Oracle® Solaris Studio 12.3: Performance Analyzer

Part No: E21994
December 2011

120118@25097

Contents

Preface

This *Performance Analyzer* manual describes the performance analysis tools in the Oracle Solaris Studio software. The Collector and Performance Analyzer are a pair of tools that collect a wide range of performance data and relate the data to program structure at the function, source line, and instruction level. The performance data collected include statistical clock profiling, hardware counter profiling, and tracing of various calls.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64" points out specific 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the Oracle Solaris Studio 12.3 Release Notes.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.

- Man pages for command-line tools describe a tool's command options.

Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Description	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

CHAPTER 1

Overview of the Performance Analyzer

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

This chapter covers the following topics:

- "The Tools of Performance Analysis" on page 17
- "The Performance Analyzer Window" on page 20

The Tools of Performance Analysis

This manual describes the Collector and Performance Analyzer, a pair of tools that you use to collect and analyze performance data for your application. The manual also describes the `er_print` utility, a command-line tool for displaying and analyzing collected performance data in text form. The Analyzer and `er_print` utility show mostly the same data, but use different user interfaces.

An additional Oracle Solaris Studio tool called Spot can be used to produce a performance report about your application. This tool is complementary to the Performance Analyzer. See the `spot`(1) man page for more information.

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

The Collector and Performance Analyzer tools help to answer the following kinds of questions:

- How much of the available resources does the program consume?
- Which functions or load objects are consuming the most resources?

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

The Collector Tool

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

The Performance Analyzer Tool

The Performance Analyzer tool displays the data recorded by the Collector, so that you can examine the information. The Performance Analyzer processes the data and displays various metrics of performance at the level of the program, the functions, the source lines, and the instructions. These metrics are classed into five groups:

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

The Performance Analyzer's Timeline displays the raw data in a graphical format as a function of time.

The Performance Analyzer can also display metrics of performance for structures in the dataspace of the target program, and for structural components of the memory subsystem. This data is an extension of the hardware counter metrics.

In addition, the Performance Analyzer can display data for the Thread Analyzer, a specialized view of the Performance Analyzer that is designed for examining thread analysis experiments. A separate command, `tha`, is used to start the Performance Analyzer with this specialized view, and the tool when started in this way is known as the Thread Analyzer.

The Thread Analyzer can show data races and deadlocks in experiments that you can generate specifically for examining these types of data. A separate manual, the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* describes how to use the Thread Analyzer.

See Chapter 4, "The Performance Analyzer Tool," and the online help in the Performance Analyzer for detailed information about the Performance Analyzer.

Chapter 5, "The `er_print` Command Line Performance Analysis Tool," describes how to use the `er_print` command line interface to analyze the data collected by the Collector.

Chapter 6, "Understanding the Performance Analyzer and Its Data," discusses topics related to understanding the Performance Analyzer and its data, including: how data collection works, interpreting performance metrics, call stacks and program execution.

Chapter 7, "Understanding Annotated Source and Disassembly Data," provides an understanding of the annotated source and disassembly, providing explanations about the different types of index lines and compiler commentary that the Performance Analyzer displays. Annotated source code listings and disassembly code listings that include compiler commentary but do not include performance data can be viewed with the `er_src` utility, which is also described in this chapter.

Chapter 8, "Manipulating Experiments," describes how to copy, move, and delete experiments; add labels to experiments; and archive and export experiments.

Chapter 9, "Kernel Profiling," describes how you can use the Oracle Solaris Studio performance tools to profile the kernel while the Oracle Solaris operating system is running a load.

Note – You can download demonstration code for the Performance Analyzer in the sample applications zip file from the Oracle Solaris Studio 12.3 Sample Applications downloads page at `http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/`.

After downloading, you can extract the zip file in a directory of your choice. The sample applications are located in the `PerformanceAnalyzer` subdirectory of the `SolarisStudioSampleApplications` directory. See the `README` file in each sample directory for information about how to use the sample code with the Analyzer.

The `er_print` Utility

The `er_print` utility presents in plain text all the displays that are presented by the Performance Analyzer, with the exception of the Timeline display, the MPI Timeline display, and the MPI Chart display. These displays are inherently graphical and cannot be presented as text.

The Performance Analyzer Window

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

The Performance Analyzer window consists of a multi-tabbed display, with a menu bar and a toolbar. The tab that is displayed when the Performance Analyzer is started shows a list of functions for the program with exclusive and inclusive metrics for each function.

Powerful filtering technology can be applied to drill down into performance problems by selecting filters from a context menu in most of the tabs. Data can be filtered by function, thread, CPU, or time, and by any combination of these and other filter elements.

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

Two other tabs display source code that is annotated line-by-line with performance metrics and interleaved with compiler commentary, and disassembly code that is annotated with metrics for each instruction and interleaved with both source code and compiler commentary if they are available.

The performance data is displayed as a function of time in another tab.

MPI tracing data is displayed as processes, messages, and functions in a timeline in one tab, and as charts in another tab.

OpenMP parallel regions are displayed on one tab, OpenMP tasks on another tab.

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

Other tabs show Index Objects, Memory Objects, Data Objects, Data Layout, Lines, and PCs. See the "Analyzer Data Displays" on page 89 for more information about each tab.

For experiments that have recorded Thread Analyzer data, tabs for data Races and Deadlocks are also available. Tabs are shown only if the loaded experiments have data supporting them.

See the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information about Thread Analyzer.

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

CHAPTER 2

Performance Data

The performance tools work by recording data about specific events while a program is running, and converting the data into measurements of program performance called metrics. Metrics can be shown against functions, source lines, and instructions.

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

This chapter covers the following topics.

- "What Data the Collector Collects" on page 21
- "How Metrics Are Assigned to Program Structure" on page 36

See Chapter 3, "Collecting Performance Data," for information on collecting and storing performance data.

What Data the Collector Collects

The Collector collects various kinds of data using several methods:

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

- Function and instruction count data is collected for the executable and for any shared objects that are instrumented and that the executable statically links with. The number of times functions and instructions were executed is recorded.
- Thread analysis data is collected to support the Thread Analyzer.

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

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

- A header identifying the data
- A high-resolution timestamp
- A thread ID
- A lightweight process (LWP) ID
- A processor (CPU) ID, when available from the operating system
- A copy of the call stack. For Java programs, two call stacks are recorded: the machine call stack and the Java call stack.
- For OpenMP programs, an identifier for the current parallel region and the OpenMP state are also collected.

For more information on threads and lightweight processes, see Chapter 6, "Understanding the Performance Analyzer and Its Data."

In addition to the common data, each event-specific data packet contains information specific to the data type.

