `gprof`, and `tcov`

Who Should Use This Book

This manual is intended for programmers with a working knowledge of Sun WorkShop and some understanding of the Solaris™ operating environment and UNIX® commands. Some knowledge of performance analysis is also helpful in understanding how to use the data that is derived from the tools, but is not required for using the tools. The traditional profiling tools `prof`, `gprof`, and `tcov` do not require a working knowledge of Sun WorkShop.

Multiplatform Release

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

Note – The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture, whereas "*Intel Platform Edition*" appears in the product name.

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your `PATH` environment variable:

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

```
% echo $PATH
```

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

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

To determine if you need to set your `MANPATH` environment variable:

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

```
% man workshop
```

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

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

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the `/opt` directory. Contact your system administrator if your Sun WorkShop software is not installed in `/opt`.

The `PATH` and `MANPATH` variables should be set in your home `.cshrc` file if you are using the C shell or in your home `.profile` file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your `PATH` variable:

  ```
  /opt/SUNWspro/bin
  ```

- To access Sun WorkShop man pages with the `man` command, add the following to your `MANPATH` variable:

  ```
  /opt/SUNWspro/man
  ```

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

How This Book Is Organized

Chapter 1, "Overview of Performance Profiling and Analysis Tools," introduces the performance analysis tools, briefly discussing what they do and when to use them.

Chapter 2, "Tutorial: Using the Sampling Collector and Analyzer," is a tutorial demonstrating how to use the Sampling Collector and Analyzer to fine-tune the performance of three sample programs.

Chapter 3, "Sampling Collector Reference," describes how to use the Sampling Collector to collect information about program execution.

Chapter 4, "Sampling Analyzer Reference," describes how to use the Sampling Analyzer to fine-tune program performance.

Chapter 5, "`er_print` Reference," describes how to use the `er_print` utility.

Chapter 6, "Advanced Topics: Understanding the Sampling Analyzer and Its Data," discusses advanced issues involving the impact of optimization on data that is displayed by the Sampling Analyzer.

Chapter 7, "Loop Analysis Tools," presents LoopReport and LoopTool, which help you analyze program loops that have been parallelized by your compiler.

Appendix A, "Traditional Profiling Tools," covers the traditional profiling tools `prof`, `gprof`, and `tcov`. These tools help you find the parts of your program that are most heavily used, and determine how much of your program is being tested.

Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

TABLE P-1 Typographic Conventions

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

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

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

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the `docs.sun.com`[sm] Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:

  `http://docs.sun.com`

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:
  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:

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

  (Contact your system administrator if your Sun WorkShop software is not installed in the `/opt` directory.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document's title.

TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection

Document Collection	Document Title	Description
Forte™ Developer 6 / Sun WorkShop 6 Release Documents	*About Sun WorkShop 6 Documentation*	Describes the documentation available with this Sun WorkShop release and how to access it.
	What's New in Sun WorkShop 6	Provides information about the new features in the current and previous release of Sun WorkShop.
	Sun WorkShop 6 Release Notes	Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.
Forte Developer 6 / Sun WorkShop 6	*Debugging a Program With dbx*	Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.
	Introduction to Sun WorkShop	Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.
Forte™ C 6 / Sun WorkShop 6 Compilers C	*C User's Guide*	Describes the C compiler options, Sun-specific capabilities such as pragmas, the `lint` tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.
Forte™ C++ 6 / Sun WorkShop 6 Compilers C++	*C++ Library Reference*	Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
	C++ Migration Guide	Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.
	C++ Programming Guide	Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.
	C++ User's Guide	Provides information on command-line options and how to use the compiler.
	Sun WorkShop Memory Monitor User's Manual	Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see `/opt/SUNWspro/docs/index.html`) and not at the `docs.sun.com` Web site.
Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	*Fortran Library Reference*	Provides details about the library routines supplied with the Fortran compiler.
	Fortran Programming Guide	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran User's Guide	Provides information on command-line options and how to use the compilers.
	FORTRAN 77 Language Reference	Provides a complete language reference.
	Interval Arithmetic Programming Reference	Describes the intrinsic `INTERVAL` data type supported by the Fortran 95 compiler.
Standard Library 2	*Standard C++ Class Library Reference*	Provides details on the Standard C++ Library.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
	Standard C++ Library User's Guide	Describes how to use the Standard C++ Library.
Tools.h++ 7	*Tools.h++ User's Guide*	Discusses use of the C++ classes for enhancing the efficiency of your programs.
	Tools.h++ Class Library Reference	Provides details on the `Tools.h++` class library.

TABLE P-4 describes related Solaris documentation available through the `docs.sun.com` Web site.

TABLE P-4 Related Solaris Documentation

Document Collection	Document Title	Description
Solaris Software Developer	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.
	Programming Utilities Guide	Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.

CHAPTER 1

Overview of Performance Profiling and Analysis Tools

Developing high performance applications requires a combination of compiler features, libraries of optimized routines, and tools that you use to analyze and isolate code. *Analyzing Program Performance With Sun WorkShop* describes the tools that are available to help you achieve the goal of isolating and analyzing your code.

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

- The Sampling Collector collects performance data (statistical profiles of call stacks, thread-synchronization delay events, hardware-counter overflow profiles, address space data, and summary information for the operating system), and stores it in an experiment file. See Chapter 3 for detailed information about the Sampling Collector.
- The Sampling Analyzer displays the data recorded by the Sampling Collector, so you can examine the information. The Analyzer processes the data and displays various metrics of performance at function, caller-callee, source-line, disassembly-instruction, and program levels. See Chapter 4 for detailed information about the Sa mpling Analyzer.

  The Sampling Analyzer can also help you to fine-tune your application's performance by creating a mapfile that you can use to improve the order of function loading in the application address space.

The Collector and Analyzer are designed for use by any software developer, even if performance tuning is not the developer's main responsibility.

Command-line equivalents of the Collector and Analyzer are available:

- `dbx` includes a data-collection feature that has the same functionality as the Collector. See "Starting a Process Under the Collector in `dbx`" on page 43.
- The command-line utility `er_print`, which prints out an ASCII version of the various Analyzer displays, operates as a command-line sampling analyzer. See Chapter 5 for more information.

The Sampling Collector and Sampling Analyzer are included in the following Sun Workshop™ products:

- Sun WorkShop Professional™ C
- Sun Visual WorkShop™ C++
- Sun Performance WorkShop™M Fortran
- Sun WorkShop™ University Edition

This manual also includes information about the following performance tools:

- LoopTool and LoopReport

  LoopTool is a loop analysis tool that supports performance tuning of automatically parallelized programs. LoopReport is the command line version of LoopTool. See Chapter 7 for more information.

- `prof`, `gprof`, and `tcov`

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

  `tcov` is a code coverage tool. It is included in Sun Workshop.

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

CHAPTER 2

Tutorial: Using the Sampling Collector and Analyzer

This chapter demonstrates how to use the Sampling Collector and the Sampling Analyzer to understand the performance of three sample programs:

- Example 1: `synprog`. A simple program that demonstrates various programming constructs, as well as an example of the `gprof` fallacy
- Example 2: `omptest`. A program that uses OpenMP parallelization in Fortran
- Example 3: `mttest`. A program that uses explicit multithreading

In these examples, each of the programs is subjected to the same sequence of performance questions:

- What can I change in my program to improve its performance?

  Program analysis at this level is concerned with higher-level algorithm issues. The compiled code is already as optimized as the compiler can make it; you are now looking for ways to refine the program algorithm itself to make it execute more efficiently.

- What resources is my program using?

  For example, how much CPU time is my program using?

- Where in my program are most of these resources being used?

  For example, is there a particular function where my program is spending most of its time? Which line or instruction in the function accounts for most of the time being spent?

- How did the program arrive at that line or instruction in the execution process?

Once you determine where your program is using the most resources, the Analyzer offers various ways of examining your code so you can determine why.

Note – In all these examples, the times shown represent data collected in particular experiments. When you record data, you will see different numbers, because your platform is probably different from the one used to collect the data reported here.

The source code for each of these demonstration programs is included on the distribution. Before you start the tutorial:

1. **Make sure the Sun WorkShop directory has been added to your path:**

```
/opt/SUNWspro/bin
```

2. **Copy the files in one or more of the demo directories to a work area of your own, and do a make to construct the demo program.**

Example 1: synprog

The `synprog` program is a simple program that demonstrates a number of programming constructs, each of which exhibits some interesting performance characteristic: simple Metric analysis, recursion, loading and unloading dynamically linked shared objects. You can use `synprog` to exercise the Collector and Analyzer.

Among other things, `synprog` demonstrates the so-called "`gprof` fallacy". `gprof` is a standard UNIX performance tool that in this case properly identifies the function where the program is spending most of its CPU time (exclusive time, that is, time spent in the function itself), but wrongly reports the caller responsible for most of that time.

Copying synprog

The Sun WorkShop installation program installs the `synprog` source files in the following directory.

```
/installation_directory/SUNWspro/examples/analyzer/synprog
```

In a default installation, *installation_directory* is `/opt`.

Before you start on this part of the tutorial, create a working directory and copy the `synprog` source files and the makefile into it:

```
cp -r installation_directory ~/synprog
```

Building synprog

Before you build the synprog program:

- **Open the makefile in a text editor:**

  The makefile contains alternative settings for the environment variables ARCH and OFLAGS.

  - You can accept the default setting for ARCH, which works on SPARC 7, 8, or 9 platforms, and x86 platforms.

Note – The default architecture (the -xarch compiler flag) is v7 for SPARC, to accommodate older machines, and ia32 for Intel. Most newer SPARC machines support v8. If your machine supports v8, comment out the default and uncomment the line ARCH = -xarch=v8 or, on a v9 machine, ARCH = -xarch=v9. Using the default generates code that calls the libc.so .mul and .div routines rather than using integer multiply and divide instructions, and the time spent in these arithmetic operations shows up in the <Unknown> function; see "The <Unknown> Function" on page 104 for more information.

  - The makefile offers you a couple of settings for OFLAGS that affect program optimization. You can build synprog with the default, which uses the command-line options -g -xF -v -V, run the Collector on it, and look at the results in the Analyzer. Then you can repeat the process with the other setting to get an idea of how the different settings affect how the compiler optimizes code. For an explanation of the various compiler options in the OFLAGS settings, see the *C User's Guide*.

Note – The sample program described in this section is compiled with no optimizations.

To build synprog:

1. **Save the makefile and close the editor.**
2. **Type make at the command prompt.**

Collecting Data About `synprog`

To collect performance data on `synprog`:

1. **Start Sun WorkShop by typing:**

```
% workshop
```

2. **Click the Debugging button to open the Debugging window.**

3. **Load `synprog` into the Debugging window:**

   a. **Choose Debug ➤ New Program from the main Debugging menu.**

   b. **In the Debug New Program dialog box, enter the path for `synprog`'s in the Name text box, or use the list box to navigate to `synprog`.**

   c. **Click OK.**

4. **From the Debugging window menu bar, choose Windows ➤ Sampling Collector to open the Sampling Collector window.**

   Observe the following points:

   - The Collector is enabled for one run only.
   - The default path and file name for the experiment-record file appear in the Experiment File text box.
   - Under the Data to Collect check boxes, the only type of data selected for collection is the default, Clock-Based Profiling.

   For this experiment you are going to collect only clock-based profiling data and use the default sampling interval, so use the default settings.

5. **Click the Start button:**

   `synprog` runs in the Debugging window, and the Collector collects clock-based profiling data and stores it in the default experiment-record file, `test.1.er`.

Analyzing `synprog` Performance Metrics

To open the Analyzer and load `test.1.er` into it:

1. **Click on the Analyzer button in either the Sun WorkShop main window tool bar or the Sampling Collector window.**

2. **In the Load Experiment dialog box, type `test.1.er`, then click on OK.**

To start the Analyzer and load `test.1.er` from the command line:

- **Type:**

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

The Analyzer window displays the function list for `synprog`. The function list displays the default clock-based profiling metrics:

- 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.

Simple Metric Analysis

First, look at execution times for two very simple functions, `cputime()` and `icputime()`. Both contain a `for` loop that increments a variable `x` by one, but in `cputime()`, `x` is floating-point, while in `icputime()`, `x` is an integer.

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

You can see immediately by looking at the exclusive user CPU time for the two functions that `cputime()` takes much longer to execute than `icputime()`. You can use other features of the Analyzer to find out why this is so.

2. **In the Function List display, click on `cputime()` to select it.**

3. **Click Source in the lower tool bar.**

   A text editor opens, displaying the annotated source code for function `cputime()`. (You may need to widen the text editor window.)

   You can see that most of the execution time is used by the loop line and the line in which `x` is incremented:

```
for(j=0; j<1000000; j++) {
        x = x + 1.0;
}
```

4. **Select `icputime()` in the Function List display and click Source.**

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

```
for(j=0; j<1000000; j++) {
        x = x + 1;
}
```

   You can see that this line takes much less time to execute than the corresponding line in `cputime()`. (The time on the loop line is about the same as it was for function `cputime()`.)

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

5. **Select `cputime()` in the Function List display, and click Disassembly in the lower tool bar.**

   The annotated disassembly for `cputime()` appears in the text editor. Scroll down until you see some non-zero metric values. Look at the ones for the line of source code in which x is incremented.

   You can see that while a small amount of time is spent in loading and adding the values for `x` and 1, the greatest amount of time is spent executing the `fstod` instruction, which converts the value of `x` from a single floating-point value to a double floating-point value, so it can be incremented by 1, a double. Afterwards, another small amount of CPU time is spent by the `fdtos` command, which converts `x` back to a single. Altogether, these two operations account for about three-quarters of the CPU time expended.

6. **Now select `icputime()` in the Function List display and click Disassembly.**

   The annotated disassembly for `icputime()` appears in the text editor. Scroll down until you see the performance metrics for the line of source code where `x` is incremented.

   You can see that all that is involved is a simple load, add, and store operation that takes approximately a third of the time of the floating-point add, because no conversions are necessary. The value 1 does not even need to be loaded into a register—it can be added directly to `x`.

The Effects of Recursion

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 which point it performs some work that requires user CPU time, after which 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, and if it is not, calls function `bounce_b()`. `bounce_b()` in turn calls `bounce_a()`. This sequence continues until the test condition in `bounce_a()` is met, at which time `bounce_a()` performs some work that requires user CPU time. After that, 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, look at the metrics for `recurse()` and `real_recurse()`:

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

   Notice that function `recurse()` shows inclusive user CPU time, but its exclusive user CPU time is 0 (zero). This is because all `recurse()` does is execute a call to `real_recurse()`.

Note – For some runs, `recurse()` may show a small exclusive CPU time value. Because profile experiments are statistical in nature, the experiment you ran on synprog may record one or two ticks in the `recurse()` function. However, the exclusive time is much less than the inclusive time.

---

2. **Click Callers-Callees to open the Callers-Callees window, which shows that `recurse()` calls one function, `real_recurse()`.**

3. **Click on `real_recurse()` to select it.**

   The Callers-Callees window now displays the following information:

   - Both `recurse()` and `real_recurse()` appear in the callers pane as callers of `real_recurse()`. You would expect this, because after `recurse()` calls `real_recurse()`, `real_recurse()` calls itself recursively.
   - In order to simplify the display, `real_recurse()` does not appear in the callee pane as its own callee.
   - Exclusive metrics as well as inclusive metrics are recorded for `real_recurse()`, where the actual user CPU time is spent. These are passed back up to `recurse()` as inclusive metrics.
   - Exclusive metrics are also displayed for `real_recurse()` in the caller pane. If a function generates exclusive metrics, the Analyzer displays them for that function anywhere it appears in the Callers-Callees window.

   Now look at what happens in the indirect recursive sequence:

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

   Notice that function `bounce()` shows inclusive user CPU time, but its exclusive user CPU time is 0 (zero). This is because all `bounce()` does is execute a call to `bounce_a()`.

2. **Click Callers-Callees to open the Callers-Callees window, which shows that `bounce()` calls only one function, `bounce_a()`.**

3. **Click on `bounce_a()` to select it.**

   The Callers-Callees window now displays the following information:

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

4. **Click on `bounce_b()` to select it.**

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

The gprof Fallacy

Now look at how the Analyzer resolves the gprof fallacy in synprog.

Select the function gpf_work(). This is one of the functions in which synprog is spending most of its time.

In this example, however, it is less helpful to see where the program is spending time than to figure out why. To do so, look at the functions that call gpf_work() and how these functions call it:

- **Click the Callers-Callees button in the lower Analyzer tool bar to open the Callers-Callees window.**

  The Callers-Callees window is divided into three horizontal panes:

  - The middle pane shows data pertaining to the selected function, in this case gpf_work().
  - The top pane shows data pertaining to all functions that call the selected function, in this case gpf_b() and gpf_a().
  - The bottom pane shows data pertaining to all functions called by the selected function. In this case, the bottom pane is empty, because gpf_work() does not call any other functions.

  By looking at the Callers pane, you can see that gpf_work is called by two functions, gpf_b() and gpf_a(). The Analyzer shows that most of the time in gpf_work() results from calls from gpf_b(), whereas much less time results from calls from gpf_a(). You must look at the callers, to see why gpf_b() calls account for over ten times as much time in gpf_work() as calls from gpf_a():

1. **Click on gpf_a() in the Callers pane to select it.**

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

2. **Go back to the main Analyzer window (where gpf_a() is now the selected function), and click the Source button to open a text editor displaying the annotated source code for gpf_a().**

3. **In the text editor, scroll down so that you can see the code for both gpf_a() and gpf_b().**

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

   Now look at the code for gpf_work(), to see why the way the callers call gpf_work() makes a difference:

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

  From the line `imax = 4 * amt * amt`, which sets the upper limit for the following `for` loop, you can see that time spent in `gpf_work()` depends on the square of its argument. So ten times as much time, more or less, will be spent on one call from a function with an argument of 10 (400 iterations) than will be spent on ten calls from an function with an argument of 1 (10 instances of 4 iterations).

  And what has all this to do with `gprof`? The "`gprof` fallacy" is the result of `gprof` estimating the amount of time spent in a function from the number of times the function is called, regardless of how the function might be using the arguments passed to it. So for an analysis of `synprog`, `gprof` would attribute ten times as much time to calls from `gpf_a()` as it would to calls from `gpf_b()`. That is the `gprof` fallacy.

  The distortion increases the higher the power of the argument on which the CPU time in the function depends. For example, matrix multiply, whose work goes as the cube of the order, would be even more sensitive.

Loading Dynamically Linked Shared Objects

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 it was loaded at first is 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 look at their source code. Therefore they should take the same amount of user CPU time to execute.

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

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

1. **In the Function List display, click on `sx_burncpu()` to select it.**
2. **Choose View ➤ Show summary metrics. The Summary Metrics window for `sx_burncpu()` appears.**

   Note the user CPU time for `sx_burncpu()`.

3. **Click on `so_burncpu()` to select it. The summary data for `so_burncpu()` appears in the Summary Metrics window.**

   You can see that although `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()`. This is because `so_burncpu()` was executed twice; the Analyzer recognized that the same function was executing and aggregated the data for it, even though it appeared at two different addresses in the course of program execution.

Example 2: `omptest`

Note – `omptest` is a Fortran program that contains OpenMP directives and compiles only with a Fortran 95 compiler on SPARC platforms.

The `omptest` program uses OpenMP parallelization and is designed to test the efficiency of parallel distribution. Analysis of `omptest` deals with the following questions:

- What are the obstacles to parallelization?
- How is load balanced across threads?
- What are the costs of memory and bus contention?

Note – This example assumes that you are running `omptest` on a machine with at least four CPUs.

Copying `omptest`

The Sun WorkShop installation program installs the `omptest` source files in the following directory.

```
/installation_directory/SUNWspro/examples/analyzer/omptest
```

In a default installation, *installation_directory* is `/opt`.

Before you start on this part of the tutorial, create a working directory and copy the `omptest` source files and the makefile into it:

```
% cp -r installation_directory ~/omptest
```

Building `omptest`

Before you build the `omptest` program:

- **Open the makefile in a text editor.**

  The makefile contains alternative settings for the environment variables `ARCH` and `OFLAGS`.

  - You can accept the default setting for `ARCH`, which works on SPARC 7, 8, or 9 platforms.

Note – The default architecture for the `-xarch` compiler flag is v7 for SPARC, to accommodate older machines. The default setting in the `omptest` makefile is `ARCH = -xarch=v8`, since most newer SPARC machines support v8; furthermore, using the v7 default generates code that calls the `libc.so .mul` and `.div` routines rather than using integer multiply and divide instructions, and the time spent in these arithmetic operations shows up in the `<Unknown>` function; see "The `<Unknown>` Function" on page 104 for more information.

  - The makefile offers you several settings for `OFLAGS` that affect optimization and parallelization. You can build `omptest` with the default, which uses the command-line options `-g -O3 -mp=openmp -explicitpar -depend -stackvar -loopinfo -v -V`, run the Collector on it, and look at the results in the Analyzer, then repeat the process with the other settings to get an idea of how the different settings affect how the compiler optimizes and parallelizes code. For an explanation of the various compiler options in the `OFLAGS` settings, see the *Fortran User's Guide*.

Note – If you do not specify parallelization (`-mp=openmp -explicitpar`), the compiler does not interpret OpenMP directives, and compiles the program to execute serially.

To build `omptest`:

1. **Save the makefile and close the editor.**
2. **Type `make` at the command prompt.**

Collecting Data About `omptest`

For this part of the demonstration, you must have `omptest` running on a four-CPU system or larger. Sun WorkShop must be installed on the machine, so you can run the Collector to collect performance data.

Note – You might want to refer to the chapters on parallelization and OpenMP in the *Fortran Programming Guide* for background on parallelization strategies and OpenMP directives.

---

1. **Start Sun WorkShop by typing:**

```
% workshop
```

2. **Click the Debugging button to open the Debugging window.**

3. **Load `omptest` into the Debugging window:**

   a. **Choose Debug ➤ New Program from the main Debugging menu bar.**

   b. **In the Debug New Program dialog box, enter the path for `omptest` in the Name text box, or use the list box to navigate to `omptest`.**

   c. **Click OK.**

4. **Set the value of the `PARALLEL` environment variable to `4`:**

   a. **Choose Debug ➤ Edit Run Parameters to open the Run Parameters dialog box.**

   b. **Click Environment Variables to open the Environment Variables dialog box.**

   c. **Type `PARALLEL` in the Name text field and `4` in the Value text field.**

   d. **Click OK to exit the Environment Variables dialog box and the Run Parameters dialog box.**

5. **From the Debugging window menu bar, choose Windows ➤ Sampling Collector to open the Sampling Collector window.**

   Observe the following points:

   - The Collector is enabled for one run only.
   - The default path and file name for the experiment-record file appear in the Experiment File text box. Change the name of the experiment-record file to `omptest.1.er`.

- Under the Data to Collect check boxes, the only type of data selected for collection is the default, Clock-Based Profiling.

  For this experiment you are going to collect only clock-based profiling data and use the default sampling interval, so use the default settings.

6. **Click the Start button.**

`omptest` runs in the Debugging window, and the Collector collects clock-based profiling data and stores it in the default experiment-record file, `omptest.1.er`.

7. **Repeat Step 4 through Step 6 with `PARALLEL` set to 2 and save the performance information in the experiment-record file `omptest.2.er`.**

Analyzing `omptest` Performance Metrics

For purposes of analyzing parallelization strategies, we will be looking at four functions and comparing how they behave on the four-CPU run versus the two-CPU run:

- `psec()_` is a `PARALLEL SECTION` routine
- `pdo_()` is a `PARALLEL DO` routine
- `critsum_()` is a `CRITICAL SECTION` routine
- `redsum_()` is a `REDUCTION` routine

All these routines were generated at compile time as a result of OpenMP directives inserted into `omptest` source code. Each pair (`psec_()` and `pdo_()`, `critsum_()` and `redsum_()`) represents contrasting strategies for dealing with effective parallelization.