The data types and how you might use them are described in the following subsections:

- "Clock Data" on page 22
- "Hardware Counter Overflow Profiling Data" on page 26
- "Synchronization Wait Tracing Data" on page 29
- "Heap Tracing (Memory Allocation) Data" on page 30
- "MPI Tracing Data" on page 31
- "Global (Sampling) Data" on page 35

Clock Data

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

Clock-based Profiling Under Oracle Solaris

In clock-based profiling under Oracle Solaris, the state of each thread is stored at regular time intervals. This time interval is called the profiling interval. The information is stored in an integer array: one element of the array is used for each of the ten microaccounting states maintained by the kernel. The data collected is converted by the Performance Analyzer into times spent in each state, with a resolution of the profiling interval. The default profiling interval is approximately 10 milliseconds (10 ms). The Collector provides a high-resolution profiling interval of approximately 1 ms and a low-resolution profiling interval of approximately 100 ms, and, where the operating system permits, allows arbitrary intervals. Running the `collect -h` command with no other arguments prints the range and resolution allowable on the system on which it is run.

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

TABLE 2–1 Solaris Timing Metrics

Metric	Definition
User CPU time	Time spent running in user mode on the CPU.
Wall time	Elapsed time spent in Thread 1. This is usually the “wall clock time”
Total thread time	Sum of all thread times.
System CPU time	Thread time spent running in kernel mode on the CPU or in a trap state.
Wait CPU time	Thread time spent waiting for a CPU.
User lock time	Thread time spent waiting for a lock.
Text page fault time	Thread time spent waiting for a text page.
Data page fault time	Thread time spent waiting for a data page.
Other wait time	Thread time spent waiting for a kernel page, or time spent sleeping or stopped.

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

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

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

- High text page fault time means that the code ordered by the linker is organized in memory so that many calls or branches cause a new page to be loaded.
- High data page fault time indicates that access to the data is causing new pages to be loaded. Reorganizing the data structure or the algorithm in your program can fix this problem.

Clock-based Profiling Under Linux

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

Clock-based Profiling for MPI Programs

Clock-profiling data can be collected on an MPI experiment that is run with Oracle Message Passing Toolkit, formerly known as Sun HPC ClusterTools. The Oracle Message Passing Toolkit must be at least version 8.1.

The Oracle Message Passing Toolkit is made available as part of the Oracle Solaris 11 release. If it is installed on your system, you can find it in `/usr/openmpi`. If it is not already installed on your Oracle Solaris 11 system, you can search for the package with the command `pkg search openmpi` if a package repository is configured for the system. See the manual *Adding and Updating Oracle Solaris 11 Software Packages* in the Oracle Solaris 11 documentation library for more information about installing software in Oracle Solaris 11.

When you collect clock-profiling data on an MPI experiment, two additional metrics can be shown:

- MPI Work, which accumulates when the process is inside the MPI runtime doing work, such as processing requests or messages
- MPI Wait, which accumulates when the process is inside the MPI runtime, but waiting for an event, buffer, or message

On Oracle Solaris, MPI Work accumulates when work is being done either serially or in parallel. MPI Wait accumulates when the MPI runtime is waiting for synchronization, and accumulates whether the wait is using CPU time or sleeping, or when work is being done in parallel, but the thread is not scheduled on a CPU.

On Linux, MPI Work and MPI Wait are accumulated only when the process is active in either user or system mode. Unless you have specified that MPI should do a busy wait, MPI Wait on Linux is not useful.

Note – If your are using Linux with Oracle Message Passing Toolkit 8.2 or 8.2.1, you might need a workaround. The workaround is not needed for version 8.1 or 8.2.1c, or for any version if you are using an Oracle Solaris Studio compiler.

The Oracle Message Passing Toolkit version number is indicated by the installation path such as /opt/SUNWhpc/HPC8.2.1, or you can type mpirun –V to see output as follows where the version is shown in italics:

```
mpirun (Open MPI) 1.3.4r22104-ct8.2.1-b09d-r70
```

If your application is compiled with a GNU or Intel compiler, and you are using Oracle Message Passing Toolkit 8.2 or 8.2.1 for MPI, to obtain MPI state data you must use the -WI and --enable-new-dtags options with the Oracle Message Passing Toolkit link command. These options cause the executable to define RUNPATH in addition to RPATH, allowing the MPI State libraries to be enabled with the LD_LIBRARY_PATH environment variable.

Clock-based Profiling for OpenMP Programs

If clock-based profiling is performed on an OpenMP program, two additional metrics are provided: OpenMP Work and OpenMP Wait.

On Oracle Solaris, OpenMP Work accumulates when work is being done either serially or in parallel. OpenMP Wait accumulates when the OpenMP runtime is waiting for synchronization, and accumulates whether the wait is using CPU time or sleeping, or when work is being done in parallel, but the thread is not scheduled on a CPU.

On the Linux operating system, OpenMP Work and OpenMP Wait are accumulated only when the process is active in either user or system mode. Unless you have specified that OpenMP should do a busy wait, OpenMP Wait on Linux is not useful.

Data for OpenMP programs can be displayed in any of three view modes. In User mode, slave threads are shown as if they were really cloned from the master thread, and have call stacks matching those from the master thread. Frames in the call stack coming from the OpenMP runtime code (libmtsk.so) are suppressed. In Expert user mode, the master and slave threads are shown differently, and the explicit functions generated by the compiler are visible, and the frames from the OpenMP runtime code (libmtsk.so) are suppressed. For Machine mode, the actual native stacks are shown.

Clock-based Profiling for the Oracle Solaris Kernel

The er_kernel utility can collect clock-based profile data on the Oracle Solaris kernel.

The er_kernel utility captures kernel profile data and records the data as an Analyzer experiment in the same format as an experiment created on user programs by the collect utility. The experiment can be processed by the er_print utility or the Performance Analyzer.

A kernel experiment can show function data, caller-callee data, instruction-level data, and a timeline, but not source-line data (because most Oracle Solaris modules do not contain line-number tables).

See Chapter 9, "Kernel Profiling," for more information.

Hardware Counter Overflow Profiling Data

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

Various processor chip families support from two to eighteen simultaneous hardware counter registers. The Collector can collect data on one or more registers. For each register the Collector allows you to select the type of counter to monitor for overflow, and to set an overflow value for the counter. Some hardware counters can use any register, others are only available on a particular register. Consequently, not all combinations of hardware counters can be chosen in a single experiment.

Hardware counter overflow profiling can also be done on the kernel with the `er_kernel` utility. See Chapter 9, "Kernel Profiling," for more information.

Hardware counter overflow profiling data is converted by the Performance Analyzer into count metrics. For counters that count in cycles, the metrics reported are converted to times; for counters that do not count in cycles, the metrics reported are event counts. On machines with multiple CPUs, the clock frequency used to convert the metrics is the harmonic mean of the clock frequencies of the individual CPUs. Because each type of processor has its own set of hardware counters, and because the number of hardware counters is large, the hardware counter metrics are not listed here. The next subsection tells you how to find out what hardware counters are available.

One use of hardware counters is to diagnose problems with the flow of information into and out of the CPU. High counts of cache misses, for example, indicate that restructuring your program to improve data or text locality or to increase cache reuse can improve program performance.

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

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

When such backtracking is supported and specified during collection, hardware counter profile packets additionally include the PC (program counter) and EA (effective address) of a candidate memory-referencing instruction appropriate for the hardware counter event. (Subsequent processing during analysis is required to validate the candidate event PC and EA.) This additional information about memory-referencing events facilitates various data-oriented analyses, known as dataspace profiling. Backtracking is supported only on SPARC based platforms running the Oracle Solaris operating system.

On some SPARC chips, the counter interrupts are precise, and no backtracking is needed. Such counters are indicated by the word `precise` following the event type.

If you prepend a + sign to precise counters that are related to memory, you enable *memoryspace profiling*, which can help you to determine which program lines and memory addresses are causing memory-related program delays. See "Dataspace Profiling and Memoryspace Profiling" on page 165 for more information about memoryspace profiling.

Backtracking and recording of a candidate event PC and EA can also be specified for clock-profiling, although the data might be difficult to interpret. Backtracking on hardware counters is more reliable.

Hardware Counter Lists

Hardware counters are processor-specific, so the choice of counters available to you depends on the processor that you are using. The performance tools provide aliases for a number of counters that are likely to be in common use. You can obtain a list of available hardware counters on any particular system from the Collector by typing `collect -h` with no other arguments in a terminal window on that system. If the processor and system support hardware counter profiling, the `collect -h` command prints two lists containing information about hardware counters. The first list contains hardware counters that are aliased to common names; the second list contains raw hardware counters. If neither the performance counter subsystem nor the `collect` command know the names for the counters on a specific system, the lists are empty. In most cases, however, the counters can be specified numerically.

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

```
Aliased HW counters available for profiling:
cycles[/{0|1|2|3}],31599989 ('CPU Cycles', alias for Cycles_user; CPU-cycles)
insts[/{0|1|2|3}],31599989 ('Instructions Executed', alias for Instr_all; events)
loads[/{0|1|2|3}],9999991 ('Load Instructions', alias for Instr_ld;
      precise load-store events)
stores[/{0|1|2|3}],1000003 ('Store Instructions', alias for Instr_st;
      precise load-store events)
dcm[/{0|1|2|3}],1000003 (‘L1 D-cache Misses’, alias for DC_miss_nospec;
      precise load-store events)
...
Raw HW counters available for profiling:
...
Cycles_user[/{0|1|2|3}],1000003 (CPU-cycles)
Instr_all[/{0|1|2|3}],1000003 (events)
Instr_ld[/{0|1|2|3}],1000003 (precise load-store events)
Instr_st[/{0|1|2|3}],1000003 (precise load-store events)
DC_miss_nospec[/{0|1|2|3}],1000003 (precise load-store events)
```

Format of the Aliased Hardware Counter List

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

The second field lists the available registers for the counter; for example, `[/{0|1|2|3}]`.

The third field, for example, `9999991`, is the default overflow value for the counter. For aliased counters, the default value has been chosen to provide a reasonable sample rate. Because actual rates vary considerably, you might need to specify a non-default value.

The fourth field, in parentheses, contains type information. It provides a short description (for example, `CPU Cycles`), the raw hardware counter name (for example, `Cycles_user`), and the type of units being counted (for example, `CPU-cycles`).

Possible entries in the type information field include the following:

- `precise` – the counter interrupt occurs precisely when an instruction causes the event counter to overflow. You can prepend a + sign to the name of a load-store event counter (for example, `+dcm`) in the `collect -h` command for a precise counter to perform memoryspace profiling on ordinary binaries that were not specially compiled for profiling.
- `load`, `store`, or `load-store`, the counter is memory-related. You can prepend a + sign to the counter name (for example, `+dcrm`) in the `collect -h` command, to request a search for the precise instruction and virtual address that caused the event. The + sign also enables dataspace profiling; see “The DataObjects Tab” on page 99, “The DataLayout Tab” on page 100, and “The MemoryObjects Tabs” on page 104 for details.
- `not-program-related`, the counter captures events initiated by some other program, such as CPU-to-CPU cache snoops. Using the counter for profiling generates a warning and profiling does not record a call stack.

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

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

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

Format of the Raw Hardware Counter List

The information included in the raw hardware counter list is a subset of the information in the aliased hardware counter list. Each line in the raw hardware counter list includes the internal counter name as used by `cputrack`(1), the register numbers on which that counter can be used, the default overflow value, the type information, and the counter units, which can be either `CPU-cycles` or `events`.

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

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

Synchronization Wait Tracing Data

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

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

Synchronization tracing is not supported for Java programs.

Synchronization wait tracing data is converted into the following metrics.

TABLE 2–2 Synchronization Wait Tracing Metrics

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

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

Heap Tracing (Memory Allocation) Data

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

Heap profiling for Java programs is not supported.

Heap tracing data is converted into the following metrics.

TABLE 2–3 Memory Allocation (Heap Tracing) Metrics

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

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

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

MPI Tracing Data

The Collector can collect data on calls to the Message Passing Interface (MPI) library.

MPI tracing is implemented using the open source VampirTrace 5.5.3 release. It recognizes the following VampirTrace environment variables:

`VT_STACKS`
Controls whether or not call stacks are recorded in the data. The default setting is `1`. Setting `VT_STACKS` to `0` disables call stacks.

`VT_BUFFER_SIZE`
Controls the size of the internal buffer of the MPI API trace collector. The default value is `64M` (64 MBytes).

`VT_MAX_FLUSHES`
Controls the number of times the buffer is flushed before terminating MPI tracing. The default value is `0`, which allows the buffer to be flushed to disk whenever it is full. Setting `VT_MAX_FLUSHES` to a positive number sets a limit for the number of times the buffer is flushed.

`VT_VERBOSE`
Turns on various error and status messages. The default value is `1`, which turns on critical error and status messages. Set the variable to `2` if problems arise.

For more information on these variables, see the Vampirtrace User Manual on the Technische Universität Dresden web site.

MPI events that occur after the buffer limits have been reached are not written into the trace file resulting in an incomplete trace.

To remove the limit and get a complete trace of an application, set the `VT_MAX_FLUSHES` environment variable to 0. This setting causes the MPI API trace collector to flush the buffer to disk whenever the buffer is full.

To change the size of the buffer, set the `VT_BUFFER_SIZE` environment variable. The optimal value for this variable depends on the application that is to be traced. Setting a small value increases the memory available to the application, but triggers frequent buffer flushes by the

MPI API trace collector. These buffer flushes can significantly change the behavior of the application. On the other hand, setting a large value such as 2G minimizes buffer flushes by the MPI API trace collector, but decreases the memory available to the application. If not enough memory is available to hold the buffer and the application data, parts of the application might be swapped to disk leading to a significant change in the behavior of the application.

The functions for which data is collected are listed below.