Note – Behavior of parallel code can be unexpected. Read "Parallel Execution and Compiler-Generated Body Functions" on page 96 and "Compiler-Generated Body Functions" on page 103 for more information on how this code behaves.

Look first at the performance of `psec_()` and `pdo_()` on the four-CPU and the two-CPU runs:

- **Open two Analyzer windows, with `omptest.1.er` loaded into one, and `omptest.2.er` loaded into the other. Position the two windows so you can compare the metrics displayed in them.**

In each of the Analyzer windows:

1. **In the File List display, find the data for `psec_()` and click on the line to select it.**

2. **Choose View ➤ Show Summary Metrics.**

   The Summary Metrics window appears, displaying the metrics for `psec_()`.

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

   For the two-CPU run, `psec_()` uses about 8.9 seconds user CPU time, about 4.8 seconds wall-clock time, and about 9.6 seconds total LWP. 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, however, `psec_()` takes about the same user CPU time (8.4 seconds), but both the wall-clock time and the total LWP are higher (6.12 seconds and 11.55 seconds respectively). `psec_()` takes longer to run on the four-CPU machine. 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 slightly poorer performance on the four-CPU machine is due to overhead involved in scheduling the threads among the four CPUs.

4. **Scroll to the function `pdo_()` and click on the line to select it.**

5. **In the main Analyzer menu, choose View ➤ Show Summary Metrics.**

   The Summary Metrics window appears, displaying the metrics for `pdo_()`.

6. **Look at 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_()` (about 8.4 seconds for the two-CPU run, about 8.6 seconds for the four-CPU run). But now the ratio of wall-clock time to user CPU time is about 1 to 2 on the two-CPU run, but 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.

   Now look at the relative efficiency of the routines `critsec_()` and `reduc_()`. In this case, the annotated source code shows how efficiently each parallelization strategy deals with an identical assignment statement embedded in a pair of `do` loops; its purpose is to sum the contents of three two-dimensional arrays:

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

1. **In the Function List display, find the data for `critsum_()` and click on the line to select it.**

2. **Click Source in the lower Analyzer tool bar.**

   The annotated source code from which `critsum_()` was generated appears in a text editor.

3. **Look at the inclusive user CPU time.**

   It is enormous—almost 13 seconds. The inclusive user CPU time is so high 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 strategy for making use of parallelization.

4. **Close the text editor and scroll to the data for `redsum_()`. Click on the line to select it.**

5. **Click Source in the lower Analyzer tool bar.**

   The annotated source code from which `redsum_()` was generated appears in a text editor.

6. **Look at the inclusive user CPU time.**

   It is much smaller, only about 1.7 seconds, because `redsum_()` uses a reduction strategy, by which the partial sums of `(a(j,i)+b(j,i)+c(j,i))/k` are distributed over multiple processors, after which these intermediate values are added to `sum`. This strategy makes much more efficient use of the available CPUs.

Example 3: `mttest`

The `mttest` program emulates the server in a client-server, where clients queue up 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.

Copying `mttest`

The Sun WorkShop installation program installs the `mttest` source files in the following directory.

```
/installation_directory/SUNWspro/examples/analyzer/mttest
```

In a default installation, *installation_directory* is `/opt`.

Before you start on this part of the tutorial, create a working directory and copy the `mttest` source files and the makefile into it:

```
% cp -r installation_directory ~/mttest
```

Building `mttest`

Before you build the `mttest` program:

- **Open the makefile in a text editor:**

  The makefile contains alternative settings for the environment variables `ARCH`, `OFLAGS`, `THREADS`, and `FLAG`.

  - You can accept the default setting for `ARCH`, which works on SPARC 7, 8, or 9 platforms, and x86 platforms.

Note – The default architecture (the `-xarch` compiler flag) is v7 for SPARC, to accommodate older machines, and ia32 for Intel. Most newer SPARC machines support v8. If your machine supports v8, comment out the default and uncomment the line `ARCH = -xarch=v8` or, on a v9 machine, `ARCH = -xarch=v9`. Using the default generates code that calls the `libc.so .mul` and `.div` routines rather than using integer multiply and divide instructions, and the time spent in these arithmetic operations shows up in the `<Unknown>` function; see "The `<Unknown>` Function" on page 104 for more information.

  - The makefile offers you several settings for `OFLAGS` that affect optimization and parallelization. You can build `omptest` with the default, which uses the command-line options `-g -xF -v -V`, run the Collector on it, and look at the results in the Analyzer, then repeat the process with the other settings to get an idea of how the different settings affect how the compiler optimizes and parallelizes code. For an explanation of the various compiler options in the `OFLAGS` settings, see the *Fortran User's Guide*.
  - For `THREADS` and `FLAG`, comment whichever setting is closest to your target application, and comment out the other one. The settings for `THREADS` are `SOLARIS` and `POSIX`, which determine which threads standard your program is compiled for. The settings for `FLAG` are `BOUND` and `UNBOUND`; they determine whether your program uses bound or unbound threads.

To build `mttest`:

1. **Save the makefile and close the editor.**
2. **Type `make` at the command prompt.**

Collecting and Analyzing Data About `mttest`

The executable mttest you have just built 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 if you run it on a single CPU system.

The following first two sections, "Collecting Data About `mttest` (Four-CPU System)" and "Analyzing `mttest` Performance Metrics (Four-CPU System)" on page 29, look at `mttest`'s performance on a four-CPU machine; the next two sections, "Collecting Data About `mttest` (One-CPU System)" on page 31 and "Analyzing `mttest` Performance Metrics (One-CPU System)" on page 32, look at the same set of performance metrics on a one-CPU machine.

Collecting Data About `mttest` (Four-CPU System)

For this part of the demonstration, you must have `mttest` running on a four-CPU system. Sun Workshop must be installed on the machine, so you can run the Collector to collect performance data.

1. **Start Sun WorkShop by typing:**

```
% workshop
```

2. **Click the Debugging button to open the Debugging window.**

3. **Load `mttest` into the Debugging window:**

   a. **Choose Debug ➤ New Program from the main Debugging menu.**

   b. **In the Debug New Program dialog box, enter the `mttest` path in the Name text box, or use the list box to navigate to `mttest`.**

   c. **Click OK.**

4. **From the main Debugging window menu, choose Windows ➤ Sampling Collector to open the WorkShop Sampling Collector window.**

   Observe the following points:

   - The Collector is enabled for one run only.
   - The default path and file name for the experiment-record file appear in the Experiment File text box. Change the name of the experiment-record file to `mttest.1.er`.

- Under the Data to Collect check boxes, the only type of data selected for collection is the default, Clock-Based Profiling.

5. **For this experiment, in addition to clock-based profiling, you will collect synchronization wait tracing information, so select the Synchronization Wait Tracing check box.**

6. **Click the Start button.**

`mttest` runs in the Debugging window, and the Collector collects clock-based profiling and synchronization wait tracing data and stores it in the experiment-record file, `mttest.1.er`.

Analyzing `mttest` Performance Metrics (Four-CPU System)

To open the Analyzer and load `test.1.er` into it:

1. **Click on the Analyzer button in either the Sun WorkShop main window tool bar or the Sampling Collector window.**

2. **In the Load Experiment dialog box, type `test.1.er`, then click on OK.**

To open the Analyzer and load `test.1.er` from the command line:

- **Type:**

```
% analyzer mttest.1.er
```

The Analyzer window displays the Function List for `mttest`.

1. **In the File List display, scroll down to the data for `lock_local()` and `lock_global()`.**

Note that both functions have the same inclusive user CPU time (about 3.2 seconds). This indicates that both functions are doing the same amount of work.

However, `lock_global()` spends a lot of time in synchronization waiting (almost five seconds), whereas `lock_local()` spends no time in synchronization waiting. If you bring up the annotated source code for the two functions, you can see why this is so.

2. **Click on the line of data for `lock_global()` to select it.**

3. **Click the Source button in the lower Analyzer tool bar.**

   The annotated source code for function `lock_global()` appears in the text editor. You can see that `lock_global()` applies a global lock to the data:

   ```
   mutex_lock(&global_lock);
   ```

   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.

4. **Click on the line of data for `lock_local()` to select it.**

5. **Click the Source button in the lower tool bar.**

   The annotated source code for function `lock_local()` appears in the text editor. `lock_local()` applies a lock only to data in a particular thread's work block:

   ```
   mutex_lock(&(array->lock));
   ```

   No thread can have access to another thread's work block, so their work can proceed without contention or time wasted in synchronization waits, and the wait time for `lock_local()` is 0 (zero) seconds.

6. **Return to the Function List display in the main Analyzer window and scroll to the data for the functions `ComputeA()` and `ComputeB()`.**

7. **Click on the line of data for `ComputeA()` to select it.**

8. **Click the Source button in the lower Analyzer tool bar.**

   The text editor opens, displaying the annotated source code for `ComputeA()`.

9. **Scroll down in the text editor so that the source for `ComputeA()` and `ComputeB()` is displayed.**

   Observe that although the code for these functions is virtually identical (a loop adding one to a variable), `ComputeB()` uses almost ten seconds of inclusive user CPU time, whereas `ComputeA()` uses only about 3.3 seconds. To find out the reason for this discrepancy, you must examine the code that calls `ComputeA()` and `ComputeB()`.

10. **Return to the Function List display in the main Analyzer window, and click on the line of data for `ComputeA()` to select it.**

11. **Click the Callers-Callees button in the lower tool bar.**

    The Callers-Callees window opens, showing the selected function in the middle display pane, and its caller in the upper pane.

12. **Click on caller `lock_none()` to select it.**

13. **Return to the Function List display in the main Analyzer window.**

    Note that `lock_none()` is now the selected function.

14. **Click the Source button in the lower Analyzer tool bar.**

    The text editor opens, displaying the annotated source code for `lock_none()`.

15. **Scroll down to display the code for the function which calls `ComputeB()`, which is `cache_trash()`.**

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

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

    `ComputeB()`, however, is called with a series of doubles that occupy successive words in memory (`&element[array->index]`). Whenever a thread writes to one of these addresses in memory, any other threads that have that address in their cache must delete the data, which is now out-of-date. If one of the threads needs the data again later in the program, the data must be copied back into the data cache from memory, which is a time-consuming operation. The resulting cache misses, which are attempts to access data not available in the data cache, waste a lot of CPU time, which explains why `ComputeB()` uses three times as much user CPU time as `ComputeA()`.

17. **Return to the Function List display in the main Analyzer window. Click on the line for `ComputeB()` to select it.**

18. **Click on the Disassembly button in the lower tool bar.**

    The text editor opens, displaying annotated disassembly code for `ComputeB()`. You can see that most of the user CPU time (over 7 seconds) is being used by the `faddd` instruction, which is waiting for a register load that misses in the cache.

Collecting Data About `mttest` (One-CPU System)

For this part of the demonstration, you must have `mttest` running on a single-CPU system. As before, Sun Workshop must be installed on the machine, so you can run the Collector to collect performance data.

1. **Start Sun WorkShop by typing:**

```
% workshop
```

2. **Click the Debugging button to open the Debugging window.**

3. **Load `mttest` into the Debugging window:**

   a. **Choose Debug ➤ New Program from the main Debugging menu.**

   b. **In the Debug New Program dialog box, enter the `mttest` path in the Name text box, or use the list box to navigate to `mttest`.**

   c. **Click OK.**

4. **From the Debugging window menu bar, choose Windows ➤ Sampling Collector to open the Sampling Collector window.**

   Observe the following points:

   - The Collector is enabled for one run only.
   - The default path and file name for the experiment-record file appear in the Experiment File text box. Change the name of the experiment-record file to `mttest.2.er`.
   - Under the Data to Collect check boxes, the only type of data selected for collection is the default, Clock-Based Profiling.

5. **For this experiment, in addition to clock-based profiling, you are going to collect synchronization wait tracing information, so select the Synchronization Wait Tracing check box.**

6. **Click the Start button:**

`mttest` runs in the Debugging window, and the Collector collects clock-based profiling and synchronization wait tracing data and stores it in the experiment-record file, `mttest.2.er`.

Analyzing `mttest` Performance Metrics (One-CPU System)

Load `mttest.2.er` into the Analyzer:

1. **Type:**

```
% analyzer mttest.2.er
```

The Analyzer window opens with the Function List for `mttest` displayed.

2. **In the File List display, scroll down to the data for `lock_local()` and `lock_global()`.**

   As on the four-CPU system, both functions have the same inclusive user CPU time (in this case about 10 seconds). This indicates that both functions are doing the same amount of work.

   However, total LWP time for `lock_global()` is actually less than for `lock_local()` (25 seconds versus 37 seconds). This could be because of the way each locking system 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, so that the first thread executes for 2.5 seconds and then exits, while the others remain in a wait state until it is through; then the next thread, having waited for the first thread, executes for 2.5 seconds, for a total LWP time of 5 seconds; the third thread waits 5 seconds for the first two threads and then executes, for a total LWP time of 7.5 seconds; and the fourth thread, having waited 7.5 seconds, executes with a total LWP time of 10 seconds. `lock_local()`'s local lock, on the other hand, 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, so that all four threads remain in a wait state for almost three-quarters of the 10-second execution time.

   `lock_global()` still uses a lot of time in synchronization waiting (almost 15 seconds: 2.5 plus 5 plus 7.5), whereas `lock_local()` still uses no time in synchronization waiting. `lock_local` applies a lock only to data in a particular thread's work block. Its work proceeds as on the four-CPU system without contention or time wasted in synchronization waits. The synchronization wait time for `lock_local()` again is 0 (zero) seconds.

3. **Return to the Function List display in the main Analyzer window, and scroll to the data for the functions `ComputeA()` and `ComputeB()`.**

4. **Click on the line of data for `ComputeA()` to select it.**

5. **Click the Source button in the lower Analyzer tool bar.**

   The text editor opens, displaying the annotated source code for `ComputeA()`.

6. **Scroll down in the text editor so that the source for `ComputeA()` and `ComputeB()` is displayed.**

   Observe that on the single-CPU system, inclusive user CPU time for `ComputeA()` and `ComputeB()` is virtually identical.

7. **Return to the Function List display in the main Analyzer window, and click on the line of data for `ComputeA()` to select it.**

8. **Click the Callers-Callees button in the lower tool bar.**

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

9. **Click on caller `cache_trash()` to select it.**

10. **Return to the Function List display in the main Analyzer window.**

    Note that `cache_trash()` is now the selected function.

11. **Click the Source button in the lower Analyzer tool bar.**

    The text editor open, displaying the annotated source code for `cache_trash()`.

12. **Scroll up to display the code for the function that calls `ComputeA()`, which is `lock_none()`.**

13. **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]`), which can be read and written to directly without danger of contention with other threads.

    `ComputeB()` is called with a series of doubles that occupy successive words in memory (`&element[array->index]`), accessed via a data cache. Recall that if multiple threads are running on multiple CPUs, 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 obsolete. If one of them needs the data again later in the program, it must copied back into the data cache from memory, a time-consuming operation. The resulting cache misses waste a lot of CPU time.

    However, when only one thread is running, no other threads write to memory, so 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 on a one-CPU machine as the performance for `ComputeA()`.

CHAPTER 3

Sampling Collector Reference

This chapter introduces the Sampling Collector and explains how to use it. It covers the following topics:

- What the Sampling Collector Collects
- Collecting Performance Data in Sun WorkShop
- Starting a Process Under the Collector in `dbx`
- Attaching to a Running Process
- Using the Collector for Programs Written with MPI

The Sampling Collector collects performance data from your target application and the kernel under which your application is running, and writes that data to an experiment record file. An *experiment* is the data collected during one execution of your application.

Unless you have specified otherwise, experiment-record files generated by the Sampling Collector have the extension `.`*n*`.er`, where *n* is an integer 1 or higher. The default experiment-file name is `test.`*n*`.er`. If you use the *filename*`.1.er` format for your experiment-record file name, the Collector automatically increments the names of subsequent experiments by one—for example, `my_test.1.er` is followed by `my_test.2.er`, `my_test.3.er`, and so on.

Caution – Do not use the `rm` utility to delete an experiment-record file. The actual experiment information is stored in a hidden directory, `.`*filename*`.`*n*`.er`, which `rm` does not remove. To delete an experiment record file and also its hidden directory, use the performance-tool utility `er_rm`, which is included with the Collector and Analyzer. See the `er_rm` man page for information about using `er_rm`.

What the Sampling Collector Collects

The Sampling Collector records performance data, and organizes the data into *samples*, each of which represents an interval within an experiment. The Sampling Collector terminates one sample and begins a new one in the following circumstances:

- When it encounters a breakpoint (see *Introduction to Sun WorkShop* for information about setting breakpoints in Sun WorkShop Debugging)
- When the sampling interval expires, if you have set a sampling interval
- When you choose Collect ➤ New Sample or click the New Sample button, if you have selected manual sampling

The data recorded for each sample consists of microstate accounting information from the kernel and various other statistics maintained within the kernel.

All data recorded at sample points is global to the program and does not include function-level metrics. However, if function-level metrics have been recorded during the sampling interval, the Collector associates these function metrics with the sampling interval during which they were collected.

The Sampling Collector can gather the following types of function-level information:

- Clock-based profiling data
- Thread-synchronization wait tracing
- Hardware-counter overflow profiling

Exclusive, Inclusive, and Attributed Metrics

The Collector collects exclusive, inclusive, and attributed function and load-object metrics.

- Exclusive data applies to time spent in the function itself.
- Inclusive data applies to time spent in the function itself and also to time spent in any function it calls. Time from callees is counted only for calls from the given function.
- Attributed data of a given function applies to the sum of metrics that occur in a callee of the given function and any functions the callee calls as a result of the call from the given function. The following conditions apply to attributed metrics:
  - The attributed metric for any caller of a given function is the metric that occurs in the given function and any function or functions it calls as a result of the caller's call to it.

- A caller's attributed metric equals the contribution of the given function to the caller's inclusive metric.
- The sum of the attributed metrics of all a given function's callers equals the given function's inclusive metric.
- The attributed metric of a callee of the given function is that fraction of the callee's inclusive metric that resulted from the call from the given function.
- The difference between a callee's attributed metric and its inclusive metric represents that portion of the callee's inclusive metric that resulted from calls from callers other than the given function.
- The inclusive metric of the given function equals its own exclusive metric plus the sum of all the attributed metrics of the given function's callees.

Clock-Based Profiling Data

Clock-based profiling records information to support the following metrics:

- **User CPU time.** Time during which your application is running on the CPU.
- **Total LWP time.** Total execution time across all LWPs (lightweight processes).
- **Wall-clock time.** LWP time spent in thread 1.
- **System CPU time.** Total CPU time, within the operating system or in trap state for the LWP.
- **System wait time.** LWP time spent waiting for the CPU, for a lock, or for a kernel page, or time spent sleeping or stopped.
- **Text-page fault time.** LWP time spent waiting for a text page.
- **Data-page fault time.** LWP time spent waiting for a data page.

This information appears in the Function List display and the Callers-Callees window of the Analyzer. (See "Examining Metrics for Functions and Load-Objects" on page 52.) It also appears in the Summary Metrics window and annotated source and disassembly.

Note – For multiprocessor experiments, times other than wall-clock time are summed across all LWPs in the process. Total time equals the wall-clock time multiplied by the average number of LWPs in the process. Each record contains a timestamp and the IDs of the thread and LWP running at the time of the clock tick.

Clock-based profiling helps answer the following kinds of questions:

- How much of the available resources does the application consume?
- Which functions are consuming the most resources?
- Which source lines and disassembly instructions consume the most resources?
- How did the program arrive at this point in the execution?

Thread Synchronization Wait Tracing

In multithreaded programs, thread synchronization wait tracing keeps track of wait time on calls to thread-synchronization routines in the threads library; if the real-time delay exceeds a certain user-defined threshold, an event is recorded for the call, as well as the wait time, in seconds.

Each record contains a timestamp and the IDs of the thread and LWP running at the time of the clock stamp. Synchronization-delay information supports the following metrics:

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

This information appears in the Function List display and the Caller-Callee window of the Sampling Analyzer (see "Examining Metrics for Functions and Load-Objects" on page 52). It also appears in the Summary Metrics window and annotated source and disassembly.

Hardware-Counter Overflow Profiling

Hardware-counter overflow profiling records the callstack of each LWP at the time a designated hardware counter of the CPU on which the LWP is running overflows. The data recorded includes a timestamp and the IDs of the thread and the LWP.

The collector allows you to select the type of counter whose overflow is to be monitored, and to set an overflow value for it. Typically, counters keep track of such things as instruction-cache misses, data-cache misses, cycles, or instructions issued or executed.

Note – Hardware-counter overflow profiling can be done only on Solaris 8 for SPARC (UltraSPARC III) machines and on x86 (Pentium II and compatible products). On other machines, this feature is disabled.

Hardware-counter overflow profiling produces data to support count metrics.

Global Information

Global information about your program includes the following kinds of data:

- **Execution statistics.** Include page fault and I/O data, context switches, and a variety of page residency (working-set and paging) statistics. This information appears in the Execution Statistics display of the Sampling Analyzer. (See "Examining Execution Statistics" on page 76.)
- **Address-space data (optional).** Consists of page-referenced and page-modified information for every segment of the application's address space. This information appears in the Address Space display of the Sampling Analyzer. (See "Examining Address-Space Information" on page 74.)

Collecting Performance Data in Sun WorkShop

Before you can collect data, you must do the following:

- Load your program into the Debugging window. (See *Introduction to Sun WorkShop* for information about how to start Sun WorkShop and access the Debugging window.)
- Ensure that run-time checking is turned off (the default).

Collecting data requires two steps:

1. Specifying the kinds of data you want to collect and where you want to store the data.
2. Running the Collector.

To specify the kinds of data you want to collect:

1. **From the WorkShop window menu bar, choose Window ➤ Sampling Collector.**
   The WorkShop Sampling Collector window appears.

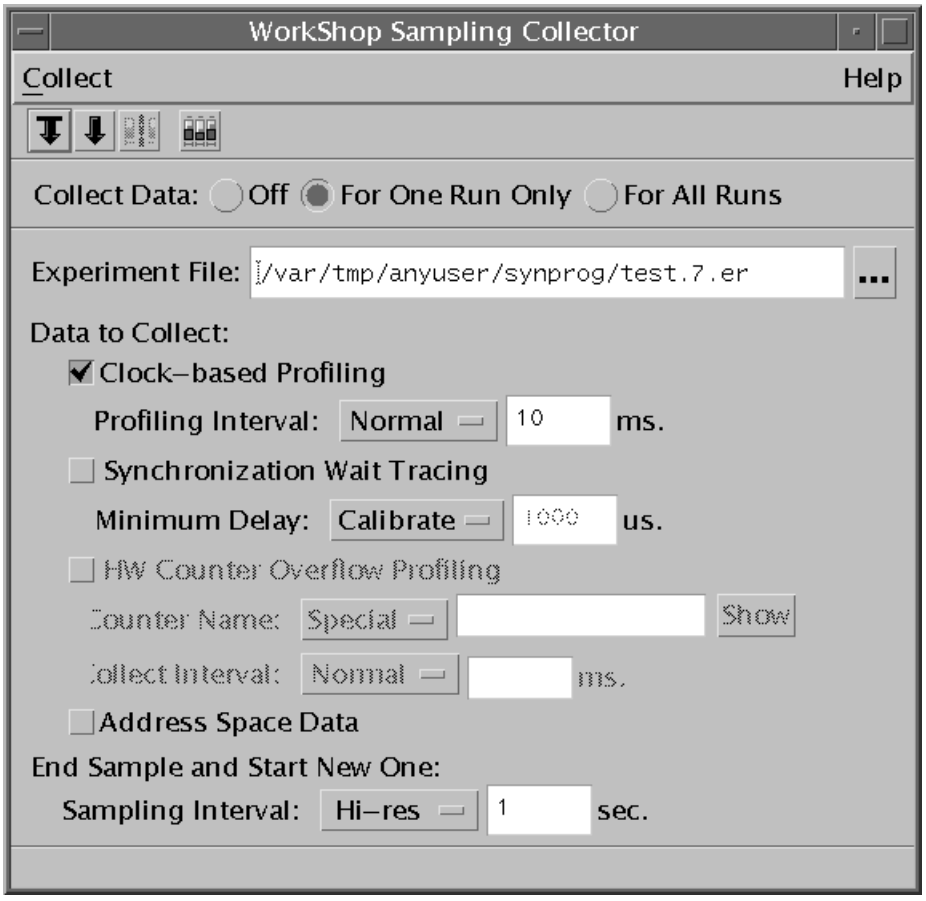

FIGURE 3-1 The Sampling Collector Window

2. **Use the Collect Data radio buttons to specify whether you want to collect data for this one run only or for multiple runs.**
   - If you select “for one run only“, the Collector is disabled after your program has run once and the data for that run is stored in the experiment record file.
   - If you select “for all runs“, the Collector remains active after your program has finished running, and for each subsequent run, it creates a new experiment record file to store the data for that run.
   - If you select “off“, the Collector is disabled and collects and stores no data until you select one of the other Collect Data radio buttons.

3. **In the Experiment File text box, specify the path and file name of the experiment-record file in which you want the data to be stored.**

   The default experiment-record file name provided by the Collector is `test.1.er`. If you want to name your file a different file name, you must enter a path (if you do not want it to be stored in the default directory) and file name for it.

   If you use the `.1.er` suffix for your experiment-record file name, the Sampling Collector automatically increments the names of subsequent experiments by one. For example, `test.1.er` is followed by `test.2.er`.

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

   You can accept the Normal profiling interval (10 milliseconds) or from the Profiling Interval list box you can select Hi-res (1 millisecond) or Custom, where you set your own interval in milliseconds.

   High-resolution profiles record ten times as much data as normal profiles for any given run. To support high-resolution profiling, the operating system must be running with a high-resolution clock routine. You can specify a high-resolution routine by adding the following line to the file `/etc/system`, and then rebooting:

   ```
   set hires_tick=1
   ```

   Note – If you try to set high-resolution profiling on a machine whose operating system does not support it, the Collector posts a warning message and reverts to the highest resolution supported. A custom setting that is not a multiple of the resolution supported by the system is rounded to the nearest multiple of that resolution, and the Collector issues a warning message.

5. **To collect information about thread-synchronization wait counts and times, select the Synchronization Wait Tracing check box.**

   To specify the threshold beyond which tracing begins:

   - You can accept the default, Calibrate (the threshold is determined at run time).
   - From the Minimum Delay list box you can select a threshold of:
     - 1000 microseconds
     - 100 microseconds
     - 0 (zero) microseconds (all synchronization waits are traced, regardless of wait time)
     - Custom (set your own threshold in microseconds)

6. **To collect information about hardware counter overflows, select the HW Counter Overflow Profiling check box.**

7. **Choose a category of counters from the Counter Name menu, then click Show for a list of all counters available in that category. The user-recognizable name of the counter you select appears in the Counter Name text box.**

8. **To specify the number of increments that take place between one overflow event and the next, choose the default Normal from the Collect Interval menu (the value of Normal depends on the counter you have selected), or choose Custom and type a value in the Collect Interval text box.**

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

---

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

10. **You can accept the default Hi-res sampling interval (a 1-second interval), or from the Sampling Interval list box you can select Normal (a 10-second interval), Custom (set your own interval in seconds), or Manual, in which you signal the end of the current sample and the beginning of a new one by either choosing Collect ➤ New Sample in the WorkShop Sampling Collector window, or clicking the New Sample button:**

Now you are ready to collect data. To run the Sampling Collector, in the WorkShop Sampling Collector window:

- **Choose File ➤ Start, or click the Start button:**

Starting a Process Under the Collector in dbx

You can run the Collector from `dbx`, as well as from the Sun WorkShop Debugging window. To do this:

1. **STart your program in `dbx` by typing:**

```
% dbx program_name
```

2. **Press the space bar until the `(dbx)` prompt appears.**

3. **Use the `collector` command with its various arguments to collect data and generate an experiment record:**

```
(dbx) collector argument
```

The `collector` command arguments are listed in TABLE 3-1.

TABLE 3-1 `collector` Command Arguments

Argument	What It Does
{ `enable` | `enable_once` | `disable` }	Enables or disables data collection. • If the mode is `enable`, data collection is enabled for the current run and all subsequent runs. • If the mode is `enable_once`, data is collected for the current run, and the mode is reset to `disable` when the run ends. • If the mode is `disable`, no performance data is collected.
`profile` { `on` | `off` }	Enables or disables collection of profiling data. The default is `on`.
`profile timer` *value*	Sets the profiling timer interval to *value*, given in milliseconds. The default is 10 ms.
`address_space` { `on` | `off` }	Enables or disables collection of address-space data (pages that have been referenced and modified). The default is `off`.
`synctrace` { `on` | `off` }	Enables or disables collecting of thread-synchronization wait tracing data. Default is `off`.

TABLE 3-1 `collector` Command Arguments *(Continued)*

Argument	What It Does
`synctrace threshold` *value*	Sets the threshold for synchronization delay tracing according to the given *value*, in microseconds. *value* is one of the following: • `calibrate`: Use a calibrated threshold, determined at runtime. • *number*: Use a threshold of *number*, given in microseconds. Setting *number* to `0` (zero) causes the collector to trace all events, regardless of wait time. The default setting is `calibrate`.
`hwprofile` { `on` | `off` }	Enables or disables hardware-counter overflow profiling. The default is `off`. If you attempt to enable hardware-counter overflow profiling on systems that do not support it, dbx returns an error message.
`hwprofile list`	Returns a list of available counters by name, with two numeric settings, one for a normal interval and one for a higher-resolution interval. If your system does not support hardware-counter overflow profiling, dbx returns an error message.
`hwprofile counter` *name interval*	Sets the hardware-counter profiling to the event *name*, and its overflow interval to *interval*. The default for *name* is `cycles`, at the normal-profiling interval.
`status`	Reports on the status of any loaded experiment.
`show`	Shows the current setting of every collector control.
`close`	Closes the current experiment.
`quit`	Terminates data collection for the current run.
`sample` { `periodic` | `manual` }	Sets the sampling mode to either `periodic` (for which you must use the `sample period` argument to set a value) or `manual`.
`sample period` *value*	Sets the sampling frequency to *value*, given in seconds.
`store directory` *directory_name*	Sets the directory where the experiment record file is stored to *directory_name*.
`store filename` *file_name*	Sets the output experiment file name to *file_name*.

To get a listing of available `collector` command arguments, type the following at the `(dbx)` prompt and press Enter:

```
(dbx) help collector
```

Attaching to a Running Process

The Collector allows you to attach to a running process and collect performance data from the process.

If you want to collect thread synchronization wait tracing, load the library `libcollector.so` before you start the executable, so the Collector's wrapper around the real synchronization routines is referenced, rather than the actual routines themselves. If you are collecting only profiling data or hardware-counter overflow profiling, you do not need to preload the collector library, although you can do so if you wish.

To preload `libcollector.so`:

- **Set the environment variable `LD_PRELOAD` to point to `libcollector.so`, as shown in** TABLE 3-2**.**

TABLE 3-2 Commands for Setting `LD_PRELOAD`

Platform	Command Sequence
`csh`	`setenv LD_PRELOAD` *install_directory*`/SUNWspro/WS6/lib/dbxruntime/libcollector.so`
`sh`, `ksh`	`LD_PRELOAD=`*install_directory*`/SUNWspro/WS6/lib/dbxruntime/libcollector.so` `export LD_PRELOAD`
SPARC-V9 executables on `csh`	`setenv LD_PRELOAD` *install_directory*`/SUNWspro/WS6/lib/v9/dbxruntime/libcollector.so`
SPARC-V9 executables on `sh/ksh`	`LD_PRELOAD=`*install_directory*`/SUNWspro/WS6/lib/v9/dbxruntime/libcollector.so` `export LD_PRELOAD`

install_directory is the directory that contains your distribution (normally, it is `/opt/`).

Note – Remove the `LD_PRELOAD` setting after the run, so it will not remain in effect for all other programs started from the same shell.

To attach to the executable and collect data:

1. **Start the executable.**
2. **Determine the executable's PID, and attach `dbx` to it.**

- If the executable is running in the background, its PID will be printed to standard output by the shell.
- You can determine the executable's PID by typing:

```
% ps -ef | grep program_name
```

3. **Enable data collection:**

   a. **Start collecting data, either directly in `dbx` using the `collector` command, or from the Sampling Collector window.**

   b. **Use the `cont` command to resume the target process from `dbx`.**

Note – If you have started the executable from `dbx` without enabling data collection, you can pause the target from `dbx` and then execute the preceding instructions to start data collection during the run.

Using the Collector for Programs Written with MPI

The Collector can collect performance data from multi-process programs that use the Sun Message Passing Interface (MPI), if you use the Sun Cluster Runtime Environment (CRE) command `mprun` to launch the parallel jobs. Use ClusterTools 3.1 or a compatible version. See the Sun HPC ClusterTools 3.1 documentation for more information.

To collect data from MPI jobs, you must either start the MPI processes under `dbx`, or attach `dbx` to each process separately. For example, suppose you run MPI jobs using a command line like the following:

```
% mprun -np 2 a.out [program-arguments]
```

You can replace that command line with the following:

```
% mprun -np 2 dbx a.out < collection.script
```

where `collection.script` is a `dbx` script, described in the following paragraphs.

When this example is executed, two MPI processes from the executable `a.out` run, and two experiments are created, named `test.`*M*`.er` and `test.`*N*`.er`, where *M* and *N* are the PIDs of the two MPI processes.

Your file `collection.script` must ensure that the experiments created are uniquely named. Otherwise, because the `dbx` instances that generate the experiments are being run simultaneously, two or more of the `dbx` instances might attempt to create experiments with the same name. One way to ensure uniquely named files is to make your script specify that each `dbx` instance must use a file name with the process ID of the MPI process in it:

```
stop in main
run [program_arguments]
collector enable
collector store filename test.$[getpid()].er
cont
quit
```

You can also create experiments named with the MPI rank, by using a slightly different `dbx` script in which you stop the target program immediately following a call to `MPI_Comm_rank()` and use the rank to specify the experiment directory. For example, suppose your MPI program contains one of the following statements on line 17:

- For a C program:

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

- For a Fortran program:

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

Change `collection.script` to read as follows:

```
stop at 18
run [program_arguments]
rank=$[me]
collector enable
collector store filename test.$rank.er
cont
quit
```

With this modification, `mprun` creates experiments named with the rank of the MPI process to which they correspond.

To examine the data collected from the MPI processes, open one experiment in the Analyzer, then add the others, so you can see the data for all the MPI processes in aggregate. See "Starting the Analyzer and Loading an Experiment" on page 50 and "Adding Experiments to the Analyzer" on page 77 for more information.

You can also use `er_print` to print out the data. `er_print` accepts multiple experiments on the command line. For information about using `er_print`, see Chapter 5.

CHAPTER 4

Sampling Analyzer Reference

This chapter discusses the Sampling Analyzer and how to use it. It covers the following topics:

- Starting the Analyzer and Loading an Experiment
- The Analyzer Window
- Examining Metrics for Functions and Load-Objects
- Examining Caller-Callee Metrics for a Function
- Examining Annotated Source Code and Disassembly Code
- Filtering Information
- Generating and Using a Mapfile
- Using the Data Option List to Access Other Data Displays
- Adding Experiments to the Analyzer
- Dropping Experiments from the Analyzer
- Printing the Display

The Sampling Analyzer analyzes the program performance data collected by the Sampling Collector. It reads in experiment-record files generated by the Collector and provides you with options for examining and manipulating the experiment data, so you can identify program execution bottlenecks and analyze and improve program performance.

See Chapter 2 for examples of how you might use the Analyzer to fine-tune an application.

See Chapter 6 for a description of program execution and the behavior you see in the Analyzer.

Starting the Analyzer and Loading an Experiment

Note – Loading an experiment discards all data for any previously loaded experiment in the Analyzer. However, it does not affect recorded experiments.

To use the Analyzer, you must start it and load an experiment record into the Analyzer window. You can do this from the command line:

- **Type the following, where *experiment_name* is the name of the experiment record file you want to load:**

```
% analyzer experiment_name
```

Experiment names are usually of the form `test.`*n*`.er`.

Or you can open the Analyzer from the main Sun WorkShop window or the Sampling Collector window, and then load the experiment:

1. **Click the Analyzer button in either the main Sun WorkShop window or the Sampling Collector window.**

2. **In the Load Experiment dialog box, which opens automatically when you start the Analyzer without specifying an experiment, double-click the experiment record file that you want to load.**

You can also use the Load Experiment dialog box to load an experiment at any time when the Analyzer is running:

- **Choose Experiment ➤ Load from the Analyzer main menu bar to open the Load Experiment dialog box.**

Analyzer Command-Line Options

There are two command-line options you can use when you invoke the Analyzer from the command line. They are described in TABLE 4-1:

TABLE 4-1 Analyzer Command-Line Options

`-s` *session_name*	If you are running multiple instances of the Analyzer, opens a text editor for each of them instead of forcing them to share a single one; this makes it possible, for example, to compare annotated source or disassembly code for two different experiments.
`-V` *experiment_file*	Prints the version number of the Analyzer to `stdout` .

Exiting the Analyzer

To exit the Analyzer:

- **Choose Experiment ➤ Exit from the Analyzer main menu bar.**

The Analyzer Window

The Analyzer window is the main display that you see when you open the Analyzer. The window contains a main menu bar, upper and lower tool bars, and a central display pane in which the experiment data appears.

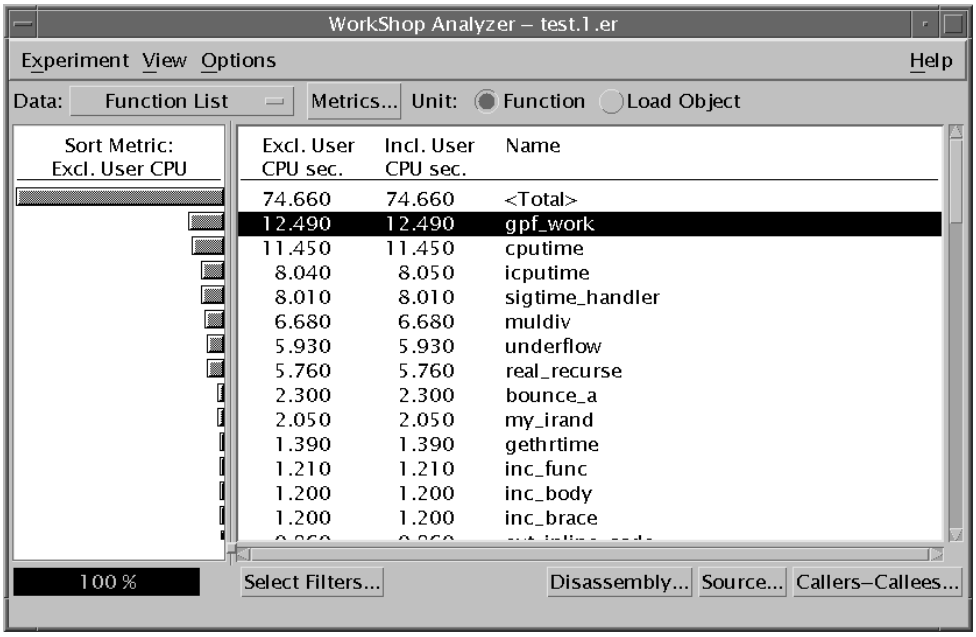

FIGURE 4-1 The Analyzer Window

Examining Metrics for Functions and Load-Objects

The Function List is the default display that appears when you open the Analyzer. It contains metrics specific to functions and load objects. It is divided into two display panels:

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

For each function or load-object metric displayed, the Function List provides an absolute value in seconds or counts and a percentage of the total program metric.

Viewing Metrics for Functions and Load Objects

By default, the Function List displays function metrics.

To switch to load-object metrics:

- **Choose Load Object from the Unit radio buttons in the upper tool bar.**

To return to the function metrics display:

- **Choose Function from the Unit radio buttons in the upper tool bar.**

Understanding the Metrics Displayed

The Function List can display the following types of function and load-object metrics:

- Clock-based profiling metrics
- Thread synchronization wait tracing metrics
- Hardware-counter overflow profiling metrics

See "What the Sampling Collector Collects" on page 36 for a description of each data type.

By default, assuming that the supporting data has been collected, the Function List displays the following metrics:

- Exclusive user CPU time
- Inclusive user CPU time
- Inclusive thread-synchronization wait time (if recorded)
- Inclusive thread-synchronization wait counts (if recorded)
- Exclusive hardware-counter overflow profiling counts (if recorded)
- Inclusive hardware-counter overflow profiling counts (if recorded)

The metrics are sorted on exclusive user CPU time, if it has been recorded.

In the Select Metrics dialog box you can select other data to display and specify a different sort order. See "Selecting Metrics and Sort Order for Functions and Load-Objects" on page 55 for instructions.

Clock-Based Profiling Metrics

Clock-based profiling is based on wall-clock time. It shows how much time, exclusive and inclusive, your program spends in each function or load object, which helps to identify where program bottlenecks are occurring. (See "Exclusive, Inclusive, and Attributed Metrics" on page 36 for an explanation of exclusive and inclusive metrics.)

Clock-based profiling data supports the following execution-time metrics for each function in the program:

- **User CPU time.** Time during which your application is running on the CPU.
- **Total LWP time.** Total execution time across all LWPs.
- **Wall-clock time.** LWP time spent in thread 1, within the operating system or in trap state for the LWP.
- **System CPU time.** Total CPU time.
- **System wait time.** LWP time spent waiting for the CPU, for a lock, or for a kernel page, or time spent sleeping or stopped.
- **Text-page fault time.** LWP time spent waiting for a text instruction or page.
- **Data-page fault time.** LWP time spent waiting for a data page.

You have the option of examining each of these values in seconds and as a percentage of the total program metric. Except for wall-clock time, all metrics are summed across all LWPs.

Thread-Synchronization Wait Tracing

In multithreaded programs, thread synchronization wait tracing keeps track of wait time on calls to thread-synchronization routines in the threads library; if the real-time delay exceeds a certain user-defined threshold, an event is recorded for the call, and the wait, in seconds, is recorded.

Synchronization wait tracing supports, for each function or load object, data concerning the count of events recorded and the total number of seconds over threshold spent waiting on calls to thread-synchronization routines. From this information you can determine if functions or load objects are either frequently left on hold, or experience unusually long wait times when they do make a call to a synchronization routine.

High synchronization wait times indicate contention among threads. You can reduce the contention by reworking your algorithms, particularly restructuring your locks so that they cover only the data for each thread that needs to be locked.

Hardware-Counter Overflow Profiling

Hardware-counter overflow profiling records the callstack of each LWP at the time the hardware counter of the CPU on which the LWP is running overflows. The data recorded includes a timestamp and the IDs of the thread and the LWP.

Typically, hardware-counter overflow profiling supports data on instruction-cache misses, data-cache misses, cycles, or instructions issued or executed.

High counts of cache misses indicate that restructuring to improve locality or increase reuse will improve program performance.

High cycle counts generally correlate with high clock-based profiles, though a cycle experiment reduces the chance of correlation with the clock.

Selecting Metrics and Sort Order for Functions and Load-Objects

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

To change the types of data that appear in the Function List display and their sort order:

1. **In the Function List display, click the Metrics button in the upper tool bar.**

   The Select Metrics dialog box appears.

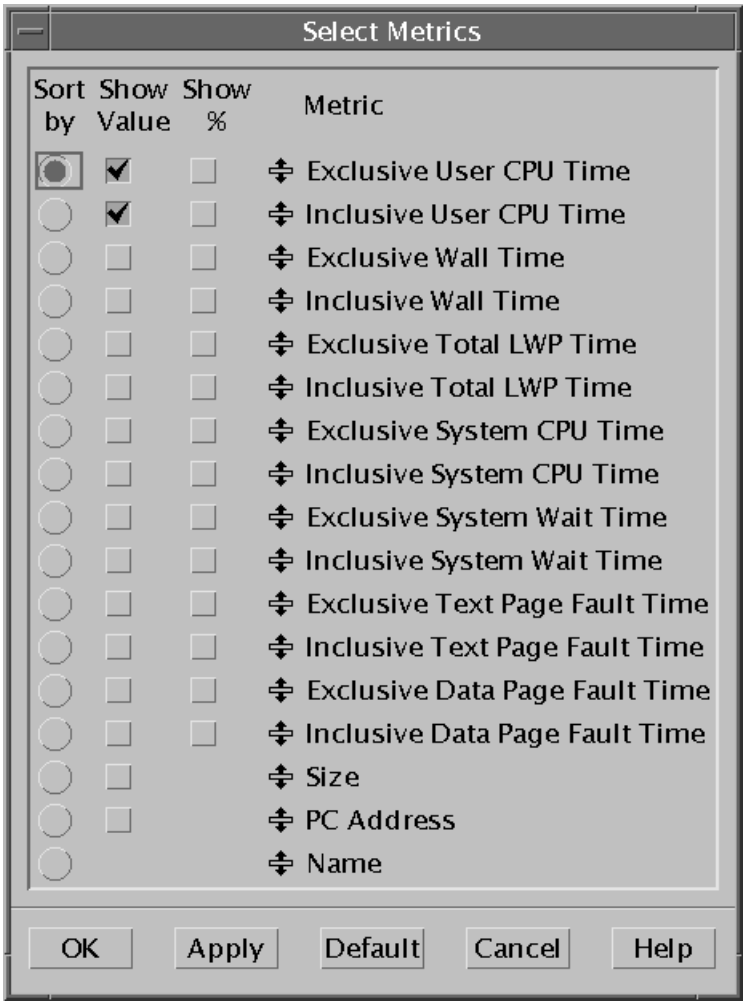

FIGURE 4-2 The Select Metrics Dialog Box

The types of data listed in the Select Metrics dialog box depend on the data collected by the Sampling Collector. If all data types were selected when the Collector was run, the following metrics, exclusive and inclusive, are available from the Metrics dialog box:

- User CPU time
- Total LWP time
- Wall-clock time (LWP time in thread 1)
- System CPU time
- System wait time
- Text-page fault time
- Data-page fault time

- Synchronization-wait counts (if recorded)
- Synchronization-wait time (if recorded)
- Hardware-counter overflow profiling counts (if recorded)

All the previously listed data is available as absolute values (time in seconds or counts) and as a percentage of the total program metric.

In addition, you can choose to display the following for functions or load objects:

- Size, in bytes
- Program-counter address

The names of functions or load objects are always displayed.

2. **To display a particular type of metric, select the appropriate check box in the "Value" or "%" column of the Select Metrics dialog box.**
3. **To specify sort order, select the appropriate radio button from the "Sort" column of the Select Metrics dialog box.**
4. **To make your selections appear in the Function List display, click OK to close the Metrics dialog box, or click Apply to apply the new selections and keep the dialog box open.**

Note – To customize how the metrics are grouped in the Select Metrics dialog box, click on the icon next to a metric name, then drag and drop it onto the metric above which you want it to appear.