MPI_Abort
MPI_Accumulate
MPI_Address
MPI_Allgather
MPI_Allgatherv
MPI_Allreduce
MPI_Alltoall
MPI_Alltoallv
MPI_Alltoallw
MPI_Attr_delete
MPI_Attr_get
MPI_Attr_put
MPI_Barrier
MPI_Bcast
MPI_Bsend
MPI_Bsend-init
MPI_Buffer_attach
MPI_Buffer_detach
MPI_Cancel
MPI_Cart_coords
MPI_Cart_create
MPI_Cart_get
MPI_Cart_map
MPI_Cart_rank
MPI_Cart_shift
MPI_Cart_sub
MPI_Cartdim_get
MPI_Comm_compare
MPI_Comm_create
MPI_Comm_dup
MPI_Comm_free
MPI_Comm_group
MPI_Comm_rank
MPI_Comm_remote_group
MPI_Comm_remote_size
MPI_Comm_size
MPI_Comm_split
MPI_Comm_test_inter
MPI_Dims_create
MPI_Errhandler_create
MPI_Errhandler_free
MPI_Errhandler_get
MPI_Errhandler_set
MPI_Error_class
MPI_Error_string
MPI_File_close
MPI_File_delete
MPI_File_get_amode
MPI_File_get_atomicity
MPI_File_get_byte_offset
MPI_File_get_group
MPI_File_get_info
MPI_File_get_position
MPI_File_get_position_shared
MPI_File_get_size
MPI_File_get_type_extent
MPI_File_get_view
MPI_File_iread
MPI_File_iread_at
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_at
MPI_File_iwrite_shared
MPI_File_open
MPI_File_preallocate
MPI_File_read
MPI_File_read_all
MPI_File_read_all_begin
MPI_File_read_all_end
MPI_File_read_at
MPI_File_read_at_all
MPI_File_read_at_all_begin

MPI_File_read_at_all_end
MPI_File_read_ordered
MPI_File_read_ordered_begin
MPI_File_read_ordered_end
MPI_File_read_shared
MPI_File_seek
MPI_File_seek_shared
MPI_File_set_atomicity
MPI_File_set_info
MPI_File_set_size
MPI_File_set_view
MPI_File_sync
MPI_File_write
MPI_File_write_all
MPI_File_write_all_begin
MPI_File_write_all_end
MPI_File_write_at
MPI_File_write_at_all
MPI_File_write_at_all_begin
MPI_File_write_at_all_end
MPI_File_write_ordered
MPI_File_write_ordered_begin
MPI_File_write_ordered_end
MPI_File_write_shared
MPI_Finalize
MPI_Gather
MPI_Gatherv
MPI_Get
MPI_Get_count
MPI_Get_elements
MPI_Get_processor_name
MPI_Get_version
MPI_Graph_create
MPI_Graph_get
MPI_Graph_map
MPI_Graph_neighbors
MPI_Graph_neighbors_count
MPI_Graphdims_get
MPI_Group_compare
MPI_Group_difference
MPI_Group_excl
MPI_Group_free
MPI_Group_incl
MPI_Group_intersection
MPI_Group_rank
MPI_Group_size
MPI_Group_translate_ranks
MPI_Group_union
MPI_Ibsend
MPI_Init
MPI_Init_thread
MPI_Intercomm_create
MPI_Intercomm_merge
MPI_Irecv
MPI_Irsend
MPI_Isend
MPI_Issend
MPI_Keyval_create
MPI_Keyval_free
MPI_Op_create
MPI_Op_free
MPI_Pack
MPI_Pack_size
MPI_Probe
MPI_Put
MPI_Recv
MPI_Recv_init
MPI_Reduce
MPI_Reduce_scatter
MPI_Request_free
MPI_Rsend
MPI_rsend_init
MPI_Scan
MPI_Scatter
MPI_Scatterv
MPI_Send
MPI_Send_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Ssend
MPI_Ssend_init
MPI_Start
MPI_Startall
MPI_Test
MPI_Test_cancelled
MPI_Testall
MPI_Testany

`MPI_Testsome`
`MPI_Topo_test`
`MPI_Type_commit`
`MPI_Type_contiguous`
`MPI_Type_extent`
`MPI_Type_free`
`MPI_Type_hindexed`
`MPI_Type_hvector`
`MPI_Type_indexed`
`MPI_Type_lb`
`MPI_Type_size`
`MPI_Type_struct`
`MPI_Type_ub`
`MPI_Type_vector`
`MPI_Unpack`
`MPI_Wait`
`MPI_Waitall`
`MPI_Waitany`
`MPI_Waitsome`
`MPI_Win_complete`
`MPI_Win_create`
`MPI_Win_fence`
`MPI_Win_free`
`MPI_Win_lock`
`MPI_Win_post`
`MPI_Win_start`
`MPI_Win_test`
`MPI_Win_unlock`

MPI tracing data is converted into the following metrics.

TABLE 2–4 MPI Tracing Metrics

Metric	Definition
MPI Sends	Number of MPI point-to-point sends started
MPI Bytes Sent	Number of bytes in MPI Sends
MPI Receives	Number of MPI point-to-point receives completed
MPI Bytes Received	Number of bytes in MPI Receives
MPI Time	Time spent in all calls to MPI functions
Other MPI Events	Number of calls to MPI functions that neither send nor receive point-to-point messages

MPI Time is the total thread time spent in the MPI function. If MPI state times are also collected, MPI Work Time plus MPI Wait Time for all MPI functions other than MPI_Init and MPI_Finalize should approximately equal MPI Work Time. On Linux, MPI Wait and Work are based on user+system CPU time, while MPI Time is based on real tine, so the numbers will not match.

MPI byte and message counts are currently collected only for point-to-point messages; they are not recorded for collective communication functions. The MPI Bytes Received metric counts the actual number of bytes received in all messages. MPI Bytes Sent counts the actual number of bytes sent in all messages. MPI Sends counts the number of messages sent, and MPI Receives counts the number of messages received.

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

Global (Sampling) Data

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

Sample packets are recorded in the following circumstances:

- When the program stops for any reason during debugging in the IDE or in `dbx`, such as at a breakpoint, if the option to do this is set
- At the end of a sampling interval, if you have selected periodic sampling. The sampling interval is specified as an integer in units of seconds. The default value is 1 second
- When you use the `dbx collector sample record` command to manually record a sample
- At a call to `collector_sample`, if you have put calls to this routine in your code (see "Program Control of Data Collection" on page 47)
- When a specified signal is delivered, if you have used the `-l` option with the `collect` command (see the `collect`(1) man page)
- When collection is initiated and terminated
- When you pause collection with the `dbx collector pause` command (just before the pause) and when you resume collection with the `dbx collector resume` command (just after the resume)
- Before and after a descendant process is created

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

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

How Metrics Are Assigned to Program Structure

Metrics are assigned to program instructions using the call stack that is recorded with the event-specific data. If the information is available, each instruction is mapped to a line of source code and the metrics assigned to that instruction are also assigned to the line of source code. See Chapter 6, "Understanding the Performance Analyzer and Its Data," for a more detailed explanation of how this is done.

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

Function-Level Metrics: Exclusive, Inclusive, and Attributed

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

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

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

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

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

- The attributed metrics for a function's callers tell you how much of the function's inclusive metric was due to calls from each caller. The attributed metrics for the callers sum to the function's inclusive metric.