Viewing Summary Metrics for a Function or Load Object

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

To see summary metrics for a function or load object:

1. **Use the Unit radio buttons to select function display or load-object display.**
2. **Select the function or load object by clicking it in the right Function List display pane.**
3. **Choose View ➤ Show Summary Metrics to open the Summary Metrics window.**

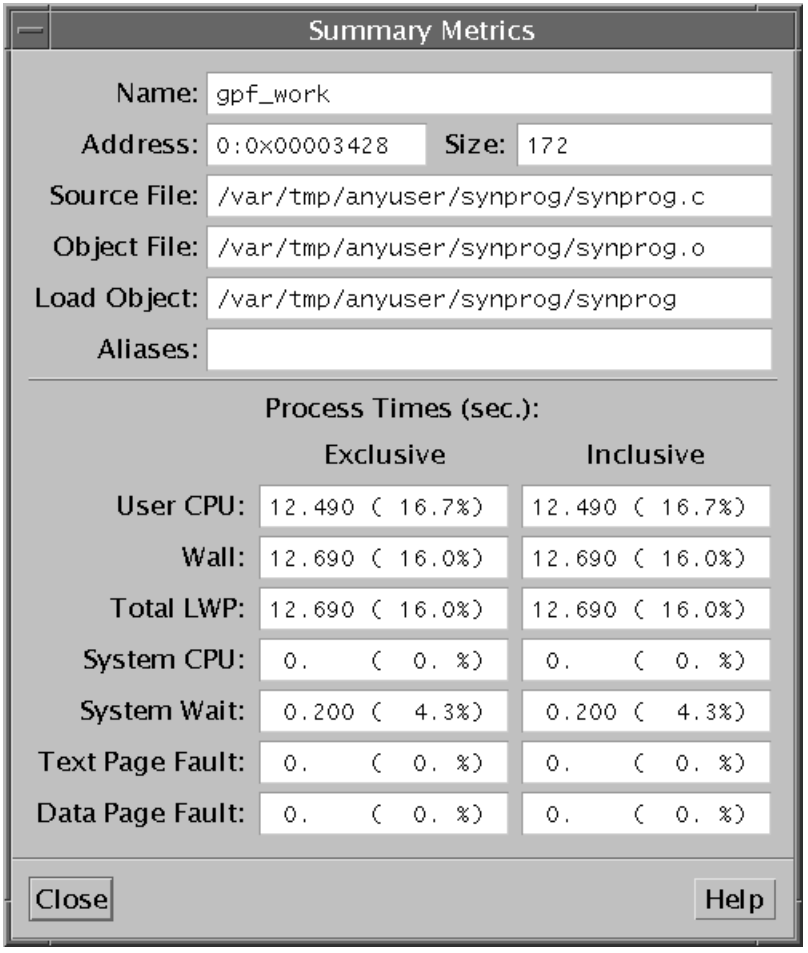

FIGURE 4-3 The Summary Metrics Window

The Summary Metrics window contains the following information, exclusive and inclusive, for the selected function or load object, if the information has been collected by the Collector:

- Memory address
- Size of the function (in bytes)
- User CPU time
- Total LWP time
- Wall clock time (LWP time in thread 1)
- System CPU time
- System wait time
- Text-page fault time
- Data-page fault time
- Synchronization wait count (if recorded)

- Synchronization wait time (if recorded)
- Hardware-counter overflow profiling count (if recorded)

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

In addition, for functions, the Summary Metrics window lists the source file, object file, and load object where code for the function resides.

Note – Metrics do not have to appear in the Function List display to be visible in the Summary Metrics window. You can use the Summary Metrics window to access all available function data without using the Select Metrics dialog box to change the Function List display.

Searching for a Function or Load Object

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

To search for a particular function or load object:

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

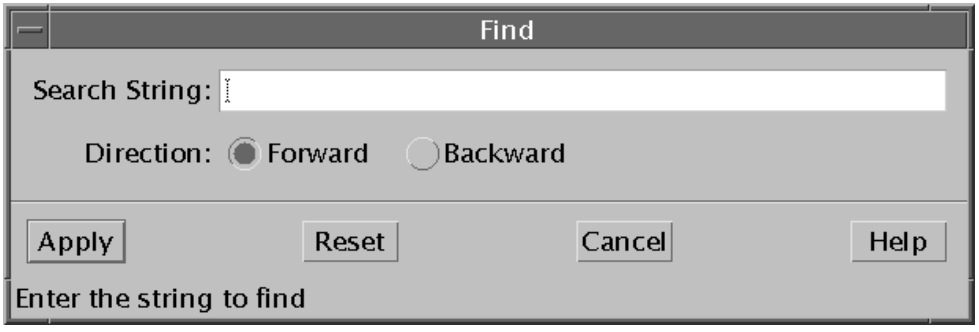

FIGURE 4-4 The Find Dialog Box

2. **In the Search String text box, type a string to search on.**
3. **You can specify the search direction by selecting one of the Direction radio buttons. The default is Forward.**

4. **Click Apply.**

   If the search is successful, the row of data for the function that you searched on is highlighted in the Function List display.

5. **To search for other function names matching the search string, click Apply.**

6. **To reset the Search String text box to the last successful search, click Reset.**

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

Examining Caller-Callee Metrics for a Function

You can examine caller and callee metrics for a selected function in the Analyzer's Callers-Callees window. To access the Callers-Callees window:

- **Click the Callers-Callees button in the lower tool bar.**

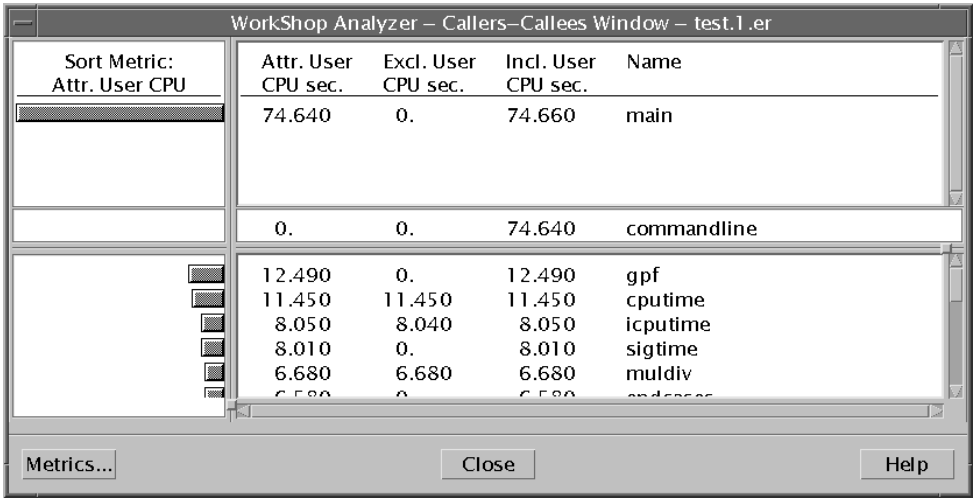

FIGURE 4-5 The Callers-Callees Window

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

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

The Callers-Callees window can display the following metrics for the selected function, any functions that call it, and any functions it calls, if the information was collected by the Sampling Collector:

- User CPU time
- Total LWP time
- Wall clock time (LWP time in thread 1)
- System CPU time
- System wait time
- Text-page fault time
- Data-page fault time
- Synchronization wait count (if recorded)
- Synchronization wait time (if recorded)
- Hardware-counter overflow profiling count (if recorded)

Each of these metrics can be displayed as an absolute value (seconds or counts) and as a percentage of the total program metric.

By default, the Callers-Callees window shows the following metrics:

- Attributed, exclusive, and inclusive user CPU time
- Attributed and inclusive synchronization wait counts
- Attributed and inclusive synchronization wait times (if recorded)
- Attributed hardware-counter overflow counts (if recorded)

The metrics are sorted on attributed user CPU time.

You can navigate through your program's structure by clicking on a function in either the Caller pane or the Callee pane; the display recenters on the newly selected function. By observing exclusive, inclusive, and attributed times, you can locate any function that uses large amounts of execution time.

Selecting Metrics and Sort Order in the Callers-Callees Window

You can specify the data displayed in the Callers-Callees window and its sort order from the Select Callers-Callees Metrics dialog box.

To open the Select Callers-Callees Metrics dialog box:

- **Click Metrics in the Callers-Callees window.**

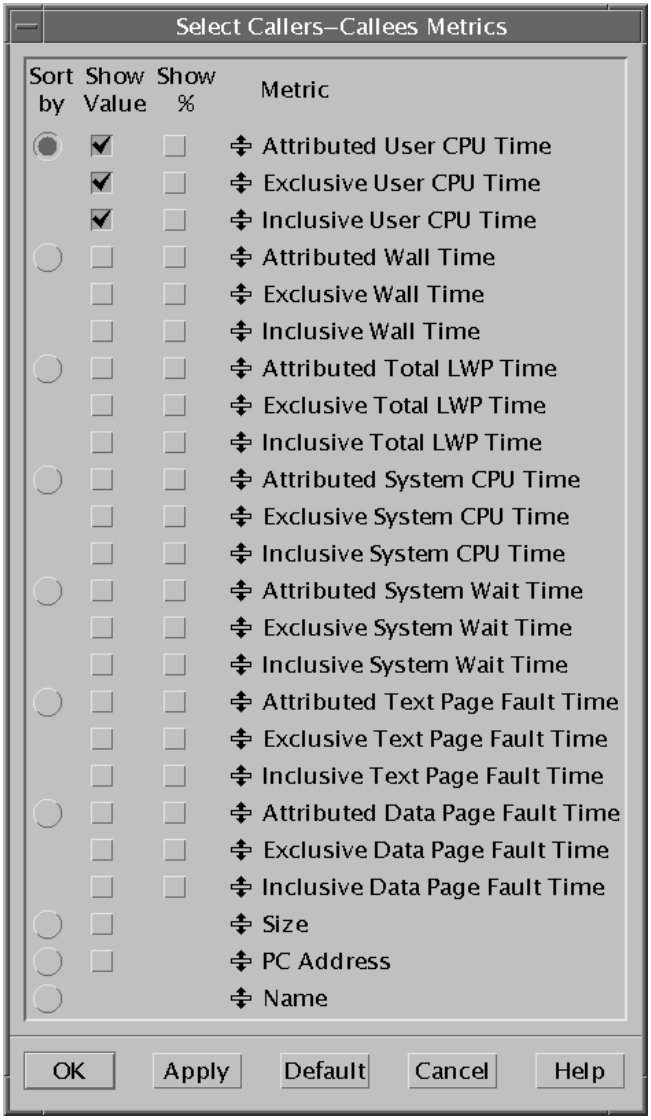

FIGURE 4-6 The Select Callers-Callees Metrics Dialog Box

The Select Callers-Callees Metrics dialog box operates the same way as the Select Metrics dialog box, except that for metrics with attributed, exclusive, and inclusive data, you can sort only on attributed data. (See "Selecting Metrics and Sort Order for Functions and Load-Objects" on page 55.)

Note – To customize how the metrics are grouped in the Select Callers-Callees Metrics dialog box, click on the icon next to a metric name, then drag and drop it onto the metric above which you want it to appear.

Examining Annotated Source Code and Disassembly Code

Once you have identified the function or functions that are slowing program execution, you can generate source code or disassembly for the trouble spot, annotated with performance metrics, so you can identify the actual lines or instructions that are causing the problem.

To display annotated source code for a function:

1. **Click on the function in the right Function List display pane to select it.**
2. **Click Source in the lower tool bar of the Analyzer window.**

   Your text editor opens, showing the code for the selected function, with performance metrics for each line of source code displayed to the left of the code.

The four types of metrics that can appear on a line of annotated source code are explained in TABLE 4-2.

TABLE 4-2 Annotated Source-Code Metrics

Metric	Significance
(Blank)	No PC in the program corresponds to this line of code. This should always happen for comment lines. It also happens for apparent code lines in the following circumstances: • All the instructions from the apparent piece of code have been optimized away. • The code is repeated elsewhere, and the compiler did common subexpression recognition and tagged all the instructions with the lines for the other copy. • The compiler mistagged the instruction that really came from that line with an incorrect line number.
`0.`	Some PCs in the program were tagged as derived from this line, but there were 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 mean that the line was not executed, only that it did not show up statistically in a profile and that a thread-synchronization call from that line never had a delay exceeding the threshold.
`0.000`	At least one PC from this line appeared in the data, but the computed metric value rounded to zero.
`1.234`	The metrics for all PCs attributed to this line added up to the non-zero numerical value shown.

To generate annotated disassembly code for a function:

1. **Click on the function in the right Function List display pane to select it.**
2. **Click Disassembly in the lower tool bar of the Analyzer window.**

   Your text editor opens, displaying the disassembly code for the selected function, with performance metrics for each instruction displayed to the left of the code.

Note – If the source for your program is available, and you click Disassembly to generate annotated disassembly code, the source code appears interleaved with the disassembly listing. If the source code is not available, you can still examine the disassembly code.

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

Choosing a Text Editor

The annotated source code and disassembly code open in a text editor, so you can begin editing the code to correct problems. You have the option of choosing the text editor you want to use.

To choose a text editor:

1. **In the Function List display, choose Options ➤ Text Editor Options to open the Text Editor Options dialog box.**
2. **From the Editor to Use list box, choose the editor that you want to use.**

   The available editors are NEdit, Vi, GNU Emacs, XEmacs, and gvim.

Note – Not all of the WorkShop text editors are available in all locales.

Filtering Information

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

- By load objects
- By samples, threads, and/or LWPs

Selecting Load Objects

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

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

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

3. **Click OK to apply your selections and close the Select Load Objects Included dialog box.**

Selecting Samples, Threads, and LWPs

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

Note – When a sample is selected, a drop shadow appears behind it in the right Overview display pane. See "Examining Sample Overview Information" on page 71 for instructions on how to access the Overview display.

You can select samples, threads, and LWPs individually, in ranges, or in groups of any order:

- **Click the Select Filters button in the lower tool bar of any display.**

  The Select Filters dialog box appears, with the following text boxes:

  - Samples
  - Threads
  - LWPs

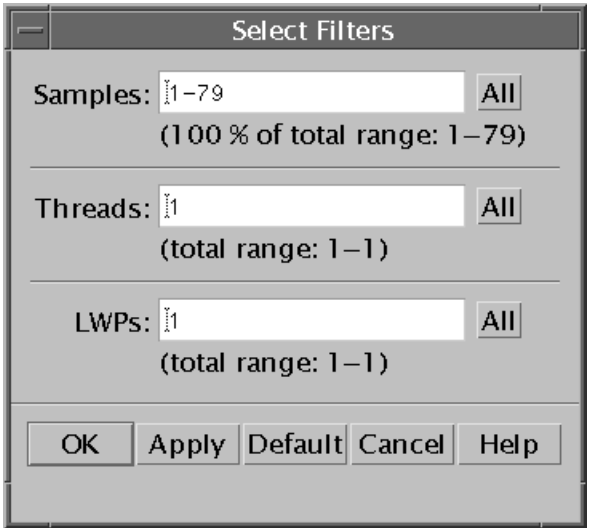

FIGURE 4-7 The Select Filters Dialog Box

Use these text boxes to specify the samples, threads, and LWPs for which you want to display data. You can select samples, threads, and LWPs in any number and any combination.

To select a single sample, thread, or LWP:

- **Type the ID number of the sample, thread, or LWP in the appropriate text box and press Enter.**

To select a range of samples, threads, or LWPs:

- **Type the lower and higher IDs of the range in the appropriate text box, separated by a hyphen (for example, `5-12`) and press Enter.**

To select a non-contiguous set of samples, threads, or LWPs:

- **Type the sample IDs in the appropriate text box, separated by commas (for example, `3,7,15,21`) and press Enter.**

To select all samples, threads, or LWPs in the experiment record:

- **Click the appropriate Samples, Threads, or LWPs All button.**

Generating and Using a Mapfile

Using the data from the experiment record, the 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 I-cache behavior, or both.

To create the mapfile:

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

   For C applications, type:

```
% cc -xF -c a.c b.c
% cc -o application_name a.o b.o
```

   For C++ applications, type:

```
% CC -xF -c a.cc b.cc
% CC -o application_name a.o b.o
```

   For Fortran applications, type:

```
% f95 -xF -c a.f b.f
% f95 -o application_name a.o b.o
```

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

```
ld: warning: mapfile: text: .text% function_name: object_file_name:
Entrance criteria not met named_file, function_name, has not been
compiled with the -xF option.
```

2. **Load your application into Sun WorkShop for debugging and use the Sampling Collector to collect performance data (see "Collecting Performance Data in Sun WorkShop" on page 39). Ensure that you have enabled Address Space data collection.**

3. **Load the experiment that you have just generated into the Analyzer (see "Starting the Analyzer and Loading an Experiment" on page 50).**

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

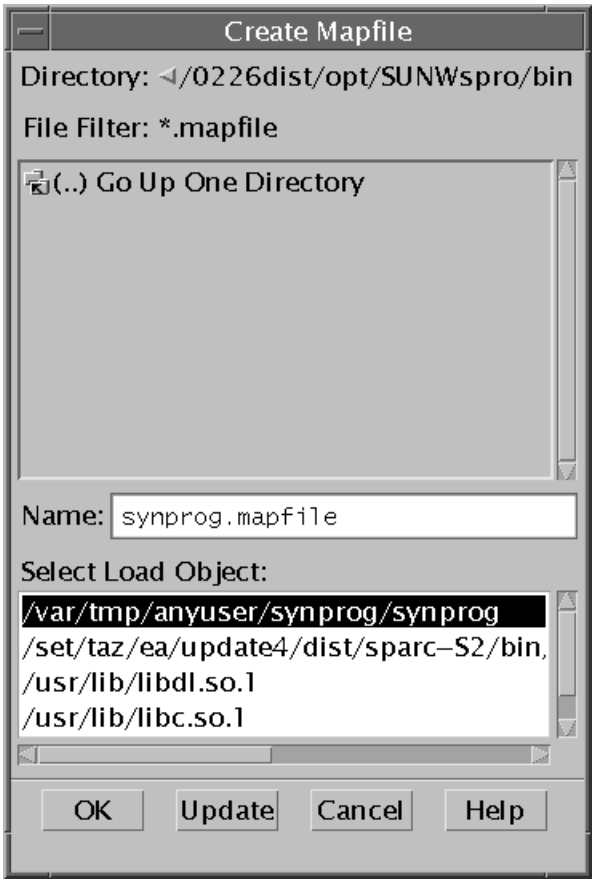

FIGURE 4-8 The Create Mapfile Dialog Box

5. **In the Create Mapfile dialog box, use the directory pane, if necessary, to navigate to the directory where you want to store the mapfile.**
6. **You can use the Name text box to:**
   - Change the file filter to display the name of an existing file and select it for overwriting
   - Type in a path and file name for the mapfile that are different from the default
7. **In the Select Load Object list box, select the load object for which you want to generate the map file (this is usually your program segment).**
8. **Click OK.**

To use the mapfile to reorder your program:

- **Link your object files as you normally would, using the mapfile. For example:**
  For C applications, type the following and press Enter:

```
% cc -Wl -M mapfile_name a.o b.o
```

For C++ applications, type the following and press Enter:

```
% CC -M mapfile_name a.o b.o
```

For Fortran applications, type the following and press Enter:

```
% F90 -M mapfile_name a.o b.o
```

Using the Data Option List to Access Other Data Displays

When you first open the Analyzer window and load an experiment, the Function List, which contains function and load-object information, is the default display. See "Examining Metrics for Functions and Load-Objects" on page 52 for more information about the Function List.

You can use the Data option list in the upper tool bar to change the contents of the display pane to show other kinds of data:

- **Overview.** High-level sample information; see "Examining Sample Overview Information" on page 71 for more information.
- **Address-space.** Information about how your program uses memory; see "Examining Address-Space Information" on page 74 for more information.
- **Execution statistics.** General information about how your program executed; see "Examining Execution Statistics" on page 76 for more information.

You can return to the Function List from other displays by choosing Function List from the Data option list.

Note – Any information that you want to examine in the Analyzer must be collected and stored in an experiment record by the Collector. See "Collecting Performance Data in Sun WorkShop" on page 39 for information on how to determine which data the Collector collects and stores in an experiment record.

Examining Sample Overview Information

To view the Overview display:

- **Choose Overview from the Data list box.**

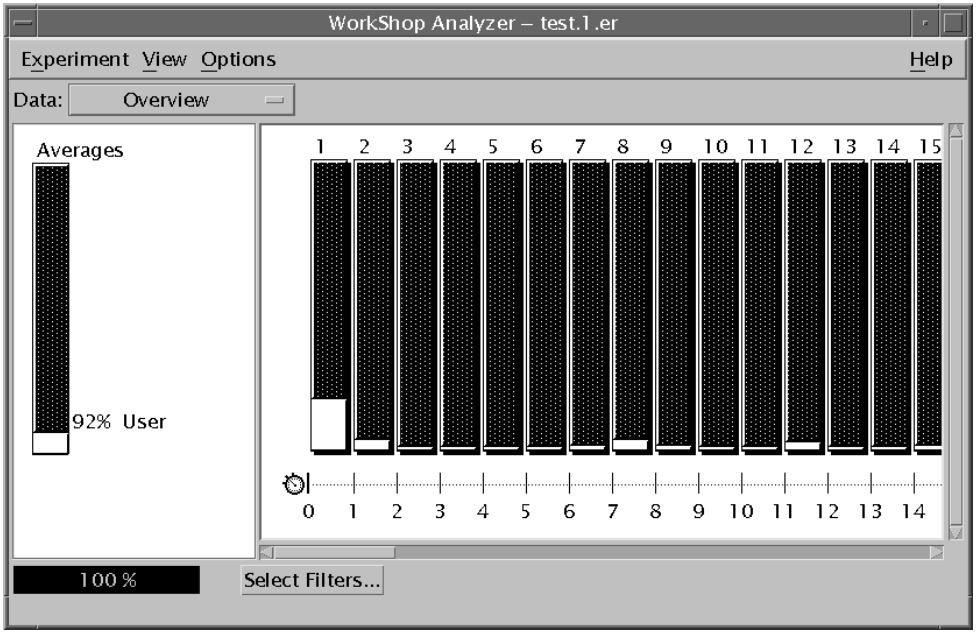

FIGURE 4-9 The Overview Display

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

- The left display pane contains a graph showing average time spent in various process states for the sample or range of samples selected.
- The right display pane contains a series of graphs showing the time spent in various process states for each sample selected for display. Each graph represents the sampling information collected by the Collector during a single sampling interval. The sample's ID number appears above the sample.

Viewing Proportional and Fixed-Width Displays

The default display for the samples in the Overview display is Fixed Width—that is, the sample graphs are all the same width, whether each sampling interval is the same length or not. You can change this to a proportional representation of the samples based on the length of each sample interval.

To switch to a proportional display:

- **Choose Options ➤ Set Overview Column Width ➤ Proportional.**

To switch back to a fixed-width display:

- **Choose Options ➤ Set Overview Column Width ➤ Fixed.**

Viewing Detailed Information About Samples

Before you can get detailed information about a sample or samples, you must select the samples. See "Selecting Samples, Threads, and LWPs" on page 66 for instructions.

The left display pane in the Overview display shows, for all the samples selected, the average time spent in various process states, and the percentage of the sampling time represented by each state. For example, over one set of samples, the program could have spent 23% of the time executing user code, 50% of the time in system wait, and 27% of the time in other states, times for which are too small to represent in the graph.