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

The relationship between the metrics can be expressed by the following equation:

$$\sum_{\text{callers}} \begin{array}{l}\text{Attributed}\\ \text{metric}\end{array} = \begin{array}{l}\text{Inclusive}\\ \text{metric}\end{array} = \left(\sum_{\text{callees}} \begin{array}{l}\text{Attributed}\\ \text{metric}\end{array} + \begin{array}{l}\text{Exclusive}\\ \text{metric}\end{array} \right)$$

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

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

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

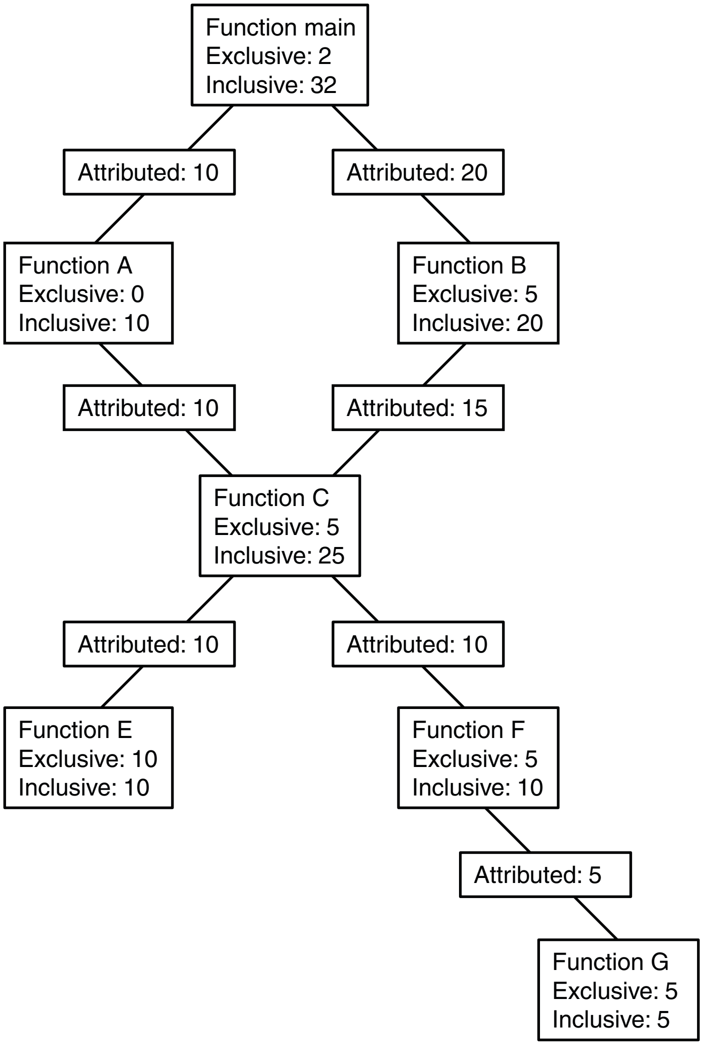
- Use exclusive metrics to locate functions that have high metric values.
- Use inclusive metrics to determine which call sequence in your program was responsible for high metric values.
- Use attributed metrics to trace a particular call sequence to the function or functions that are responsible for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

Interpreting Attributed Metrics: An Example

Exclusive, inclusive and attributed metrics are illustrated in Figure 2–1, which contains a complete call tree. The focus is on the central function, function C.

Pseudo-code of the program is shown after the diagram.

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



The Main function calls Function A and Function B, and attributes 10 units of its inclusive metric to Function A and 20 units to function B. These are the callee attributed metrics for function Main. Their sum (10+20) added to the exclusive metric of function Main equals the inclusive metric of function main (32).

Function A spends all of its time in the call to function C, so it has 0 units of exclusive metrics.

Function C is called by two functions: function A and function B, and attributes 10 units of its inclusive metric to function A and 15 units to function B. These are the caller attributed metrics. Their sum (10+15) equals the inclusive metric of function C (25)

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

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

The callee attributed metric and the callee inclusive metric are the same for function E and for function F. This means that both function E and function F are only called by function C. The exclusive metric and the inclusive metric are the same for function E but different for function F. This is because function F calls another function, Function G, but function E does not.

Pseudo-code for this program is shown below.

```
main() {
   A();
   /Do 2 units of work;/
   B();
}

A() {
   C(10);
}

B() {
   C(7.5);
   /Do 5 units of work;/
   C(7.5);
}

C(arg) {
      /Do a total of "arg" units of work, with 20% done in C itself,
      40% done by calling E, and 40% done by calling F./
}
```

How Recursion Affects Function-Level Metrics

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

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

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

◆ ◆ ◆ CHAPTER 3

Collecting Performance Data

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

Collecting data from the kernel requires a separate tool, `er_kernel`. See Chapter 9, "Kernel Profiling," for more information.

This chapter covers the following topics.

Compiling and Linking Your Program

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

Source Code Information

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

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

For memoryspace profiling of precise hardware counters on some SPARC processors, you do not need to compile with `-xhwcprof` and optimization. See "Dataspace Profiling and Memoryspace Profiling" on page 165 for more information.

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

Static Linking

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

Do not statically link any of the system libraries. If you do, you might not be able to collect any kind of tracing data. Also, do not link to the Collector library, `libcollector.so`.

Shared Object Handling

Normally the `collect` command causes data to be collected for all shared objects in the address space of the target, whether they are on the initial library list, or are explicitly loaded with `dlopen()`. However, under some circumstances some shared objects are not profiled:

- When the target program is invoked with lazy loading. In such cases, the library is not loaded at startup time, and is not loaded by explicitly calling `dlopen()`, so shared object is not included in the experiment, and all PCs from it are mapped to the `<Unknown>` function. The workaround is to set the `LD_BIND_NOW` environment variable, which forces the library to be loaded at startup time.
- When the executable was built with the `-B` option. In this case, the object is dynamically loaded by a call specifically to the dynamic linker entry point of `dlopen()()`, and the `libcollector` interposition is bypassed. The shared object name is not included in the experiment, and all PCs from it are mapped to the `<Unknown>()` function. The workaround is to not use the `-B` option.

Optimization at Compile Time

If you compile your program with optimization turned on at some level, the compiler can rearrange the order of execution so that it does not strictly follow the sequence of lines in your program. The Performance Analyzer can analyze experiments collected on optimized code, but the data it presents at the disassembly level is often difficult to relate to the original source code lines. In addition, the call sequence can appear to be different from what you expect if the compiler performs tail-call optimizations. See “Tail-Call Optimization” on page 169 for more information.

Compiling Java Programs

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

Preparing Your Program for Data Collection and Analysis

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

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

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

Using Dynamically Allocated Memory

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

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

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

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

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

The following tools may help in finding such latent bugs:

- `f95 -xcheck=init_local`

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

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

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

Using System Libraries

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

- Collecting synchronization wait tracing data. The Collector interposes on functions from the Oracle Solaris C library, `libc.so`, on Oracle Solaris.
- Collecting heap tracing data. The Collector interposes on the functions `malloc`, `realloc`, `memalign` and `free`. Versions of these functions are found in the C standard library, `libc.so` and also in other libraries such as `libmalloc.so` and `libmtmalloc.so`.
- Collecting MPI tracing data. The Collector interposes on functions from the specified MPI library.
- Ensuring the integrity of clock data. The Collector interposes on `setitimer` and prevents the program from using the profiling timer.
- Ensuring the integrity of hardware counter data. The Collector interposes on functions from the hardware counter library, `libcpc.so` and prevents the program from using the counters. Calls from the program to functions from this library return a value of `-1`.
- Enabling data collection on descendant processes. The Collector interposes on the functions `fork`(2), `fork1`(2), `vfork`(2), `fork`(3F), `posix_spawn`(3p), `posix_spawnp`(3p), `system`(3C), `system`(3F), `sh`(3F), `popen`(3C), and `exec`(2) and its variants. Calls to `vfork` are replaced internally by calls to `fork1`. These interpositions are only done for the `collect` command.
- Guaranteeing the handling of the `SIGPROF` and `SIGEMT` signals by the Collector. The Collector interposes on `sigaction` to ensure that its signal handler is the primary signal handler for these signals.