To see a more detailed analysis of the sample information, including process state metrics too small to show up in the sampling graphs:

- **Choose View ➤ Show Sample Details from the main Analyzer menu.**

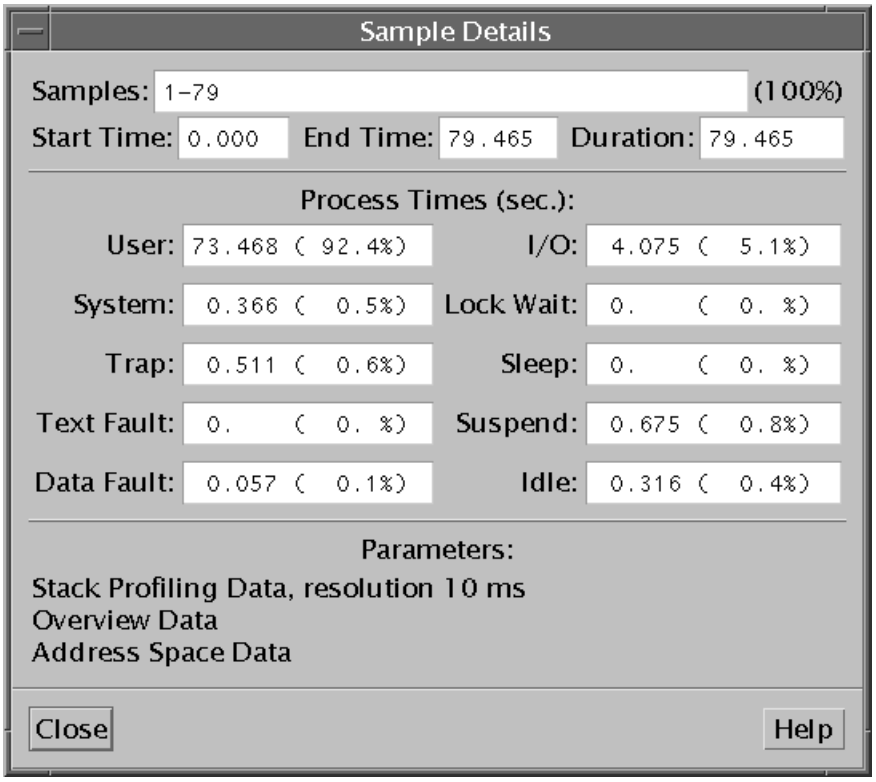

FIGURE 4-10 The Sample Details Window

The Sample Details window appears, showing the following metrics:

- The ID of the sample or samples
- The percentage of the total samples selected
- The sampling start time, end time, and duration, in seconds
- A listing of process states and the time spent in each state, represented in seconds and in percentage of the total metric for all the samples selected:
  - User
  - System
  - Trap
  - Text fault
  - Data fault
  - I/O
  - Lock wait
  - Sleep
  - Suspend
  - Idle

- A parameter list showing the types of data that the Collector recorded in the experiment record file

Examining Address-Space Information

Note – Address-space information is available to the Analyzer only if address-space data was selected when the Collector generated the experiment record. Otherwise, the Analyzer reports that address-space data is not available.

To view the Address Space display:

- **Choose Address Space from the Data option list.**

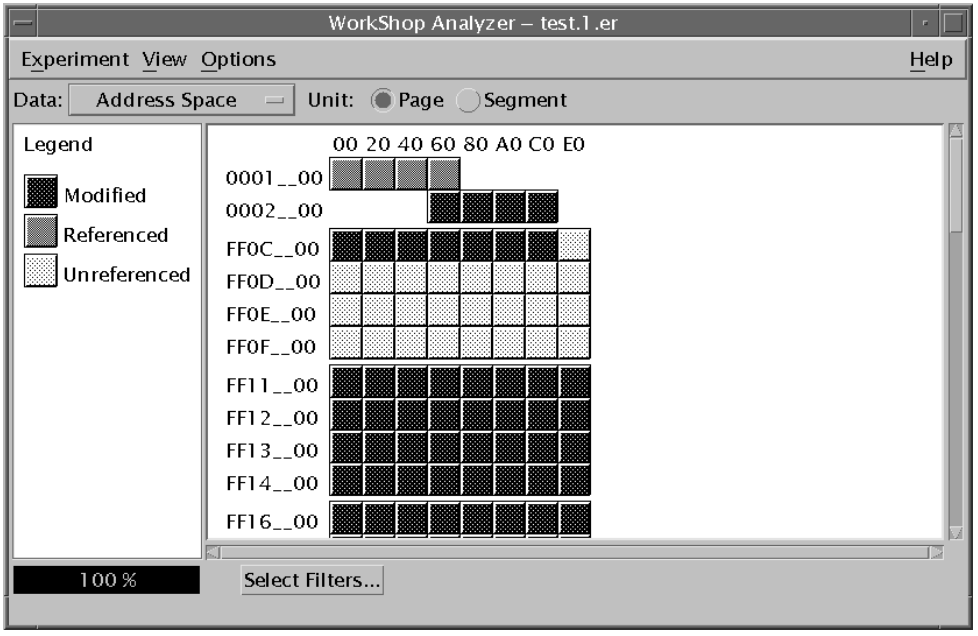

FIGURE 4-11 The Address Space Display

The Address Space display is divided into two display panes:

- The left display pane contains a legend for interpreting the graphical representation on the right.
- The right display pane shows a graphical representation of the program's address space.

The default display is by page (this display also appears if you click Page from the Unit radio buttons). Each square represents a page in the address space, and the fill pattern indicates how your program has affected the page:

- Modified it (written to it)
- Referenced it (read from it)
- Left it unreferenced

To see a display of the address-space segments:

- **Click Segment from the Unit radio buttons in the upper tool bar.**

The right display pane then shows an undifferentiated representation of the blocks of memory used by your program.

Viewing Detailed Information about Pages and Segments

To see detailed information about a page or segment:

1. **Use the Unit radio buttons to select page display or segment display.**
2. **Click the page or segment in the right Address Space display pane to select it.**

   When a page or segment is selected, a drop shadow appears behind it in the right Address Space display pane.
3. **Choose View ➤ Show Page Properties or View ➤ Show Segment Properties from the main Analyzer menu.**

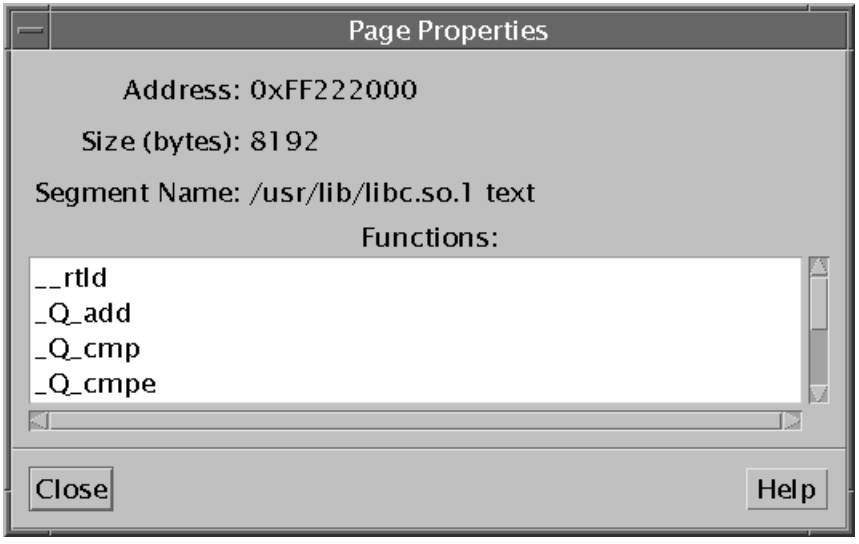

FIGURE 4-12 The Page Properties Window

The Page Properties window or the Segment Properties window appears, showing the following information:

- Address of the page or segment
- Page or segment size in bytes
- Segment name, if known
- A list of functions, if any, stored on that page or segment

Examining Execution Statistics

To examine the Execution Statistics display:

- **Choose Execution Statistics from the Data option list.**

The Execution Statistics display lists various system statistics, summed over the selected sample or samples. (For information on how to select and group samples, see "Selecting Samples, Threads, and LWPs" on page 66.)

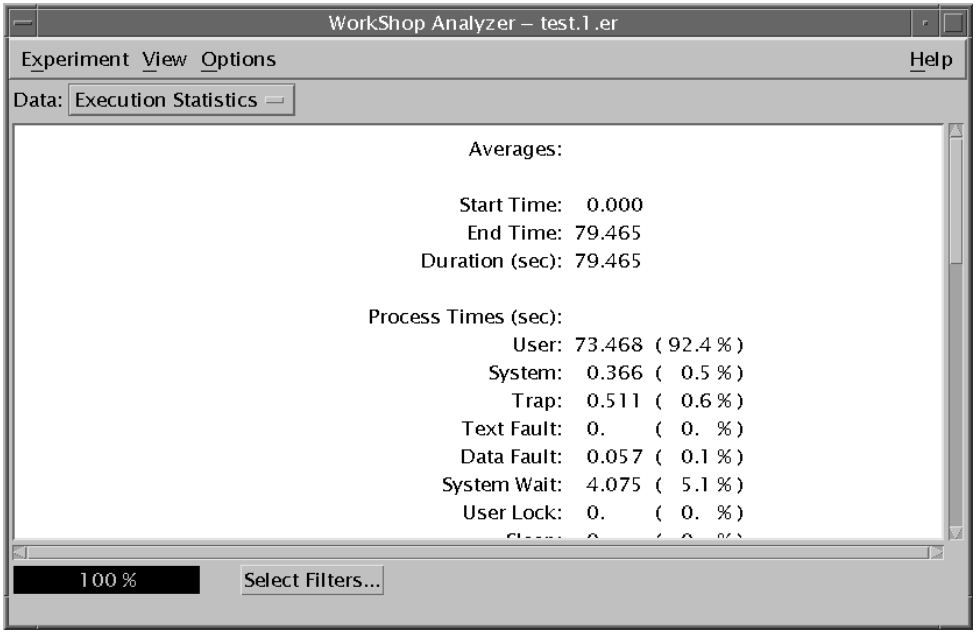

FIGURE 4-13 The Execution Statistics Display

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

Adding Experiments to the Analyzer

The Analyzer allows you to load multiple experiments. However, once you have loaded more than one experiment record into the Analyzer:

- The combined clock-based profiling, synchronization wait tracing, and hardware-counter overflow profiling data for all the experiments appear combined in the Function List display. Data for all samples from all experiment records are shown.
- Filtering by sample, thread, and LWP is disabled.
- Only the Function Display is available.

To add a new experiment record to any records already loaded into the Analyzer:

1. **Choose Experiment ➤ Add from the Analyzer main menu to open the Add Experiment dialog box.**
2. **In the Add Experiment list box, double-click the experiment-record file you want to add, or type the name of the experiment-record file in the Name text box.**
3. **Press Enter.**

Note – The Experiment ➤Add command is enabled only for the Function List display.

Dropping Experiments from the Analyzer

Note – Dropping an experiment removes it from the Analyzer, but has no effect on the experiment-record file. It is not possible to delete an experiment-record file from inside the Analyzer.

To drop an experiment record from the Analyzer:

1. **Choose Experiment ➤ Drop to open the Drop Experiment dialog box.**
2. **In the list box, click on the experiment record you want to drop from the Analyzer.**

3. **Click either Apply to drop the experiment record and leave the dialog box open, or OK to drop the experiment record and close the dialog box.**

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

Printing the Display

To print a text representation of any of the Analyzer displays:

1. **Choose Experiment ➤ Print from the main Analyzer menu to open the Print dialog box.**
2. **Use the Print To radio buttons to determine whether you are printing to a printer or a file:**
   - If you are printing to a printer, accept the default name in the Printer text box, or type in the name of a different printer.
   - If you are printing to a file, type the name of the file in the File text box, or use the browse button to open the Print to a File dialog box, in which you can navigate to a directory or file.
3. **Click the Print button.**

Note – In the case of the Overview display, what prints is not a text representation of the graphical display, but a listing of statistics for each sample in the experiment.

CHAPTER 5

er_print Reference

This chapter explains how to use the er_print utility. It covers the following topics:

- er_print Syntax
- er_print Commands

er_print is a Sun WorkShop utility that prints out an ASCII version of the various displays supported by the Analyzer. The information is written to standard output unless you redirect it to a file or printer. You must give er_print the name of one or more experiment-record files generated by the Sampling Collector as arguments. Provided the Collector has stored the data in the experiment-record file, you can display metrics of performance for functions, callers and callees, source code and disassembly listings; sampling information; address-space data; and execution statistics.

- For a description of the data collected by the Sampling Collector, see Chapter 3.
- For instructions on how to use the Analyzer to display information in a graphical format, see Chapter 4.

er_print is available with the Sun WorkShop compilers for C, C++, Fortran 77, and Fortran 95, and with the Assembler.

er_print Syntax

The command-line syntax for er_print is:

```
er_print [-script script | -command | -] exper_1 exper_2...exper_n
```

Options

`er_print` accepts the following options:

`-`	Read `er_print` commands entered at the terminal.
`-script` *script*	Read commands from the file *script*, which contains a list of `er_print` commands (see "`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*	Process the given command.

Multiple options can appear on the `er_print` command line. They are processed in the order they appear. You can mix scripts, "`-`" arguments, and explicit commands in any order. If you do not supply any command or script arguments, `er_print` acts as if you had given a "`-`" argument: It goes into interactive mode to read commands entered from the keyboard.

`er_print` Commands

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.

Function List Commands

The following commands control the display of function information.

`functions`

Write the function list with the currently selected metrics.

Default is 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.

`fsummary`

Write a summary metrics panel for each function in the function list.

For a description of the summary metrics for a function, see "Viewing Summary Metrics for a Function or Load Object" on page 57.

`metrics` *metric_spec*

Specify a new current selection of function-list metrics.

metric_spec is a list of metric keywords, separated by colons. For example:

```
% metrics i.user:i%user:e.user:e%user
```

This command instructs `er_print` to display:

- 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 has finished executing, it prints a message showing the new current metric selection. For example:

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

To see a listing of the currently selected metrics, use the `functions` command.

Note – For a list of all available `er_print` metric keywords, see TABLE 5-1 on page 86. To generate a listing of the keywords available for your experiment, use the `metric_list` or `cmetric_list` command.

`objects`

Write the load-object list with the currently selected metrics.

Default is exclusive and inclusive user CPU time, in seconds and percentage of total program metric. You can change the current metrics displayed with the `metrics` command.

`osummary`

Write a summary metrics panel for each load object in the load-object list.

For a description of the summary metrics for a load object, see "Viewing Summary Metrics for a Function or Load Object" on page 57.

`sort` *metric_keyword*

Sort the function list on the specified metric. *metric_keyword* is one of the metrics in TABLE 5-1 on page 86. For example:

```
% sort i.user
```

This command tells `er_print` to sort the function list by inclusive user CPU time.

Callers-Callees List Commands

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

`callers-callees`

Print the callers-callees panel for each of the functions, in the order in which they are sorted. The selected (middle) function is marked with an asterisk. For example:

```
Excl. User  Incl. User  Attr. User   Name
CPU sec.    CPU sec.    CPU sec.
0.          0.010       0.010        _doprnt
0.          0.          0.           _xflsbuf
0.          0.010       0.          *_realbufend
0.          0.620       0.           _rw_rdlock
0.          0.010       0.010        rw_unlock
```

In this example, `_realbufend` is the selected function; it is called by `_doprnt` and `_xflsbuf`, and it calls `_rw_rdlock` and `rw_unlock`.

`cmetrics` *metric_spec*

Specify a new current selection of caller-callee metrics.

metric_spec is a list of metric keywords, separated by colons. For example:

```
% cmetrics i.user:i%user:a.user:a%user
```

This command instructs `er_print` to display:

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

- Attributed user CPU time in seconds
- Attributed user CPU time percentage

To see a listing of the currently selected metrics, use the `callers-callees` command.

Note – For a list of all available `er_print` metric keywords, see TABLE 5-1 on page 86. To generate a listing of the keywords available for your experiment, use the `metric_list` or `cmetric_list` command.

`csort` *metric_keyword*

Sort the callers-callees display on the specified metric. *metric_keyword* is one of the keywords listed in TABLE 5-1 on page 86. For example:

```
% csort a.user
```

This command tells `er_print` to sort the callers-callees display by attributed user CPU time.

Source and Disassembly Listing Commands

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

`disasm { `*file*` | `*function*` } [`*N*`]`

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

You use the optional parameter *N* (an integer equal to 1 or greater) 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.

`source | src { `*file*` | `*function*` } [`*N*`]`

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

You use the optional parameter *N* (an integer equal to 1 or greater) 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.

Selectivity Commands: Samples, Threads, LWPs, and Load Objects

The following commands control selection of samples, threads, and LWPs for display.

`lwp_list`

Display the list of LWPs currently selected for analysis. For example:

```
% lwp_list
current: 1,3-9,17,20-38,40, total: 1-42
```

`lwp_select` *lwp_spec*

Select the LWPs about which you want to display information. *lwp_spec* is either the word `all` (to select all available LWPs), or a list of LWP ID numbers or ranges of ID numbers (*n-m*) separated by commas but no spaces. For example:

```
% lwp_select 2,4,9-11,23-32,38,40
```

`object_list`

Display the list of load objects currently selected for analysis. For example:

```
% object_list
+ /home/user/a.out
+ /usr/lib/libthread.so.1
+ /usr/lib/libc.so.1
+ /usr/lib/libdl.so.1
```

`object_select` *object_spec*

Select the load objects about which you want to display information. *object_spec* is a list of load objects, separated by commas but no spaces. If an object name itself contains a comma, you must surround the comma with double quotation marks.

The names of the objects should be either full pathnames, or the basename.

For example:

`sample_list`

Display the list of samples currently selected for analysis. For example:

```
% sample_list
current: 1,3-5,10,20-78, total: 1-78
```

`sample_select` *sample_spec*

Select the samples about which you want to display information. *sample_spec* is either the word `all` (to select all available samples), or a list of sample ID numbers or ranges of ID numbers (*n-m*) separated by commas but no spaces. For example:

```
% sample_select 1,3-5,10,20-78
```

`thread_list`

Display the list of threads currently selected for analysis. For example:

```
% thread_list
current: 1-41, total: 1-41
```

`thread_select` *thread_spec*

Select the threads about which you want to display information. *thread_spec* is either the word `all` (to select all available threads), or a list of thread ID numbers or ranges of ID numbers (*n-m*) separated by commas but no spaces. For example:

```
% thread_select all
```

Metric Commands

The following commands list available metric specification keywords.

`metric_list`

Display 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.

A keyword, except for the `size`, `address`, and `name` keywords, consists of the letter `e` (exclusive), `i` (inclusive), or `a` (attributed); a period (`.`), which indicates an absolute value, or a percent sign (`%`), which indicates percentage of total program metric; and a character string describing the metric.

Attributed metrics can be specified for display only with the `cmetrics` command, not the `metrics` command, and displayed only with the `callers-callees` command, not the `functions` command.

TABLE 5-1 lists the available `er_print` metric keywords.

TABLE 5-1 Metric Specification Keywords

Absolute Value	Percentage Value	Description
`e.user`	`e%user`	Exclusive user CPU time
`i.user`	`i%user`	Inclusive user CPU time
`a.user`	`a%user`	Attributed user CPU time
`e.wall`	`e%wall`	Exclusive wall-clock time
`i.wall`	`i%wall`	Inclusive wall-clock time
`a.wall`	`a%wall`	Attributed wall-clock time
`e.total`	`e%total`	Exclusive total LWP time
`i.total`	`i%total`	Inclusive total LWP time
`a.total`	`a%total`	Attributed total LWP time
`e.system`	`e%system`	Exclusive system CPU time
`i.system`	`i%system`	Inclusive system CPU time
`a.system`	`a%system`	Attributed system CPU time
`e.wait`	`e%wait`	Exclusive system wait time
`i.wait`	`i%wait`	Inclusive system wait time
`a.wait`	`a%wait`	Attributed system wait time

TABLE 5-1 Metric Specification Keywords *(Continued)*

Absolute Value	Percentage Value	Description
`e.text`	`e%text`	Exclusive text-page fault time
`i.text`	`i%text`	Inclusive text-page fault time
`a.text`	`a%text`	Attributed text-page fault time
`e.data`	`e%data`	Exclusive data-page fault time
`i.data`	`i%data`	Inclusive data-page fault time
`a.data`	`a%data`	Attributed data-page fault time
`e.sync`	`e%sync`	Exclusive thread synchronization wait time
`i.sync`	`i%sync`	Inclusive thread synchronization wait time
`a.sync`	`a%sync`	Attributed thread synchronization wait time
`e.syncn`	`e%syncn`	Exclusive thread synchronization wait count
`i.syncn`	`i%syncn`	Inclusive thread synchronization wait count
`a.syncn`	`a%syncn`	Attributed thread synchronization wait count
`size`		Function size, in bytes
`address`		Address of the function in memory
`name`		Function name

`cmetric_list`

Display 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.

A keyword, except for the `size`, `address`, and `name` keywords, consists of the letter `e` (exclusive), `i` (inclusive), or `a` (attributed); a period (`.`), which indicates an absolute value, or a percent sign (`%`), which indicates percentage of total program metric; and a character string describing the metric

Note – Attributed metrics can be specified for display only with the `cmetrics` command, not the `metrics` command, and displayed only with the `callers-callees` command, not the `functions` command.

TABLE 5-1 on page 86 lists the available `er_print` metric keywords.

Output Commands

The following commands control `er_print` output.

`limit` *n*

Limit output to the first *n* entries of the report; *n* is an unsigned integer 1 or higher.

`name { long | short }`

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

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

Close any open output file, then open *filename* for subsequent output.

If you specify a dash (`-`) instead of *filename*, output is written to standard output.

Miscellaneous Commands

`address_space`

Display address-space data for the current experiment.

`header`

Display descriptive information about the current experiment.

`help`

Print help information.

`mapfile` *load-object* `{` *mapfilename* `| - }`

Write the mapfile for the specified load object to `mapfilename`. If you specify a dash (`-`) instead of *mapfilename*, `er_print` writes the mapfile to standard output.

`overview`

Write out the overview data of each of the currently selected samples.

`quit`

Exit `er_print`.

`script` *script*

Process additional commands from the script file *script*.

`statistics`

Write out execution statistics, aggregated over the current sample set.

`{ Version | version }`

Print the current release number of `er_print`.

CHAPTER 6

Advanced Topics: Understanding the Sampling Analyzer and Its Data

The Sampling Analyzer reads the data collected by the Sampling Collector and converts the data to performance metrics, which are computed against various elements in the structure of the target program. 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.

This chapter covers the following topics:

- Event-Specific Data and What It Means
- Call Stacks and Program Execution
- Mapping Addresses to Program Structure
- Annotated Source Code and Disassembly Code
- Understanding Performance Costs

Event-Specific Data and What It Means

The event-specific data for each event recorded contains a high-resolution timestamp, a thread ID, and an LWP ID. The timestamp can be used to select only part of a run, while the thread and LWP IDs can be used to select a subset of threads and LWPs. In addition, each event generates specific raw data, which is described in the following sections:

- "Clock-Based Profiling"
- "Synchronization Wait Tracing"
- "Hardware-Counter Overflow Profiling"

Clock-Based Profiling

Clock-based profiling data consists of a set of tick-counts delivered with each profiling signal to each LWP. There is a separate tick for each of the microaccounting states maintained by the kernel. Some of those states are aggregated into System CPU Time, while others are aggregated into System Wait Time. The remaining states are presented individually in the Analyzer.

When the LWP is in user-mode in the CPU, the tick array delivered typically contains a 1 (one) for the User-CPU state, and zeros for all the other states. When the LWP is in one of the other states, the ticks are accumulated, but a profile signal is not sent until the process returns to user-CPU state.

Since the ticks are integer counts, each representing one profile interrupt interval, while LWP scheduling is done at finer granularity, there is an inherent uncertainty in the state as attributed in the profile packet. Typically, the total LWP time, computed by summing all the ticks in all states, is accurate to a few tenths of a percent, as compared with the values returned by `gethrtime()` in the process. The CPU time may vary by several percentage points, compared with values returned by `gethrvtime()` in the process. Under heavy load, the variation may be even more pronounced. However, the CPU time differences do not represent a systematic distortion, and the relative times reported for different routines, source-lines, and such are not substantially distorted.