Under some circumstances the interposition does not succeed:

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

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

The `er_sync.so`, `er_heap.so`, and `er_mpview`*n*`.so` (where *n* indicates the MPI version) libraries are loaded only if synchronization wait tracing data, heap tracing data, or MPI tracing data, respectively, are requested.

Using Signal Handlers

The Collector uses two signals to collect profiling data: SIGPROF for all experiments, and SIGEMT (on Solaris platforms) or SIGIO (on Linux platforms) for hardware counter experiments only. The Collector installs a signal handler for each of these signals. The signal handler intercepts and processes its own signal, but passes other signals on to any other signal handlers that are installed. If a program installs its own signal handler for these signals, the Collector reinstalls its signal handler as the primary handler to guarantee the integrity of the performance data.

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

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

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

Using setuid and setgid

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

The collect command operates by inserting a shared library, libcollector.so, into the target's address space (LD_PRELOAD). Several problems might arise if you invoke the collect command invoked on executables that call setuid or setgid, or that create descendant processes that call setuid or setgid. If you are not root when you run an experiment, collection fails because the shared libraries are not installed in a trusted directory. The workaround is to run the experiments as root, or use crle(1) to grant permission. Take great care when circumventing security barriers; you do so at your own risk.

When running the collect command, your umask must be set to allow write permission for you, and for any users or groups that are set by the setuid attributes and setgid attributes of a program being executed with exec(), and for any user or group to which that program sets itself. If the mask is not set properly, some files might not be written to the experiment, and processing of the experiment might not be possible. If the log file can be written, an error is shown when you attempt to process the experiment.

Other problems can arise if the target itself makes any of the system calls to set UID or GID, or if it changes its umask and then forks or runs exec() on some other executable, or crle was used to configure how the runtime linker searches for shared objects.

If an experiment is started as root on a target that changes its effective GID, the er_archive process that is automatically run when the experiment terminates fails, because it needs a shared library that is not marked as trusted. In that case, you can run the er_archive utility (or the er_print utility or the analyzer command) explicitly by hand, on the machine on which the experiment was recorded, immediately following the termination of the experiment.

Program Control of Data Collection

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

The API functions are defined as follows.

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

Similar functionality is provided for Java programs by the CollectorAPI class, which is described in “The Java Interface” on page 48.

The C and C++ Interface

You can access the C and C++ interface of the Collector API by including collectorAPI.h and linking with -lcollectorAPI, which contains real functions to check for the existence of the underlying libcollector.so API functions.

If no experiment is active, the API calls are ignored.

The Fortran Interface

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

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

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

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

The Java Interface

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

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

The Java `CollectorAPI` methods are defined as follows:

```
CollectorAPI.sample(String name)
CollectorAPI.pause()
CollectorAPI.resume()
CollectorAPI.terminate()
```

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

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

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

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

If you intend to use the API functions in a multithreaded program, you should ensure that they are only called by one thread. The API functions perform actions that apply to the process and not to individual threads. If each thread calls the API functions, the data that is recorded might not be what you expect. For example, if `collector_pause()` or `collector_terminate_expt()`

is called by one thread before the other threads have reached the same point in the program, collection is paused or terminated for all threads, and data can be lost from the threads that were executing code before the API call.

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

The descriptions of the API functions follow.

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

  Fortran: `collector_sample(string name)`

  Java: `CollectorAPI.sample(String name)`

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

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

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

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

  Java: `CollectorAPI.pause()`

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

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

  Java: `CollectorAPI.resume()`

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

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

  Java: `CollectorAPI.terminate`

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

Dynamic Functions and Modules

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

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

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

The descriptions of the API functions follow.

collector_func_load()

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

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

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

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

Parameter	Definition
`lntable`	A table containing `lntsize` entries, each of which is a pair of integers. The first integer is an offset, and the second entry is a line number. All instructions between an offset in one entry and the offset given in the next entry are attributed to the line number given in the first entry. Offsets must be in increasing numeric order, but the order of line numbers is arbitrary. If `lntable` is `NULL`, no source listings of the function are possible, although disassembly listings are available.

`collector_func_unload()`

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

Limitations on Data Collection

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

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

The Collector can support up to 16K user threads. Data from additional threads is discarded, and a collector error is generated. To support more threads, set the `SP_COLLECTOR_NUMTHREADS` environment variable to a larger number.

By default, the Collector collects stacks that are, at most, up to 256 frames deep. To support deeper stacks, set the `SP_COLLECTOR_STACKBUFSZ` environment variable to a larger number.

Limitations on Clock-Based Profiling

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

On Linux systems, clock-profiling of multithreaded applications might report inaccurate data for threads. The profile signal is not always delivered by the kernel to each thread at the specified interval; sometimes the signal is delivered to the wrong thread. If available, hardware counter profiling using the cycle counter will give more accurate data for threads.

Runtime Distortion and Dilation with Clock-profiling

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

Where possible, a default value is set not to an exact number of milliseconds, but to slightly more or less than an exact number (for example, 10.007 ms or 0.997 ms) to avoid correlations with the system clock, which can also distort the data. Set custom values the same way on SPARC platforms (not possible on Linux platforms).

Limitations on Collection of Tracing Data

You cannot collect any kind of tracing data from a program that is already running unless the Collector library, `libcollector.so`, had been preloaded. See "Collecting Tracing Data From a Running Program" on page 79 for more information.

Runtime Distortion and Dilation with Tracing

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

Limitations on Hardware Counter Overflow Profiling

Hardware counter overflow profiling has several limitations:

- You can only collect hardware counter overflow data on processors that have hardware counters and that support overflow profiling. On other systems, hardware counter overflow profiling is disabled. UltraSPARC processors prior to the UltraSPARC III processor family do not support hardware counter overflow profiling.
- You cannot collect hardware counter overflow data on a system running Oracle Solaris while the `cpustat(1)` command is running, because `cpustat` takes control of the counters and does not let a user process use the counters. If `cpustat` is started during data collection, the hardware counter overflow profiling is terminated and an error is recorded in the experiment.
- You cannot use the hardware counters in your own code if you are doing hardware counter overflow profiling. The Collector interposes on the `libcpc` library functions and returns with a return value of `-1` if the call did not come from the Collector. Your program should be coded so as to work correctly if it fails to get access to the hardware counters. If not coded to handle this, the program will fail under hardware counter profiling, or if the superuser invokes system-wide tools that also use the counters, or if the counters are not supported on that system.

- If you try to collect hardware counter data on a running program that is using the hardware counter library by attaching dbx to the process, the experiment may be corrupted.