For information about `gethrtime()` and `gethrvtime()`, see the man pages for these functions.

Note – Be careful in comparing the LWP times reported in the Analyzer with the numbers from `vmstat`. The Analyzer times represent the sum of the various microstate accounting times during the lifetime of each LWP, whereas vmstat reports times summed over physical CPUs. If, for example, the target process has many more LWPs than the system on which it is running has CPUs, the Analyzer shows much more wait time than `vmstat` reports. In the simplest such case, with two CPU-bound LWPs and one physical CUP, the Analyzer reports the sum of the two LWPs, as well as each LWP separately, as having approximately 50% wait (idle) time; `vmstat` reports no idle time. The CPU is busy all the time, but each LWP is spending half its time waiting while the other LWP is running.

Synchronization Wait Tracing

Synchronization wait tracing events are collected by tracing calls to the functions in the threads library. The event 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). Only

events for which the difference between request and grant times exceeds the specified threshold are recorded. Synchronization trace data is accurate to within a few tenths of a percent, compared with time stamps recorded in the process itself.

Hardware-Counter Overflow Profiling

Hardware-counter overflow profiling allows you to specify a hardware counter and an overflow value (number of increments) for that counter on the CPU on which a given LWP is running. Hardware counters typically tally instruction-cache misses, data-cache misses, clock ticks, instructions executed, and the like. When the designated counter reaches the overflow value, the Collector records the call stack for the LWP and includes a timestamp and the IDs of the LWP and the thread running on it. You can access this data in the Analyzer and use it to support count metrics.

Hardware counters are system-specific, so the choice of counters available to you depend on the system you are using. Many systems do not support hardware-counter overflow profiling. On these machines, the feature is disabled.

Call Stacks and Program Execution

A call stack is a series of program 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. The process of recording a call stack is referred to as "unwinding the stack" and is described in "Unwinding the Stack" on page 98.

The leaf PC in a call stack is used to attribute exclusive metrics from the performance data to the function in which that PC is found. All the other PCs on the stack are used to attribute inclusive metrics to the function in which they are found.

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 Analyzer's reported metrics correspond directly to those functions. Sometimes, however, the actual execution of the program may not correspond to a simple intuitive model of how the program would execute, and the Analyzer's reported metrics may be confusing. See "Mapping Addresses to Program Structure" on page 99 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 in the function `_start()`, built into every executable.

When the program runs, each instruction executes in sequence until an instruction is encountered that represents a call, jump, or branch. At that point, control is transferred to the address given by the target of the branch, and execution proceeds from there.

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 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 call 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.

Signals

When a signal is sent to a process, various register and stack operations occur that make it look as though the leaf PC at the time of the signal is the return address for a call to a system routine, `sigacthandler()`. `sigacthandler()` calls the user-specified signal handler just as any function would call another. The Analyzer treats the stack frames for such calls normally, although the stack frames can make it look as though any instruction can generate a call.

Fast Traps

Some instructions trap into the kernel and then are passed back to user mode in a lightweight version of signals, known as fast traps. The Analyzer knows about one of these, the exception for misaligned integer memory references in SPARC-v9. In that case, frames for the misaligned integer trap appear, as if the trapping instruction called the handler.

Kernel Traps

Some instructions trap into the kernel, and are emulated in the kernel. One example is the `fitos` instruction on the UltraSPARC-III platform, which converts a large integer to single-precision floating point. No special handling is done in the Analyzer, but the instruction following the trapping instruction appears to take a long time, because it cannot issue until the kernel is through.

Tail-Call Optimization

One particular optimization can be done whenever the last thing a particular routine does is to call another routine. Rather than actually making the call and then popping the frame from the stack and returning, the caller pops the stack and then calls its callee. The motivation for this optimization is to reduce the size of the stack, and, on SPARC machines, to reduce the use of register windows.

In effect, your program source implies that it behaved like this:

```
A -> B -> C -> D
```

But when `B` and `C` are tail-call optimized, the call stack looks as if the program is doing this:

```
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 O4 or higher. When code is compiled without the `-g` option, tail-call optimization takes place at O2 or higher.

Explicit Multithreading

A simple program executes in a single thread, on a single LWP (light-weight process). Multithreaded executables make calls to a thread creation routine, which creates additional LWPs to run the threads. The operating system controls the assignment of LWPs to CPUs for execution, while the threads library controls the scheduling of threads onto LWPs. Newly created threads begin execution at a routine called `_thread_start()`, which calls the function passed in the thread creation call. Threading can be done with either bound threads, where each thread is bound to a specific LWP, or with unbound threads, where each thread may be scheduled on a different LWP at different times.

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 only for Fortran 95. You might want to refer to the chapters on parallelization and OpenMP in the *Fortran Programming Guide* for background on parallelization strategies and OpenMP directives.)

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.

When the compiler encounters a parallel construct, it sets up the code for parallel execution by placing the body of the construct in a separate *body function* and replacing the construct with a call to a microtasking library routine. The microtasking library routine is responsible for dispatching threads to execute the body function. The address of the body function is passed to the microtasking library routine as an argument.

- If the parallel construct is delimited with one of the following:
  - The Sun directive `c$par doall`
  - The Cray directive `c$mic doall`
  - An OpenMP `c$omp PARALLEL`, `c$omp PARALLEL DO`, or `c$omp PARALLEL SECTIONS` directive

  then the construct is replaced with a call to the microtasking library routine `__mt_MasterFunction_()`. A loop that is parallelized automatically by the compiler is also replaced by a call to `__mt_MasterFunction_()`.
- If a `c$omp PARALLEL` construct contains one or more worksharing `c$omp DO` or `c$omp SECTIONS` directives, each worksharing construct is replaced by a call to the microtasking library routine `__mt_Worksharing_()`.

The compiler assigns names to body functions of the form:

```
_$1$mf_string1_$namelength$functionname$linenumber$string2
```

- *string1* denotes the type of parallel construct, either `parallel`, `sections`, `doall`, or `DOALL`.
- *namelength* is the number of characters in *functionname.*
- *functionname* is the name of the function from which the construct was extracted, usually ending in an underscore.
- *linenumber* is the line number of the construct in the original source.
- *string2* is related to the name of the source file.

To make the data easier to analyze, the Analyzer provides these functions with a more readable name, in addition to the compiler-generated name.

At run time, initially only the main thread executes. The first time it executes a call to `__mt_MasterFunction_()`, `__mt_MasterFunction_()` initiates the creation of multiple worker threads, the number based on the value specified by the environment variable `PARALLEL` or `OMP_NUM_THREADS`, or by a call to the OpenMP run-time routine omp_`set_num_threads()`. Thereafter `__mt_MasterFunction_()` manages the distribution of available work among the master thread and the worker threads.

In the main thread, `__mt_MasterFunction_()` calls a sequence of dispatcher functions that eventually call the body function. (This is also the behavior you see for code compiled for parallelization but running on a single-CPU machine, or on a multiprocessor machine using only one thread.)

Worker threads are created using the Solaris threads library. The call stack for a worker thread begins with the threads library routine `_thread_start()`. `_thread_start()` makes a call to `__mt_SlaveFunction_()`, which the thread continues to execute during its lifetime. `__mt_SlaveFunction_()` calls `__mt_WaitForWork_()`, in which the thread waits for available work. When work becomes available, the thread returns to `__mt_SlaveFunction_()`, which then initiates a call to the body function. When the work is finished, the worker thread returns to `__mt_SlaveFunction_()`, which calls `__mt_WaitForWork_()` again. You can observe the control flow for a thread in the Analyzer Callers-Callees window. See "Examining Caller-Callee Metrics for a Function" on page 60 for information about how to use this window.

Note – In these call sequences, the Analyzer shows an imputed call to the function from which the compiler-generated body functions were extracted. This call is inserted as if the original function called the compiler-generated body functions, so that inclusive data is reported against the original function.

Worker threads typically use CPU time while they are in `__mt_WaitForWork_()` in order to reduce latency when new work arrives, that is, when the main thread reaches a new parallel construct. (This is known as a *busy-wait*.) However, you can set an environment variable to specify a sleep wait, which shows up in the Analyzer as LWP time, but not CPU time. There are generally two situations where the worker threads spend time waiting for work, where you might want to redesign your program to reduce the waiting:

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

By default, the microtasking library uses threads that are bound to LWPs. You can override this default by setting the `FLAG` variable in the makefile to `UNBOUND` before you build the program, or by setting the environment variable `MT_BIND_LWP` to `FALSE`.

Loops with a `long long` index call somewhat different microtasking library routines than loops with an `integer` or `long` index.

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

Unwinding the Stack

When the Collector records an event, it records the call stack of the process at the time of the event. The call stack recorded consists of the address of the next instruction to be executed (the leaf PC), the contents of the return register, and the contents of the return address on each frame of the stack, eventually reaching the address of the call instruction in `_start()` for the main thread and `_thread_start()` for the worker threads.

The Collector always records the return register, and the Analyzer uses a heuristic to determine whether or not the return address has been pushed on the stack. If it has, the return register is ignored; if it has not, the return register is used as the calling PC. A specific register, known as the frame pointer, is used to find the first frame on the stack; each frame contains a previous frame pointer used to find the frame of its caller. On Intel machines, for optimized code, a previous frame pointer is not maintained in each stack frame, and a heuristic is used to unwind the stack.

Mapping Addresses to Program Structure

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

The Process Image

When a program is run, a process is instantiated from the executable for that program. The process has a number of regions in its address space, some of which are text and represent executable instructions, and some of which are data 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 may correspond to different functions at different times during the run. In addition, different PCs may 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, a data section for data, and various symbol tables. All load objects must contain an ELF symbol table, which gives the names and addresses of all the globally-known functions in that object. Load objects compiled with the `-g` option contain additional symbolic information, which can augment the ELF symbol table and provide information about functions that are not global, additional information about object modules from which the functions came, and line number information relating addresses to source lines.

The term function is used to describe a set of instructions that represent a high-level operation described in the source code. The term covers subroutines as used in Fortran, methods as used in C++, and the like. Functions are described cleanly in the source code, and normally their names appear in the symbol table representing a set of addresses; if the program counter is within that set, the program is executing within that function.

In principle, any address within the text segment of a load object can be mapped to a function. Exactly the same mapping is used for the leaf PC and all the other PCs on the call stack. Most of the functions correspond directly to the source model of the program. Some do not; these functions are described in the following sections:

Aliased Functions

Typically, functions are defined as global, meaning that their names are known everywhere in the program. The name of a global function must be unique within the executable. If there is more than one global function of a given name within the address space, the runtime linker resolves all references to one of them, and the others are never executed, and so do not appear in the function list. From the Summary Metrics window, you can see the shared object and object module that contain the selected function.

Under various circumstances, a function may 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 typically the same as the corresponding weak name, except that it has a leading underscore. Many of the functions in the thread library also have alternate names for pthreads and Solaris threads, as well as strong and weak names and alternate internal symbols. In all such cases, only one name is used in the function list of the Analyzer. The name chosen is the last symbol at the given address in alphabetic order. This choice most often corresponds to the name that the user would use. In the Summary Metrics window, all the aliases for the selected function are shown.

Non-Unique Function Names

While aliased functions reflect multiple names for the same piece of code, there are circumstances under which multiple pieces of code have the same name:

- Sometimes, for reasons of modularity, functions are defined as static, meaning that their names are known only in some parts of the program (typically a single compiled object module). In such cases, the Analyzer may see several functions of the same name referring to quite different parts of the program. In the Summary Metrics window, the object module name for each of these functions is given to distinguish them from one another. In addition, any selection of one of these functions can be used to show the source, disassembly, and the callers and callees of that specific function.
- Sometimes a program may use wrapper or interposition functions that have the weak name of a function in a library and supersede calls to that library function. Some wrapper functions call the original function in the library, in which case both instances of the name appear in the Analyzer function list. Such functions come from different shared objects and different object modules, and can be distinguished from each other in that way. The Collector also wraps some library functions, and both the wrapper function and the real function can appear in the Analyzer.

Static Functions from Stripped Shared Libraries

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

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 the the main body of code.

Different compilers order the pieces of a Fortran subroutine with alternate entry points differently. The prologue code for each entry point always corresponds to a region of text that has the name of that entry point, but the code for the main body of the routine can receive either of the two entry point names.

- On the SPARC platform, the WS 6 compilers generate the alternate entry point prologue first, and the main entry point and body code second, and all metrics from the main body receive the name of the main entry point.
- On the x86 platform, the compilers generate the prologue for the main entry point first, then the prologue for the alternate entry point and the main body. Metrics for the main body are associated with the alternate entry point name. 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.

Inlined Functions

An inlined function is code defined as a function in the source, which compiles to instructions that are inserted at the call site of the function, instead of an actual call. There are two kinds of inlining, both of which will affect the Analyzer:

- C++ inline function definitions
- Explicit or automatic inlining performed at high optimization (levels O4 and O5)

Both versions are done to improve performance.

To specify C++ inlining, either include the body of a method in the class definition for that method, or tag the method explicitly as being an inline function. 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. Normally, when you compile with the `-g` option, even the functions defined as being inlined are compiled as normal functions. However, if you compile C++ with `-g0`, even with no other optimizations, all functions defined as inlined are compiled as such.

Explicit and automatic inlining is performed at high optimization, 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 that can be subject to register usage and instruction scheduling optimizations.

Both kinds of inlining have the same effect on the function list. Functions that appear in the source code but have been inlined do not show up in the function list, and metrics that would normally be thought of as inclusive metrics at the call site of the inlined function (representing time spent in the called function) are actually shown as exclusive metrics (representing the instructions of the inlined function, attributed to the call site).

Note – In many cases, inlining can make data difficult to interpret, so you might want to disable inlining when you measure performance.

In some cases, even when a function is inlined, a so-called out-of-line function is left. Sometimes some call sites do call the out-of-line version, but others have the instructions inlined. In such cases, the functions may appear in the function list, as if they were never inlined, but the metrics attributed to them reflect 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 do not explicitly appear in the source model of the program. These functions are described in more detail in "Parallel Execution and Compiler-Generated Body Functions" on page 96.

The Analyzer shows these functions as normal functions, and assigns a label 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.

When a function containing parallel loops is inlined, the names of its compiler-generated body functions reflect the function into which it was inlined, not the original function.

Outline Functions

Outline functions can be created during feedback optimization. They represent code that is not normally expected to be executed. Specifically, it is code that is not executed during the "training run" used to generate the feedback. To improve paging and instruction-cache behavior, such code is moved elsewhere in the address space, and is made into a function with a name of the form:

_1outline*string1*$*namelength*$*functionname*$*linenumber*$*string2*

- *string1* is related to the specific section of outlined code.
- *namelength* is the number of characters in *functionname.*
- *functionname* is the name of the function from which the construct was extracted.
- *linenumber* is the line number of the construct in the original source.
- *string2* is related to a compiler internal name.

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.

As with compiler-generated body functions, the Analyzer displays an imputed call from the function from which the outline function is derived.

The <Unknown> Function

Under some circumstances, a PC does not map to a known function. In such cases, the PC is mapped to the special function named <Unknown>.

The following circumstances will show PCs mapping to <Unknown>:

- When the PC belongs to the dynamic linker, `ld.so`.
- When the PC corresponds to the PLT (Program Linkage Table) in a load object. This happens whenever a function in one load object calls a function in a different shared object. The actual call transfers first to a three-instruction sequence in the PLT, and then to the real destination.
- When the PC corresponds to an address in the data section of the executable or a shared object; normally data is not executable, so a data address never appears in a call stack. Sometimes, however, if code is self-modifying, the program writes these writable instructions into the program data space before executing them. The SPARC v7 version of `libc.so` has several functions (`.mul` and `.div`, for example) in its data section (the code is in the data section so that it can be dynamically rewritten to use machine instructions when the library detects that it is executing on a SPARC v8 or v9 machine).
- 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.)

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.

The Callers-Callees Window

This section discusses the Callers-Callees window, and how the program execution is reflected in that window.

The <Total> Function

The special function <Total> 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.

Fortran Alternate Entry Points

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. In any case, the collected data does not allow the Analyzer to distinguish between calls to the main entry point and calls to the alternate entry point.

Likewise, all calls from the subroutine are shown as being made from the name associated with the main body of the subroutine.

Inlined Functions

Inlined functions do not show up as callees of the routines into which they have been inlined. Be careful of interpreting data for functions that are inlined in some places, but appear as normal functions elsewhere. Only the metrics on the regular function show up in the Analyzer, and this usage may represent a small fraction of the total metrics for all the instances of that function, inlined and normal.

Compiler-Generated Body Functions

Compiler-generated body functions are directly called by routines in the microtasking library, as described in “Parallel Execution and Compiler-Generated Body Functions” on page 96. However, in order to make the behavior shown in the Analyzer more closely related to the source model of execution, the Analyzer imputes an artificial call from the function from which the loop routine was extracted, at the line from which it was extracted. Thus in the Analyzer, the function from which a body routine was extracted appears as the caller, and inclusive time propagates up to it.

Outline Functions

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 model, the Analyzer imputes an artificial call from the main routine to its outline portion.

Tail-Call Optimization

Intermediate calls that have been tail-call optimized may not appear explicitly in the Callers-Callees window.

Signals

The Analyzer treats the frames resulting from signal delivery as ordinary frames. The user code at the point at which the signal was delivered is shown as "calling" the system routine `sigacthandler()`, and it in turn is shown as calling the user's signal-handler. Inclusive metrics from both `sigacthandler()` and any user signal handler, and any other functions they call, appear as inclusive metrics for the interrupted routine.

Stripped Static Functions

Stripped static functions are shown as called from the correct caller, except when the PC from the static function is a leaf PC that appears after the save instruction in the static function. Without the symbolic information, the Analyzer does not know the save address, and cannot tell whether to use the return register as the caller. It always ignores the return register. Since several functions can be coalesced into a single `<static>@0x12345` function, the real caller or callee might not be distinguished from the adjacent routines.

The `<Unknown>` Function

Callers and callees of the `<Unknown>` function represent the previous and next PCs in the call stack, and are treated normally.

Recursive Calls

A recursive call is one in which a function calls itself. In the Callers-Callees window, the recursive function is shown as a caller of itself, but not as a callee.

Annotated Source Code and Disassembly Code

The annotated source code and disassembly code features of the Analyzer are useful for which operations within a function are causing poor performance.

Annotated Source Code

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

The compilation process goes through many stages, depending on the level of optimization requested, and transformation take place which may confuse the mapping of instructions to source lines. For some optimizations, source line information may be completely lost, while for others, it may be confusing. The compiler relies on various heuristics to track the source line for an instruction, and these heuristics are not infallible.

The four types of metrics that can appear on a line of annotated source code are explained in TABLE 6-1.

TABLE 6-1 Annotated Source-Code Metrics

Metric	Significance
(Blank)	No PC in the program corresponds to this line of code. This should always happen for comment lines. It also happens for apparent code lines in the following circumstances: • All the instructions from the apparent piece of code have been optimized away. • The code is repeated elsewhere, and the compiler did common subexpression recognition and tagged all the instructions with the lines for the other copy. • The compiler simply mistagged the instruction that really came from that line with an incorrect line number.
`0.`	Some PCs in the program were tagged as derived from this line, but there were 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 mean that the line was not executed, only that it did not show up statistically in a profile and that a thread-synchronization call from that line never had a delay exceeding the threshold.
`0.000`	At least one PC from this line appeared in the data, but the computed metric value rounded to zero.
`1.234`	The metrics for all PCs attributed to this line added up to the non-zero numerical value shown.

Compiler Commentary

Various parts of the compiler can incorporate commentary into the executable. Each comment is associated with a specific line of source.

Some of the commentary is inserted by the `f95` compiler, reflecting potential performance costs attributable to copy-in and or copy-out required to pass an array section to a subroutine. When code is compiled for parallel analysis, additional commentary reflecting the parallelization state of loops is inserted.

When the annotated source is written, the compiler commentary for any source line appears immediately preceding the source line.

The Unknown Line: `<sum of all instructions without line numbers>`

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

Common Subexpression Elimination

One very common optimization recognizes that the same expression appears in more than one place, and that performance can be improved by generating the code for that expression in one place. For example, if the same operation appears in both the `if` and the `else` branches of a block of code, the compiler can move that operation to just before the `if` statement. When it does so, it assigns line numbers to the instructions based on one of the previous occurrences of the expression. If the line numbers assigned correspond to one branch of an `if` structure, and the code actually always takes the other branch, the annotated source might show metrics on lines within the branch that is not taken.

Annotated Disassembly

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

- If the object module is not known, the Analyzer disassembles the instructions for just the specified function, and does not show any source lines within the disassembly.
- If the object module is known, the disassembly covers all functions within the object module.
- If the source file is available, and line number data is recorded, the Analyzer interleaves the source with the disassembly.
- If the compiler has inserted any commentary into the object code, it too, is interleaved in the annotated disassembly.

When code is not optimized, line numbers are simple, and the interleaving of source and disassembled instructions appears natural. When optimization takes place, instructions from later lines sometimes appear before those from earlier lines. The Analyzer's algorithm for interleaving is that whenever an instruction is shown as coming from line *N*, all source lines up to and including line *N* are written before the instruction. Compiler commentary associated with line *N* of the source are written immediately before that line.

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
- The assembler ASCII representation of the instruction

Where possible, call addresses are resolved to symbols. Metrics are shown on the lines for instructions, but not on any interleaved source or commentary. Possible metric values are as described for source-code annotations, in TABLE 6-1 on page 108.

Understanding Performance Costs

You can examine metric values at the function level, the source-line level, or the disassembly instruction level. High metric values at each of these levels reveal different ways in which you can refine your code to make it more efficient

Performance at the Function-Level

Functions have high metric values either because they are being executed many times, or because each execution of the function takes a long time.

- If the function is being executed many times, the performance-improvement opportunities lie in reducing the number of calls, or in inlining the function.
- If each execution of a function takes a long time, the performance improvement opportunities lie in making the function's algorithms more efficient.

It is usually easiest to identify opportunities for increasing performance efficiency by examining the annotated source of the function.

Performance at the Source Line Level

Lines that have high metric values in the annotated source represent the places in the function where most of the execution time is being spent. Performance improvement opportunities lie in improving or rewriting the algorithm, or increasing the optimization level for the function. Where the algorithm seems efficient and well-optimized, performance improvement opportunities can be identified by looking at the annotated disassembly.

Performance at the Instruction Level

Typically, the burden of generating efficient code at the instruction level is on the compiler. Sometimes, specific leaf PCs appear more frequently because the instruction that they represent is delayed before issue. Sometimes a specific leaf PC appears because the previous instruction takes a long time to execute and is not interruptible, for example when an instruction traps into the kernel.

There are several causes of instruction issue delays, and each represents a potential opportunity for improving performance. Instructions issue delays can be caused by an arithmetic instruction needing a register that is not available because the register contents were set by an earlier instruction that has not yet completed. Two examples of this sort of delay are load instructions that have data cache misses, and floating-point arithmetic instructions that require more than one cycle to execute, such as floating-divide.

Instructions may also seem overrepresented because the instruction cache does not include the memory word that contains the instruction. Instructions may seem underrepresented because they are always issued in the same clock as the previous instruction, so they never represent the next instruction to be executed.

CHAPTER 7

Loop Analysis Tools

The Fortran and C compilers automatically parallelize loops for which they determine that it is safe and profitable to do so. LoopTool is a performance analysis tool that reads loop timing files created by these compilers. LoopTool uses a graphical user interface (GUI). LoopReport is the command-line version of LoopTool.

This chapter covers the following topics:

- Basic Concepts
- Setting Up Your Environment
- Creating a Loop Timing File
- Starting LoopTool
- Using LoopTool
- Starting LoopReport
- Compiler Hints
- How Optimization Affects Loops

Basic Concepts

LoopTool and LoopReport enable you to:

- Time all loops, whether serial or parallel.
- Produce a table of loop timings.
- Collect hints from the compiler during compilation.

LoopTool displays a graph of loop runtimes showing which loops were parallelized. You can go directly from the graphical display of loops to the source code for any loop you want, so you can edit your source code while in LoopTool.

LoopReport reports loop runtimes in an ASCII file instead of a graphical display.

There are four basic steps for using LoopTool and LoopReport:

1. Setting up environment variables

2. Compiling the program with the options required to create a timing file for loop analysis
3. Running the program to generate a timing file
4. Invoking LoopTool or LoopReport on the timing file

Note – The examples in this section use the Fortran (f77 and f95) compilers. The options shown (such as `-xparallel`, `-Zlp`) also work for C.

Setting Up Your Environment

Before running an executable compiled with `-Zlp`, set the environment variable `PARALLEL` to the number of processors on your machine.

The following command makes use of `psrinfo`, a system utility. *Note the backquotes:*

```
% setenv PARALLEL ‘/usr/sbin/psrinfo | wc -l‘
```

You may want to put this command in a shell startup file (such as `.cshrc` or `.profile`).

Creating a Loop Timing File

To create a loop timing file, you compile your program with compiler options that automatically parallelize and optimize your code (`-xparallel` and `-xO4`). You also add the `-Zlp` option to compile for LoopTool or LoopReport. When you run the program compiled with these options, Sun WorkShop creates a timing file for LoopTool or LoopReport to process.

The three compiler options are illustrated in this example:

```
% f77 -xO4 -xparallel -Zlp source_file
```

Note – All examples apply to FORTRAN 77, Fortran 95, and C programs.

There are a number of other useful options for looking at and parallelizing loops. TABLE 7-1 lists these options.

TABLE 7-1 Compilation Options

Option	Effect
`-o` *program*	Renames the executable to *program*
`-xexplicitpar`	Parallelizes loops marked with `DOALL` pragma
`-xloopinfo`	Prints hints to `stderr`

Other Compilation Options

Many combinations of compiler options work for LoopTool and LoopReport.

To compile for automatic parallelization, typical compiler options are `-xparallel` and `-x04`. To compile for LoopTool and LoopReport, add `-Zlp`.