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

Runtime Distortion and Dilation With Hardware Counter Overflow Profiling

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

Limitations on Data Collection for Descendant Processes

You can collect data on descendant processes subject to some limitations.

If you want to collect data for all descendant processes that are followed by the Collector, you must use the collect command with the one of the following options:

- -F on option enables you to collect data automatically for calls to fork and its variants and exec and its variants.
- -F all option causes the Collector to follow all descendant processes, including those due to calls to system, popen, posix_spawn(3p), posix_spawnp(3p), and sh.
- -F '=*regexp*' option enables data to be collected on all descendant processes whose name or lineage matches the specified regular expression.

See "Experiment Control Options" on page 65 for more information about the -F option.

Limitations on OpenMP Profiling

Collecting OpenMP data during the execution of the program can be very expensive. You can suppress that cost by setting the SP_COLLECTOR_NO_OMP environment variable. If you do so, the program will have substantially less dilation, but you will not see the data from slave threads propagate up to the caller, and eventually to main()(), as it normally will if that variable is not set.

A new collector for OpenMP 3.0 is enabled by default in this release. It can profile programs that use explicit tasking. Programs built with earlier compilers can be profiled with the new collector only if a patched version of `libmtsk.so` is available. If this patched version is not installed, you can switch data collection to use the old collector by setting the `SP_COLLECTOR_OLDOMP` environment variable.

OpenMP profiling functionality is available only for applications compiled with the Oracle Solaris Studio compilers, since it depends on the Oracle Solaris Studio compiler runtime. For applications compiled with GNU compilers, only machine-level call stacks are displayed.

Limitations on Java Profiling

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

- You should use a version of the Java 2 Software Development Kit (JDK) no earlier than JDK 6, Update 18. The Collector first looks for the JDK in the path set in either the `JDK_HOME` environment variable or the `JAVA_PATH` environment variable. If neither of these variables is set, it looks for a JDK in your `PATH`. If there is no JDK in your `PATH`, it looks for the `java` executable in `/usr/java/bin/java`. The Collector verifies that the version of the `java` executable it finds is an ELF executable, and if it is not, an error message is printed, indicating which environment variable or path was used, and the full path name that was tried.
- To get more detailed information for source-line mappings for HotSpot-compiled code, you should use a JDK version no earlier than JDK 6, Update 20 or JDK 7, build b85 early access release.
- You must use the `collect` command to collect data. You cannot use the `dbx collector` subcommands.
- Applications that create descendant processes that run JVM software cannot be profiled.
- Some applications are not pure Java, but are C or C++ applications that invoke `dlopen()` to load `libjvm.so`, and then start the JVM software by calling into it. To profile such applications, set the `SP_COLLECTOR_USE_JAVA_OPTIONS` environment variable, and add the `-j on` option to the `collect` command line. Do not set the `LD_LIBRARY_PATH` environment variable for this scenario.

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

Java profiling uses the Java Virtual Machine Tools Interface (JVMTI), which can cause some distortion and dilation of the run.

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

Where the Data Is Stored

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

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

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

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

Experiment Names

The default name for a new experiment is `test.1.er` . The suffix `.er` is mandatory: if you give a name that does not have it, an error message is displayed and the name is not accepted.

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

Experiment Groups

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

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

```
#analyzer experiment group
```

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

You can also create an experiment group by using the `-g` argument to the `collect` command.

Experiments for Descendant Processes

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

Experiments for MPI Programs

Data for MPI programs are collected by default into `test.1.er`, and all the data from the MPI processes are collected into subexperiments, one per rank. The Collector uses the MPI rank to construct a subexperiment name with the form `M_r`*m*`.er`, where *m* is the MPI rank. For example, MPI rank 1 would have its experiment data recorded in the `test.1.er/M_r1.er` directory.

Experiments on the Kernel and User Processes

Experiments on the kernel by default are named `ktest.1.er` rather than `test.1.er`. When data is also collected on user processes, the kernel experiment contains subexperiments for each user process being followed.

The subexperiments are named using the format *_process-name*`_PID_`*process-id*`.1.er`. For example an experiment run on a `sshd` process running under process ID 1264 would be named `ktest.1.er/_sshd_PID_1264.1.er`.

Moving Experiments

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

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

See "How the Tools Find Source Code" on page 191 for a description of the process used to find an experiment's source code.

To ensure that you see the correct annotated source code and annotated disassembly code for your program, you can copy the source code, the object files and the executable into the experiment before you move or copy the experiment. You can automatically copy the load objects into the experiment using the `-A copy` option of the `collect` command or the `dbx collector archive` command.

Estimating Storage Requirements

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

The data packets contain event-specific data and data that depends on the program structure (the call stack). The amount of data that depends on the data type is approximately 50 to 100 bytes. The call stack data consists of return addresses for each call, and contains 4 bytes per address, or 8 bytes per address on 64 bit executables. Data packets are recorded for each thread in the experiment. Note that for Java programs, there are two call stacks of interest: the Java call stack and the machine call stack, which therefore result in more data being written to disk.

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

For a clock-based profiling experiment or hardware counter overflow profiling experiment with a profiling interval of about 100 samples per second, and a packet size ranging from 80 bytes for a small call stack up to 120 bytes for a large call stack, data is recorded at a rate of 10 kbytes per second per thread. Applications that have call stacks with a depth of hundreds of calls could easily record data at ten times these rates.

For MPI tracing experiments, the data volume is 100-150 bytes per traced MPI call, depending on the number of messages sent and the depth of the call stack. In addition, clock profiling is enabled by default when you use the -M option of the collect command, so add the estimated numbers for a clock profiling experiment. You can reduce data volume for MPI tracing by disabling clock profiling with the -p off option.

Note – The Collector stores MPI tracing data in its own format (mpview.dat3) and also in the VampirTrace OTF format (a.otf, a.*.z). You can remove the OTF format files without affecting the Analyzer.

Your estimate of the size of the experiment should also take into account the disk space used by the archive files, which is usually a small fraction of the total disk space requirement (see the previous section). If you are not sure how much space you need, try running your experiment for a short time. From this test you can obtain the size of the archive files, which are independent of the data collection time, and scale the size of the profile files to obtain an estimate of the size for the full-length experiment.

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

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

Collecting Data

You can collect performance data on user-mode targets in Performance Analyzer in several ways:

- Using the collect command from the command line (see "Collecting Data Using the collect Command" on page 59 and the collect(1) man page). The collect command-line tool has smaller data collection overheads than dbx so this method can be superior to the others.
- Using the Oracle Solaris Studio Performance Collect dialog box in the Performance Analyzer (see "Collecting Performance Data Using the Collect Dialog Box" in the Performance Analyzer online help).
- Using the collector command from the dbx command line (see "Collecting Data Using the dbx collector Subcommands" on page 72.

The following data collection capabilities are available only with the Oracle Solaris Studio Collect dialog box and the collect command:

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

You can collect performance data on the Oracle Solaris kernel using the er_kernel utility. See Chapter 9, "Kernel Profiling," for more information.

Collecting Data Using the collect Command

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

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

Here, *collect-options* are the collect command options, *program* is the name of the program you want to collect data on, and *program-arguments* are the program's arguments. The target program is typically a binary executable or a script.