```
% f77 -x04 -xparallel -Zlp source_file
```

You can use either `-x03` or `-x04` with `-xparallel`. If you do not specify `-x03` or `-x04` but you do use `-xparallel`, then the compiler uses `-x03`. TABLE 7-2 summarizes how optimization level options are added for specific options.

TABLE 7-2 Optimization Level Options and What They Imply

You type:	Expanded To:
`-xparallel`	`-xparallel -x03`
`-xparallel -Zlp`	`-xparallel -x03 -Zlp`
`-xexplicitpar`	`-xexplicitpar -x03`
`-xexplicitpar -Zlp`	`-xexplicitpar -x03 -Zlp`
`-Zlp`	`-xdepend -x03 -Zlp`

Other compilation options include `-xexplicitpar` and `-xloopinfo`.

The Fortran compiler option `-xexplicitpar` is used with the pragma `DOALL`. If you insert `DOALL` before a loop in your source code, you are explicitly marking that loop for parallelization. The compiler parallelizes the loop when you compile with `-xexplicitpar`.

The following code fragment shows how to mark a loop explicitly for parallelization.

```
    subroutine adj(a,b,c,x,n)
      real*8 a(n), b(n), c(-n:0), x
        integer n
c$par DOALL
    do 19 i = 1, n*n
      do 29 k = i, n*n
        a(i) = a(i) + x*b(k)*c(i-k)
29    continue
19  continue
    return
    end
```

When you use `-Zlp` by itself, `-xdepend` and `-xO3` are added. The `-xdepend` option instructs the compiler to perform the data dependency analysis that it needs to do to identify loops. The option `-xparallel` includes `-xdepend`, but `-xdepend` does not imply (or trigger) `-xparallel`.

The `-xloopinfo` option prints hints about loops to `stderr` (the UNIX standard error file, on file descriptor 2) when you compile your program. The hints include the routine names, the line number for the start of the loop, whether the loop was parallelized, and the reason it was not parallelized, if applicable.

The following example redirects hints about loops in the source file `gamteb.F` to the file `gamtab.loopinfo`:

```
% f77 -xO3 -parallel -xloopinfo -Zlp gamteb.F 2> gamteb.loopinfo
```

The main difference between `-Zlp` and `-xloopinfo` is that in addition to providing compiler hints about loops, `-Zlp` also instruments your program so that timing statistics are recorded at runtime. For this reason, also, LoopTool and LoopReport analyze only programs that have been compiled with `-Zlp`.

Running the Program

After compiling with `-Zlp`, run the executable. This creates the loop timing file, *program*`.looptimes`. Both LoopTool and LoopReport process two files: the instrumented executable and the loop timing file.

Starting LoopTool

You can start LoopTool by giving it the name of a program (an executable) to load:

```
% looptool program &
```

If you start LoopTool without specifying a file, the Open File dialog box opens, which allows you to select a file to examine:

```
% looptool &
```

LoopTool reads the timing file associated with your program. The timing file contains information about loops. Typically, this file has a name of the format *program*.`looptimes` and is in the same directory as your program.

By default, LoopTool looks in the executable's directory for a timing file. Therefore, if the timing file is there (the usual case), you do not need to specify where to look for it:

```
% looptool program &
```

If you name a timing file on the command line, then LoopTool and LoopReport use that file.

```
% looptool program program.looptimes &
```

If you use the command line option `-p`, LoopTool and LoopReport check for a timing file in the directory indicated by `-p`:

```
% looptool -p timing_file_directory program &
```

If the environment variable `LVPATH` is set, the tools check that directory for a timing file.

```
% setenv LVPATH timing_file_directory
% looptool program &
```

Using LoopTool

The main window displays the runtimes of your program's loops in a bar chart arranged in the order that the source files were presented to the compiler.

FIGURE 7-1 shows the components of the LoopTool main window.

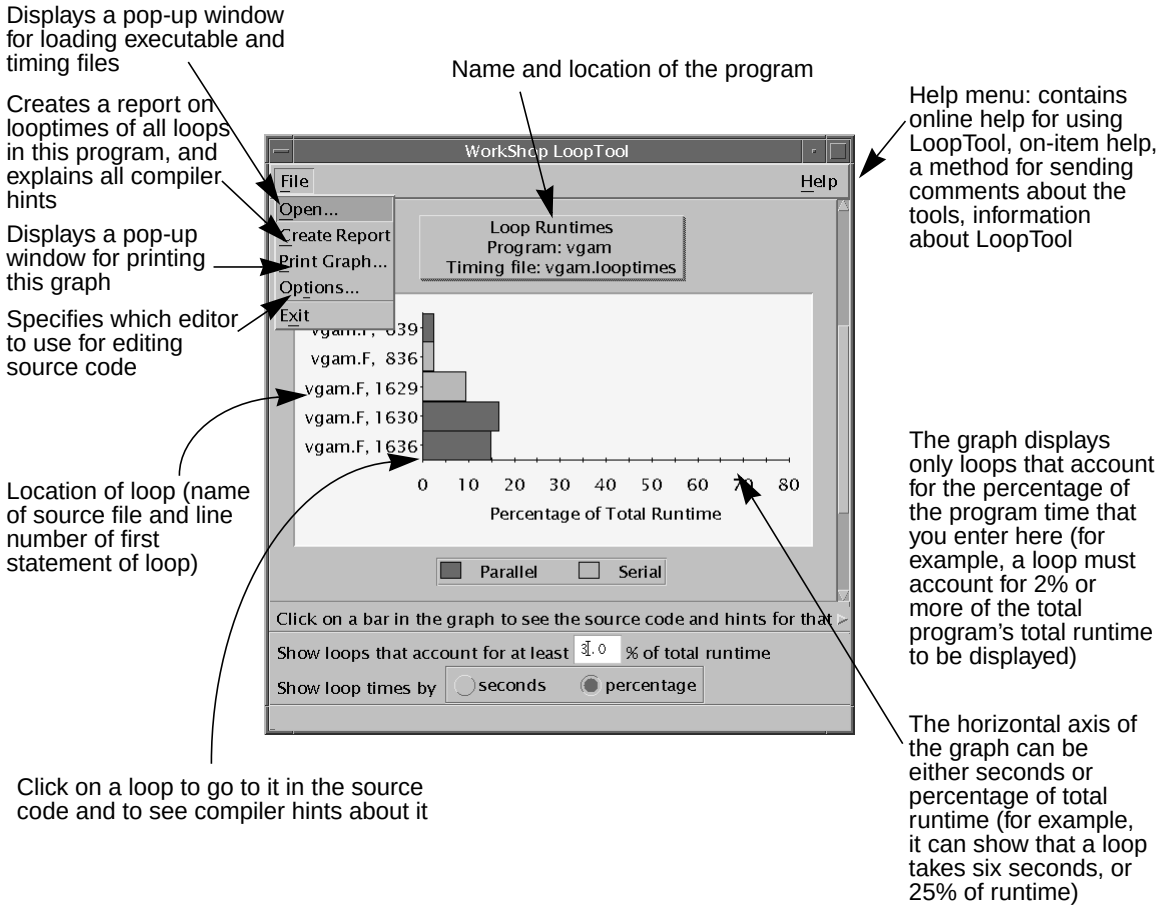

FIGURE 7-1 The LoopTool Main Window

Opening Files

To open executable and timing files, choose File ➤ Open in the main window.

There are two ways to specify the files you want to open:

1. Type in the name of the files to open.

2. Bring up a file chooser.

Once you enter the executable's path, you do not need to type in the timing file, unless it is in a different directory or has a non-default name (or both).

For more information about opening files, see the Analyzing the Loops in Your Program section of the Sun WorkShop Online Help.

Creating a Report on All Loops

To open a window with detailed information on all the loops in your program:

- **Choose File ➤ Create Report in the main window (see** FIGURE 7-2**).**

The generated report is identical to that produced by LoopReport.

The Help button in the report window links to the Sun WorkShop online help section containing compiler hints.

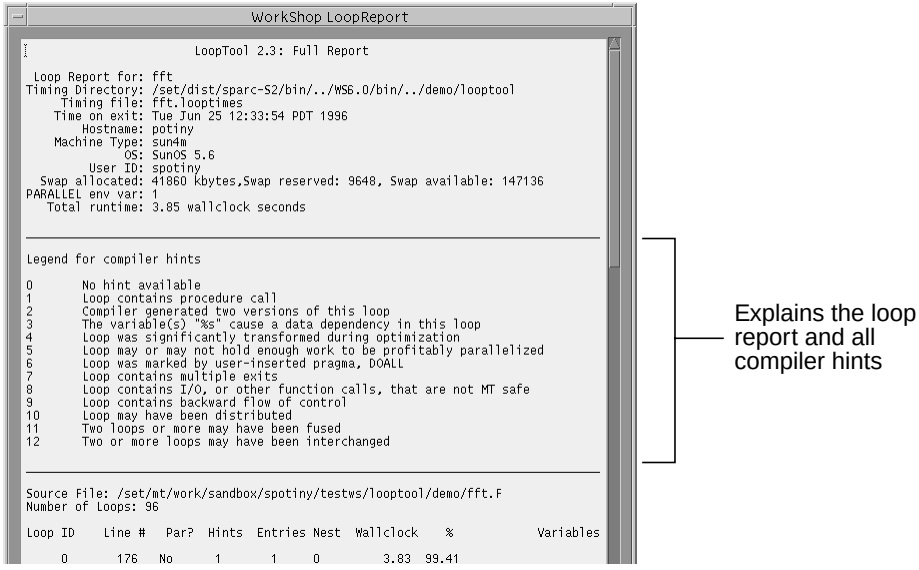

FIGURE 7-2 The LoopReport Window

Printing the LoopTool Graph

To print the LoopTool graph, choose File ➤ Print Graph in the main window and type the name of your chosen printer. To save the graph to a file, type a filename instead of a printer name.

For more information about printing see the Sun WorkShop online help.

Choosing an Editor

Choose File ➤ Options in the main window to open the Text Editor Options dialog box, where you can choose an editor for editing source code. The available editors are `vi`, `gnuemacs`, and `xemacs`.

Note – `vi` and `xemacs` are installed with LoopTool into your install directory (usually `/opt/SUNWspro/bin`) if they are not already on your system. You must provide `gnuemacs` yourself. In all cases, the editor you use must be in a directory in your search path in order for LoopTool to find it. For example, your `PATH` environment variable should include `/usr/local` if that is where `gnuemacs` is located on your system.

For more information about choosing an editor, see "Changing Text Editors" in the Sun WorkShop online help.

Editing Source Code and Getting Hints

Clicking a loop in the main window (see FIGURE 7-1 on page 118) does two things:

- It brings up a window in which you can edit your source code (see FIGURE 7-3 on page 121). The available editors are `vi`, `xemacs`, and `gnuemacs`.

  For information on `vi`, see the `vi`(1) manual page. `xemacs` and `gnuemacs` have online help (click the Help button).

  The Sun WorkShop `vi` editor has a special Version menu that allows you to make use of the Source Code Control System (SCCS) utility for sharing files. See the online help, as well as the `sccs`(1) manual page, for more information.

- It brings up a separate window that displays one or more hints about the loop you have selected. The Help button in this window displays the Sun WorkShop online help compiler hints section. See also "Compiler Hints" on page 126, which explains the hints in detail.

FIGURE 7-3 shows an `xemacs` editor window with a loop selected, and a hint window with an explanation of a compiler hint.

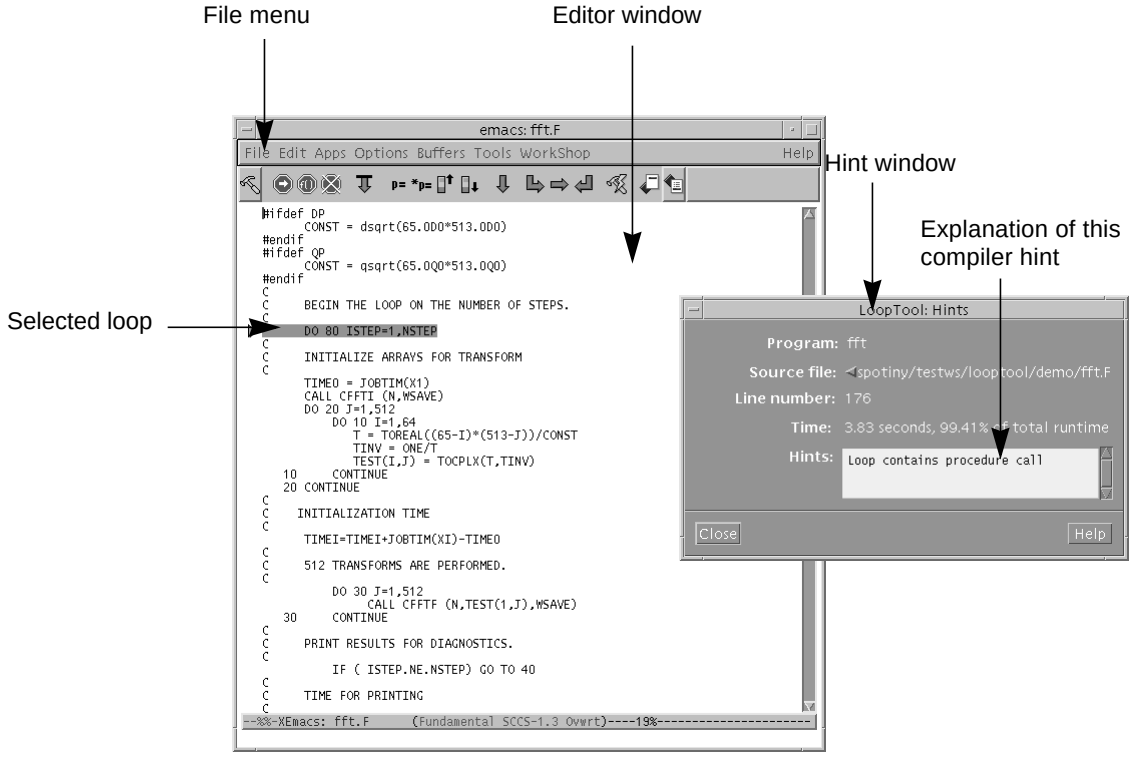

FIGURE 7-3 The Text Editor and Hints Windows

Caution – If you edit your source code, line numbers shown by LoopTool may become inconsistent with the source. You must save and recompile the edited source and then run LoopTool with the new executable, producing new loop information, for the line numbers to remain consistent.

Starting LoopReport

When you start LoopReport, you usually enter the name of your program. Type `loopreport` and the name of the program (an executable) you want to examine.

```
% loopreport program
```

You can also start LoopReport with no file specified. However, if you invoke LoopReport without giving it the name of a program, it looks for a file named `a.out` in the current working directory.

```
% loopreport > a.out.loopreport
```

You can also direct the output into a file, or pipe it into another command:

```
% loopreport program > program.loopreport
% loopreport program | more
```

Timing File

LoopReport also reads the *timing file* associated with your program. The timing file is created when you use the `-zlp` option, and it contains information about loops. Typically, this file has a name of the format *program.*`looptimes` and is found in the same directory as your program.

However, there are four ways to specify the location of a timing file. LoopReport chooses a timing file according to the following rules.

- If a timing file is named on the command line, LoopReport uses that file.

  ```
  % loopreport program newtimes > program.loopreport
  ```

- If the command-line option `-p` is used, LoopReport looks in the directory named by `-p` for a timing file.

  ```
  % loopreport program -p /home/timingfiles > program.loopreport
  ```

- If the environment variable LVPATH is set, LoopReport looks in that directory for a timing file.

```
% setenv LVPATH /home/timingfiles
% loopreport program > program.loopreport
```

- LoopReport writes the table of loop statistics to standard output, stdout. You can also redirect the output to a file, or pipe it into another command:

```
% loopreport program > program.loopreport
% loopreport program | more
```

FIGURE 7-4 shows a sample loop report.

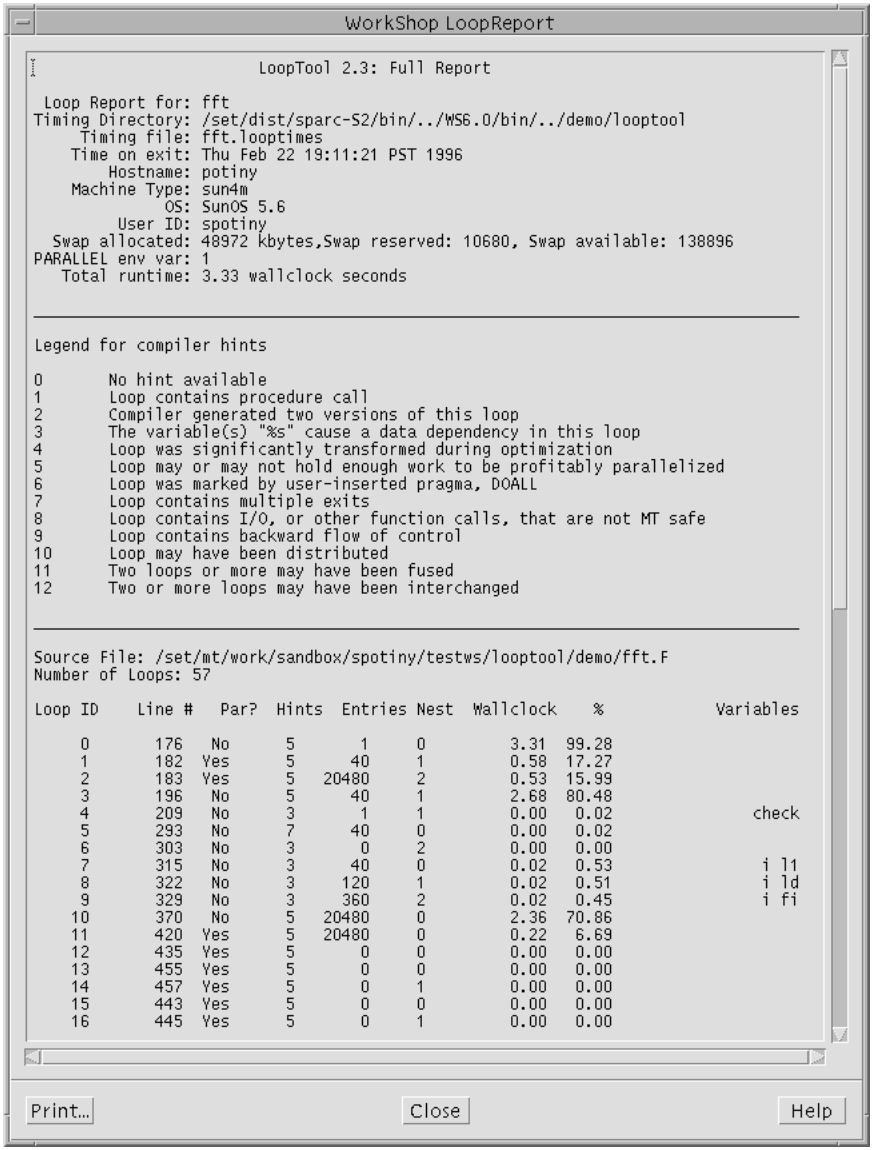

FIGURE 7-4 Sample Loop Report

Fields in the Loop Report

The following descriptions apply equally to LoopTool's "Create Report" output and LoopReport's output.

The loop report contains the following information:

- `LoopID`

  An arbitrary number, assigned by the compiler during compile time. This is just an internal loopID, useful for talking about loops, but not really related in any way to your program.

- `Line #`

  The line number of the first statement of the loop in the source file.

- `Par?`

  `Par` is an abbreviation for "Parallelized by the compiler?" `Y` means that this loop was marked for parallelization; `N` means that the loop was not.

- `Hints`

  Number corresponding to hint text in the "Legend for compiler hints" list.

- `Entries`

  Number of times this loop was entered from above. This is distinct from the number of loop iterations, which is the total number of times a loop executes. For example, these are two loops in Fortran.

```
do 10 i=1,17
    do 10 j=1,50
    ...some code...
    10 continue
```

  The first loop is entered once, and it iterates 17 times. The second loop is entered 17 times, and it iterates 17*50 = 850 times.

- `Nest`

  Nesting level of the loop. If a loop is a top-level loop, its nesting level is 0. If the loop is the child of another loop, its nesting level is 1.

  For example, in this C code, the `i` loop has a nesting level of 0, the `j` loop has a nesting level of 1, and the `k` loop has a nesting level of 2.

```
for (i=0; i<17; i++)
    for (j=0; j<42; j++)
        for (k=0; k<1000; k++)
            do something;
```

- `Wallclock`

  The total amount of elapsed wall-clock time spent executing this loop for the whole program. The elapsed time for an outer loop includes the elapsed time for an inner loop. For example:

```
for (i=1; i<10; i++)
    for (j=1; j<10; j++)
        do something;
```

  The time assigned to the outer loop (the `i` loop) might be 10 seconds, and the time assigned to the inner loop (the `j` loop) might be 9.9 seconds.

- `Percentage`

  The percentage of total program runtime measured as wall-clock time spent executing this loop. As with wall-clock time, outer loops are credited with time spent in loops they contain.

- `Variables`

  The names of the variables that cause a data dependency in this loop. This field only appears when the compiler hint indicates that this loop suffers from a data dependency. A data dependency occurs when parallelization of a loop can not be done safely because the values computed in one iteration of a loop are used in another. The following illustrates a data dependency:

```
do i = 1, N
    a(i) = b(i) + c(i)
    b(i) = 2 * a(i + 1)
end do
```

  If the example loop above is run in parallel, iteration 1 which recomputes `b(1)` based on the value of `a(2)`, may run after iteration 2 which has recomputed `a(2)`. The value of `b(1)` is determined by the new value of `a(2)` rather than the original value as would happen if the loop is not parallelized.

Compiler Hints

LoopTool and LoopReport present hints about the optimizations applied to a particular loop, and about why a loop might not have been parallelized. The hints are heuristics gathered by the compiler during optimization. They should be understood in that context; they are *not* absolute facts about the code generated for a

given loop. However, the hints are often very useful indications of how you can transform your code so that the compiler can perform more aggressive optimizations, including parallelizing loops.

For some useful explanations and tips, read the sections in the *Sun WorkShop Fortran User's Guide* that address parallelization.

0. No hint available

None of the other hints applied to this loop. This hint does not mean that none of the other hints might apply; it means that the compiler did not infer any of those hints.

1. Loop contains procedure call

The loop could not be parallelized since it contains a procedure call that is not MT safe. If such a loop were parallelized, multiple copies of the loop might instantiate the function call simultaneously, trample on each other's use of any variables local to that function, or trample on return values, and generally invalidate the function's purpose. If you are certain that the procedure calls in this loop are MT safe, you can direct the compiler to parallelize this loop no matter what by inserting the `DOALL` pragma before the body of the loop. For example, if `foo` is an MT-safe function call, then you can force it to be parallelized by inserting `c$par DOALL`:

```
c$par DOALL
    do 19 i = 1, n*n
        do 29 k = i, n*n
            a(i) = a(i) + x*b(k)*c(i-k)
            call foo()
29      continue
19  continue
```

The computer interprets the `DOALL` pragmas only when you compile with `-parallel` or `-explicitpar`; if you compile with `-autopar`, then the compiler ignores the `DOALL` pragmas.

2. Compiler generated two versions of this loop

The compiler could not tell at compile time if the loop contained enough work to be profitable to parallelize. The compiler generated two versions of the loop, a serial version and a parallel version, and a runtime check that will choose at runtime which version to execute. The runtime check determines the amount of work that the loop has to do by checking the loop iteration values.

3. The variable(s) "*list*" cause a data dependency in this loop

A variable inside the loop is affected by the value of a variable in a previous iteration of the loop. For example:

```
do 99 i=1,n
    do 99 j = 1,m
    a[i, j+1] = a[i,j] + a[i,j-1]
99 continue
```

This is a contrived example, since for such a simple loop the optimizer would simply swap the inner and outer loops, so that the inner loop could be parallelized. But this example demonstrates the concept of data dependency, which is often referred to as "loop-carried data dependency."

The compiler can often tell you the names of the variables that cause the loop-carried data dependency. If you rearrange your program to remove (or minimize) such dependencies, then the compiler can perform more aggressive optimizations.

4. Loop was significantly transformed during optimization

The compiler performed some optimizations on this loop that might make it almost impossible to associate the generated code with the source code. For this reason, line numbers may be incorrect. Examples of optimizations that can radically alter a loop are loop distribution, loop fusion, and loop interchange (see Hint 10, Hint 11, and Hint 12).

5. Loop may or may not hold enough work to be profitably parallelized

The compiler was not able to determine at compile time whether this loop held enough work to warrant parallelizing. Often loops that are labeled with this hint may also be labeled "parallelized," meaning that the compiler generated two versions of the loop (see Hint 2), and that it will be decided at runtime whether the parallel version or the serial version should be used.

Since all the compiler hints, including the flag that indicates whether or not a loop is parallelized, are generated at compile time, there's no way to be certain that a loop labeled "parallelized" actually executes in parallel.

6. Loop was marked by user-inserted pragma, `DOALL`

This loop was parallelized because the compiler was instructed to do so by the `DOALL` pragma. This hint is a useful reminder to help you easily identify those loops that you explicitly wanted to parallelize.

The `DOALL` pragmas are interpreted by the compiler only when you compile with `-parallel` or `-explicitpar`; if you compile with `-autopar`, then the compiler will ignore the `DOALL` pragmas.

7. Loop contains multiple exits

The loop contains a `GOTO` or some other branch out of the loop other than the natural loop end point. For this reason, it is not safe to parallelize the loop, since the compiler has no way of predicting the loop's runtime behavior.

8. Loop contains I/O, or other function calls, that are not MT safe

This hint is similar to Hint 1. The difference is that this hint often focuses on I/O that is not multithread-safe, whereas Hint 1 can refer to any sort of multithread-unsafe function call.

9. Loop contains backward flow of control

The loop contains a `GOTO` or other control flow up and out of the body of the loop. That is, some statement inside the loop appears to the compiler to jump back to some previously executed portion of code. As with the case of a loop that contains multiple exits, this loop is not safe to parallelize.

If you can reduce or minimize backward flows of control, the compiler will be able to perform more aggressive optimizations.

10. Loop may have been distributed

The contents of the loop may have been distributed over several iterations of the loop. That is, the compiler may have been able to rewrite the body of the loop so that it could be parallelized. However, since this rewriting takes place in the language of the internal representation of the optimizer, it's very difficult to associate the original source code with the rewritten version. For this reason, hints about a distributed loop may refer to line numbers that don't correspond to line numbers in your source code.

11. Two or more loops may have been fused

Two consecutive loops were combined into one, so the resulting larger loop contains enough work to be profitably parallelized. Again, in this case, source line numbers for the loop may be misleading.

12. Two or more loops may have been interchanged

The loop indices of an inner and an outer loop have been swapped, to move data dependencies as far away from the inner loop as possible, and to enable this nested loop to be parallelized. In the case of deeply nested loops, the interchange may have occurred with more than two loops.

How Optimization Affects Loops

As you might infer from the descriptions of the compiler hints, associating optimized code with source code can be tricky. Clearly, you would prefer to see information from the compiler presented to you in a way that relates as directly as possible to your source code. Unfortunately, the compiler optimizer "reads" your program in terms of its internal language, and although it tries to relate that to your source code, it is not always successful.

Some particular optimizations that can cause confusion are described in the following sections.

Inlining

Inlining is an optimization applied only at optimization level `-O4` and only for functions contained within one file. That is, if one file contains 17 Fortran functions, 16 of those can be inlined into the first function, and you compile at `-O4`, then the source code for those 16 functions may be copied into the body of the first function. Then, when further optimizations are applied, it becomes difficult to determine which loop on which source line number was subjected to which optimization.

If the compiler hints seem particularly opaque, consider compiling with `-O3 -parallel -Zlp`, so that you can see what the compiler says about your loops before it tries to inline any of your functions.

In particular, "phantom" loops—that is, loops that the compiler claims exist, but you know do not exist in your source code—could well be a symptom of inlining.

Loop Transformations: Unrolling, Jamming, Splitting, and Transposing

The compiler performs many loop optimizations that radically change the body of the loop. These include optimizations, unrolling, jamming, splitting, and transposing.

LoopTool and LoopReport attempt to provide hints that make as much sense as possible, but given the nature of the problem of associating optimized code with source code, the hints may be misleading.

Parallel Loops Nested Inside Serial Loops

If a parallel loop is nested inside a serial loop, the runtime information reported by LoopTool and LoopReport may be misleading because each loop is stipulated to use the wall-clock time of each of its loop iterations. If an inner loop is parallelized, it is assigned the wall-clock time of each iteration, although some of those iterations are running in parallel.

However, the outer loop is assigned only the runtime of its child, the parallel loop, which will be the runtime of the longest parallel instantiation of the inner loop. This double timing leads to the anomaly of the outer loop apparently consuming less time than the inner loop.

APPENDIX **A**

Traditional Profiling Tools

The tools discussed in this appendix are standard utilities for timing programs and obtaining performance data to analyze. The profiling tools `prof` and `gprof` are provided with Solaris versions 2.6, 7, and 8 of the Solaris Operating Environment *SPARC Platform Edition* and Solaris *Intel Platform Edition*. `tcov` is a code coverage tool.

Note – If you want to track execution counts (how many times a function is called, how often a line of source code executes), use the traditional profiling tools. If you want a detailed analysis of where your program is spending time, you can get more accurate information using the Sampling Collector and Analyzer. See Chapter 3 and Chapter 4 for information on using these applications.

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 chapter covers the following topics:

- Basic Concepts
- Using `prof` to Generate a Program Profile
- Using `gprof` to Generate a Call Graph Profile
- Using `tcov` for Statement-Level Analysis
- Using `tcov` Enhanced for Statement-Level Analysis
- Creating Profiled Shared Libraries

Basic Concepts

`prof`, `gprof`, and `tcov` extend the Sun WorkShop development environment by enabling you to collect and use performance data.

- `prof` generates a program profile in a flat file.
- `gprof` generates a call graph profile.

- `tcov` generates statement-level information in a copy of the source file, annotated to show which lines are used and how often.

TABLE A-1 describes the information 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, along with an exact count of the number of times each function is entered. This tool is included with the Solaris operating environment.
`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. This tool is included with the Solaris operating environment.
`tcov`	Generates exact counts of the number of times each statement in a program is executed. There are two versions of `tcov`: the original `tcov` and an enhanced `tcov`. They differ in the runtime support that generates the raw data used in the output.

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.

Using `prof` involves three basic steps:

1. Compiling the program for `prof`
2. Running the program to produce a profile data file
3. Using `prof` to generate a report that summarizes the data

To compile a program for profiling with `prof`, use the `-p` option to the compiler. For example, to compile a C source file named `index.assist.c` for profiling, you use this compiler command:

```
% cc -p -o index.assist index.assist.c
```

The compiler produces a program called `index.assist`.

Now you run the `index.assist` program. Each time you run the program, profiling data is sent to a profile file called `mon.out`. Every time you run the program a new `mon.out` file is created that overwrites the old version.

Finally, you would use the `prof` command to generate a report:

```
% index.assist
% ls mon.out
mon.out
% prof index.assist
```

Output Example

The `prof` output resembles this example.

%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)
```

Sample `prof` Output

The sample display shows that most of the program execution time is spent in the `compare_strings()` routine; after that, most of the CPU time is spent in the `_strlen()` library routine. To make improvements to this program, concentrate on the `compare_strings()` function.

The results of the profiling sample run are listed under these column headings:

- `%Time`—The percentage of the total CPU time consumed by this routine of the program.
- `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 routine is called.
- `msecs/call`—The average number of milliseconds this routine consumes each time it is called.
- `Name`—The name of the routine.

What results can be derived from the profile data? The `compare_strings()` function consumes nearly 20% of the total CPU time. To improve the runtime of `index.assist`, you can either improve the algorithm that `compare_strings()` uses, or cut down the number of calls to `compare_strings()`.

It is not obvious from the flat call graph 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 137. 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` allocates the time within a function to the callers in proportion to the number of times that each arc is traversed. Because all calls are not equivalent in performance, this behavior might lead to incorrect assumptions. See "The `gprof` Fallacy" on page 19 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.

Using `gprof` involves three basic steps:

1. Compiling the program for `gprof`
2. Running the program to produce a profile data file
3. Using `gprof` to generate a report that summarizes the data

To compile the program for call graph profiling, use the `-xpg` option for the C compiler or the `-pg` option for the Fortran compiler. For example:

```
% cc -xpg -o index.assist index.assist.c
```

Now run the `index.assist` program. Each time you run a program compiled for `gprof`, call-graph profile data is sent to a file called `gmon.out`. This file is overwritten each time you run the program.

Finally, use the `gprof` command to generate a report of the results of the profile. The output from `gprof` can be large. You might find the report easier to read if you redirect it to a file. To redirect the output from `gprof` to a file named `g.output`, use this sequence of commands:

```
% index.assist
% ls gmon.out
gmon.out
% gprof index.assist > g.output
```

The output from `gprof` consists of two major items:

- The full call graph profile, which shows fragments of output from a profiling run.
- The "flat" profile, which is similar to the summary the `prof` command supplies.

The output from `gprof` contains an explanation of what the various parts of the summary mean. `gprof` also identifies the granularity of the sampling:

```
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 following example 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]
-----------------------------------------------
```

If you assume that there are 761 lines of data in the input file to the `index.assist` program, you can conclude:

- `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 could 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

`tcov` gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also summarizes information about basic blocks. `tcov` does not product any time-based data.

Using `tcov` involves three basic steps:

1. Compiling the program to produce a `tcov` experiment
2. Running the experiment
3. Using `tcov` to create summaries of execution counts for each statement in the program

Compiling for `tcov`

To compile a program for code coverage, use the `-xa` option to the C compiler. For example, you compile a program named `index.assist.c` for use with `tcov` with this command:

```
% cc -xa -o index.assist index.assist.c
```

Use the `-a` compiler option instead of `-xa` with the C++ or `Fortran` compilers.

The C compiler generates an `index.assist.d` file, containing database entries for the basic blocks present in `index.assist.c`. When the program `index.assist` is run to completion, the compiler updates the `index.assist.d` file.

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. Applications compiled with `-xa` (C), `-a` (other compilers), and `+d` (C++) run more slowly than they normally would. The `+d` option inhibits expansion of C++ inline functions, and updating the `.d` file for each execution takes considerable time.

The `index.assist.d` file is created in the directory specified by the environment variable `TCOVDIR`. If `TCOVDIR` is not set, `index.assist.d` is created in the current directory.

Having compiled `index.assist.c`, you can run `index.assist`:

```
% index.assist
% ls *.d
index.assist.d
```

Now you can run `tcov` to produce a file containing the summaries of execution counts for each statement in the program. `tcov` uses the `index.assist.d` file to generate an `index.assist.tcov` file containing an annotated list of your code. The output shows the number of times each source statement is executed. At the end of the file, there is a short summary.

```
% tcov index.assist.c
% ls *.tcov
index.assist.tcov
```

This small fragment of the C code from one of the modules of `index.assist` shows the `insert_index_entry()` function, which is called so recursively.

```
        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 numbers to the left of the C code show how many times each statement was executed. The `insert_index_entry` function is called 11,152 times.

`tcov` places a summary like the following at the end of the annotated program listing for `index.assist.tcov`:

```
  Top 10 Blocks

        Line       Count

        240        21563

        241        21563

        245        21563

        251        21563

        250        21400

        244        21299

        255        20612

        257        16805

        123        12021

        124        11962


77      Basic blocks in this file

55      Basic blocks executed

71.43   Percent of the file executed


        439144    Total basic block executions

        5703.17   Average executions per basic block
```

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.

Creating `tcov` Profiled Shared Libraries

It is possible to create a `tcov` profiled shareable library and use it in place of one where binaries have already been linked. Include the `-xa` (C) or `-a` (other compilers) option when creating the shareable libraries. For example:

```
% cc -G -xa -o foo.so.1 foo.o
```

This command includes a copy of the `tcov` profiling subroutines in the shareable libraries, so that clients of the library do not need to relink. If a client of the library is also linked for profiling, then the version of the `tcov` subroutines used by the client is used to profile the shareable library.

Locking Files

`tcov` uses a simple file-locking mechanism for updating the block coverage database in the `.d` files. It employs a single file, `tcov.lock`, for this purpose. Consequently, only one executable compiled with `-xa` (C) or `-a` (other compilers) should be running on the system at a time. If the execution of the program compiled with the `-xa` (or `-a`) option is manually terminated, then the `tcov.lock` file must be deleted manually.

Files compiled with the `-xa` or `-a` option call the profiling tool subroutines automatically when a program is linked for `tcov` profiling. At program exit, these subroutines combine the information collected at runtime for file `xyz.f` (for example) with the existing profiling information stored in file `xyz.d`. To ensure this information is not corrupted by several people simultaneously running a profiled binary, a `xyz.d.lock` lock file is created for `xyz.d` for the duration of the update. If there are any errors in opening or reading `xyz.d` or its lock file, or if there are inconsistencies between the runtime information and the stored information, the information stored in `xyz.d` is not changed.

An edit and recompile of `xyz.d` may change the number of counters in `xyz.d`. 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 similar to the following is displayed after a delay of several seconds:

```
tcov_exit: Failed to create lock file '/tmp_mnt/net/rbbb/export/
home/src/newpattern/foo.d.lock' for coverage data file '/tmp_mnt/
net/rbbb/export/home/src/newpattern/foo.d' after 5 tries. Is
somebody else running this binary?
```

The stored information is not updated. This locking is safe across a network. Since locking is performed on a file-by-file basis, other files may be correctly updated.

The profiling subroutines attempt to deal with automounted file systems that have become inaccessible. They still fail if the file system containing a coverage data file is mounted with different names on different machines, or if the user running the profiled binary does not have permission to write to either the coverage data file or the directory containing it. Be sure all the directories are uniformly named and writable by anyone expected to run the binary.

Errors Reported by `tcov` Runtime Routines

The following error messages may be reported by the `tcov` runtime routines:

- The user running the binary lacks permission to read or write to the coverage data file. The problem also occurs if the coverage data file has been deleted.

  `tcov_exit: Could not open coverage data file '`*coverage_data_file_name*`' because '`*system_error_message_string*`'.`

- The user running the binary lacks permission to write to the directory containing the coverage data file. The problem also occurs if the directory containing the coverage data file is not mounted on the machine where the binary is being run.

  `tcov_exit: Could not write coverage data file '`*coverage_data_file_name*`' because '`*system_error_message_string*`'.`

- Too many users are trying to update a coverage data file at the same time. The problem also occurs if a machine has crashed while a coverage data file is being updated, leaving behind a lock file. In the event of a crash, the longer of the two files should be used as the post-crash coverage data file. Manually remove the lock file.

```
tcov_exit: Failed to create lock file 'lock_file_name' for coverage
data file 'coverage_data_file_name' after 5 tries. Is someone else
running this executable?
```

- No memory is available, and the standard I/O package will not work. You cannot update the coverage data file at this point.

```
tcov_exit: Stdio failure, probably no memory left.
```

- The lock file name is longer by six characters than the coverage data file name. Therefore, the derived lock file name may not be legal.

```
tcov_exit: Coverage data file path name too long (length
characters) 'coverage_data_file_name'.
```

- A library or binary that has `tcov` profiling enabled is simultaneously being run, edited, and recompiled. The old binary expects a coverage data file of a certain size, but the editing often changes that size. If the compiler creates a new coverage data file at the same time that the old binary is trying to update the old coverage data file, the binary may see an apparently empty or corrupt coverage file.

```
tcov_exit: Coverage data file 'coverage_data_file_name' is too short.
Is it out of date?
```

Using `tcov` Enhanced for Statement-Level Analysis

Like the original `tcov`, `tcov` Enhanced gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also gives a summary of information about basic blocks. `tcov` Enhanced works with both C and C++ source files.

Advantages of `tcov` Enhanced

`tcov` Enhanced overcomes some of the shortcomings of the original `tcov`:

- 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.

Compiling for `tcov` Enhanced

To use `tcov` Enhanced, follow the same basic steps as the original `tcov`:

1. Compile a program for a `tcov` Enhanced experiment.
2. Run the experiment.
3. Analyze the results using `tcov`.

For the original `tcov`, you compile with the `-xa` option. To compile a program for code coverage with `tcov` Enhanced, you use the `-xprofile=tcov` option for all compilers. Using a program named `index.assist` as an example, you would compile for use with `tcov` Enhanced with this command:

```
% cc -xprofile=tcov -o index.assist index.assist.c
```

`tcov` Enhanced, unlike `tcov`, does not produce a `.d` file. The coverage data file is not created until the program is run. Then one coverage data file is produced as opposed to one file for each module compiled for coverage analysis.

After you compile `index.assist.c`, you run `index.assist` to create the profile file:

```
% index.assist
% ls -dF *.profile
index.assist.profile/
% ls *.profile
tcovd
```

By default, the name of the directory where the `tcovd` file is stored is derived from the name of the executable. Furthermore, that directory is created in the directory the executable was run in (the original `tcov` created the `.d` files in the directory where the modules were compiled).

The directory where the `tcovd` file is stored is also known as the *profile bucket*. The profile bucket can be overridden by using the `SUN_PROFDATA` environment variable. 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).

You can also override the directory where the profile bucket is created. To specify a location different from the run directory, specify the path using the `SUN_PROFDATA_DIR` environment variable. Absolute or relative pathnames can be specified in this variable. Relative pathnames are relative to the program's current working directory at program completion.

`TCOVDIR` is supported as a synonym for `SUN_PROFDATA_DIR` for 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. `SUN_PROFDATA_DIR` takes precedence over `TCOVDIR`. The variables are used at runtime by a program compiled with `-xprofile=tcov`, and are used by the `tcov` command.

Note – This scheme is also used by the profile feedback mechanism.

Now that some coverage data has been produced, you could generate a report that relates the raw data back to the source files:

```
% tcov -x index.profile index.assist.c
% ls *.tcov
index.assist.c.tcov
```

The output of this report is identical to the one from the previous example (for the original `tcov`).

Creating Profiled Shared Libraries

Creating shared libraries for use with `tcov` Enhanced is accomplished by using the analogous compiler options:

```
% cc -G -xprofile=tcov -o foo.so.1 doo.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 your compiler.

The default profile bucket the program creates is named after the executable with a `.profile` extension and is created in the directory where the executable is run. Therefore, if you are in `/home/userdir`, and run a program called `/usr/bin/xyz`, the default behavior is to create a profile bucket called `xyz.profile` in `/home/userdir`.

You 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.

- `SUN_PROFDATA_DIR`

  Can be used to specify the name of the directory containing the profile bucket. It is used at runtime and in 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 by a program compiled with `-xprofile=tcov` and it is used by the `tcov` command.

Overriding the Default Definitions

To override the default, use the environment variables to change the profile bucket:

1. **Change the name of the profile bucket by using the `SUN_PROFDATA` environment variable.**
2. **Change the directory where the profile-bucket is placed by using the `SUN_PROFDATA_DIR` environment variable.**

The environment variables override the default location and name of the profile bucket. Both can be overridden independently. For example, if you only choose to set `SUN_PROFDATA_DIR`, the profile bucket will go into the directory where you set `SUN_PROFDATA_DIR`. The default name (which is the executable name followed by a `.profile` extension) will still be the name used for the profile bucket.

Absolute and Relative Pathnames

There are two forms of directories you can specify by using SUN_PROFDATA_DIR on the Profile Feedback compile line: absolute pathnames (which start with a '/'), and relative pathnames. If you use an absolute pathname, the profile bucket is dropped into that directory. If you specify a relative pathname, then it is relative to the current working directory where the executable is being run.

For example, if you are in `/home/userdir` and run a program called `/usr/bin/xyz` with SUN_PROFDATA_DIR set to `..` , then the profile bucket is called `/home/userdir/../xyz.profile`. The value specified in the environment variable was relative, and therefore, it was relative to `/home/userdir`. Also, the default profile bucket name is used, which is named after the executable.

TCOVDIR and SUN_PROFDATA_DIR

The previous version of `tcov` (enabled by compiling with the `-xa` or `-a` flag) used an environment variable called TCOVDIR. TCOVDIR specified the directory where the `tcov` counter files go to instead of next to the source files. To retain compatibility with this environment variable, the new SUN_PROFDATA_DIR environment variable behaves like the TCOVDIR environment variable. If both variables are set, a warning is output and SUN_PROFDATA_DIR takes precedence over TCOVDIR.

For `-xprofile=tcov`

By default the profile bucket is called `<argv[0]>.profile` in the current directory.

If you set SUN_PROFDATA, the profile bucket is called $SUN_PROFDATA, wherever it is located.

If you set SUN_PROFDATA_DIR, the profile bucket is placed in the specified directory.

SUN_PROFDATA and SUN_PROFDATA_DIR are independent. If both are specified, the profile bucket name is generated by using SUN_PROFDATA_DIR to find the profile bucket and, SUN_PROFDATA is used to name the profile bucket in that directory.

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.

For the `tcov` Program

The `-xprofile=bucket` option specifies the name of the profile bucket to use for the `tcov` profile. `SUN_PROFDATA_DIR` or `TCOVDIR` are prepended to this argument, if they are set.

Index

B

C

D

E

F

M

N

O

T

U

V

W

X

Z