If invoked with no arguments, collect displays a usage summary, including the default configuration of the experiment.

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

```
% collect -h
```

For a description of the list of hardware counters, see "Hardware Counter Overflow Profiling Data" on page 26. See also "Limitations on Hardware Counter Overflow Profiling" on page 52.

Data Collection Options

These options control the types of data that are collected. See "What Data the Collector Collects" on page 21 for a description of the data types.

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

If you explicitly disable clock-based profiling, and do not enable tracing or hardware counter overflow profiling, the collect command prints a warning message, and collects global data only.

`-p` *option*

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

- `off`– Turn off clock-based profiling.
- `on`– Turn on clock-based profiling with the default profiling interval of approximately 10 milliseconds.
- `lo[w]`– Turn on clock-based profiling with the low-resolution profiling interval of approximately 100 milliseconds.
- `hi[gh]`– Turn on clock-based profiling with the high-resolution profiling interval of approximately 1 millisecond. See "Limitations on Clock-Based Profiling" on page 51 for information on enabling high-resolution profiling.
- [+]*value*– Turn on clock-based profiling and set the profiling interval to *value*. The default units for *value* are milliseconds. You can specify *value* as an integer or a floating-point number. The numeric value can optionally be followed by the suffix `m` to select millisecond units or `u` to select microsecond units. The value should be a multiple of the clock resolution. If it is larger but not a multiple it is rounded down. If it is smaller, a warning message is printed and it is set to the clock resolution.

  On SPARC platforms, any value can be prepended with a + sign to enable clock-based dataspace profiling, as is done for hardware counter profiling. However, hardware counter profiling gives more reliable data.

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

See "Limitations on Clock-Based Profiling" on page 51 for notes about clock-profiling of multithreaded applications on Linux.

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

Collect hardware counter overflow profiling data. The number of counter definitions is processor-dependent.

This option is available on the Oracle Solaris and Linux operating systems running on a machine that supports hardware counters.

Note – For versions of Linux running the Linux kernel with a version greater than 2.6.32, hardware counter profiling uses the PerfEvents framework, and requires no kernel patch.

For earlier Linux systems that use the perfctr framework, you must install the required perfctr patch on the system. You can find the patch by searching for "perfctr patch" on the Web. Instructions for installation are contained within a tar file at the patch download location. The Collector searches for user-level libperfctr.so libraries using the value of the LD_LIBRARY_PATH environment variable, then in /usr/local/lib, /usr/lib, and /lib for the 32–bit versions, or /usr/local/lib64, /usr/lib64, and /lib64 for the 64–bit versions.

To obtain a list of available counters, type collect -h with no other arguments in a terminal window. A description of the counter list is given in the section "Hardware Counter Lists" on page 27. On most systems, even if a counter is not listed, you can still specify it by a numeric value, either in hexadecimal or decimal.

A counter definition can take one of the following forms, depending on whether the processor supports attributes for hardware counters.

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

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

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

- An aliased counter name
- A raw name
- A numeric value in either decimal or hexadecimal

If you specify more than one counter, they must use different registers. If they do not use different registers, the collect command prints an error message and exits.

If the hardware counter counts events that relate to memory access, you can prefix the counter name with a + sign to turn on searching for the true program counter address (PC) of the instruction that caused the counter overflow. This backtracking works on SPARC processors, and only with counters of type load , store , or load-store. If the search is successful, the virtual PC, the physical PC, and the effective address that was referenced are stored in the event data packet.

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

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

- on, or a null string– The default overflow value, which you can determine by typing collect -h with no other arguments. Note that the default value for all raw counters is the same, and might not be the most suitable value for a specific counter.
- hi[gh]– The high-resolution value for the chosen counter, which is approximately ten times shorter than the default overflow value. The abbreviation h is also supported for compatibility with previous software releases.
- lo[w]– The low-resolution value for the chosen counter, which is approximately ten times longer than the default overflow value.
- *interval*– A specific overflow value, which must be a positive integer and can be in decimal or hexadecimal format.

The default is the normal threshold, which is predefined for each counter and which appears in the counter list. See also "Limitations on Hardware Counter Overflow Profiling" on page 52.

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

-s *option*

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

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

Synchronization wait tracing data cannot be recorded for Java programs; specifying it is treated as an error.

On Oracle Solaris, the following functions are traced:

```
mutex_lock()
rw_rdlock()
rw_wrlock()
cond_wait()
cond_timedwait()
cond_reltimedwait()
thr_join()
```

```
sema_wait()
pthread_mutex_lock()
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_cond_reltimedwait_np()
pthread_join()
sem_wait()
```

On Linux, the following functions are traced:

```
pthread_mutex_lock()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_join()
sem_wait()
```

`-H` *option*

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

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

Heap tracing is turned off by default. Heap tracing is not supported for Java programs; specifying it is treated as an error.

`-M` *option*

Specify collection of an MPI experiment. The target of the `collect` command must be the `mpirun` command, and its options must be separated from the target programs to be run by the `mpirun` command by a -- option. (Always use the -- option with the `mpirun` command so that you can collect an experiment by prepending the `collect` command and its option to the `mpirun` command line.) The experiment is named as usual and is referred to as the founder experiment; its directory contains subexperiments for each of the MPI processes, named by rank.

The allowed values of *option* are:

- *MPI-version* - Turn on collection of an MPI experiment, assuming the specified MPI version which must be one of `OMPT`, `CT`, `OPENMPI`, `MPICH2`, or `MVAPICH2`. Oracle Message Passing Toolkit can be specified using `OMPT` or `CT`.
- `off` - Turn off collection of an MPI experiment.

By default, collection of an MPI experiment is turned off. When collection of an MPI experiment is turned on, the default setting for the `-m` option is changed to `on`.

The supported versions of MPI are printed when you type the `collect -h` command with no additional options, or if you specify an unrecognized version with the `-M` option.

`-m` *option*

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

- `on` – Turn on MPI tracing information.
- `off` – Turn off MPI tracing information.

MPI tracing is turned off by default unless the `-M` option is enabled, in which case MPI tracing is turned on by default. Normally MPI experiments are collected with the `-M` option, and no user control of MPI tracing is needed. If you want to collect an MPI experiment, but not collect MPI tracing data, use the explicit options `-M` *MPI-version* `-m off`.

See "MPI Tracing Data" on page 31 for more information about the MPI functions whose calls are traced and the metrics that are computed from the tracing data.

`-S` *option*

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

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

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

`-c` *option*

Record count data, for Oracle Solaris systems only.

The allowed values of *option* are

- `on`– Turn on collection of function and instruction count data. Count data and simulated count data are recorded for the executable and for any shared objects that are instrumented and that the executable statically links with. Any shared objects that are dynamically opened are not included in the simulated count data.

  In addition to count metrics for functions, lines, and so on, you can view a summary of the usage of various instructions in the Inst-Freq tab in Performance Analyzer, or with the `er_print ifreq` command.
- `off` - Turn off collection of count data.
- `static` - Generates an experiment with the assumption that every instruction in the target executable and any statically linked shared objects was executed exactly once.

By default, turn off collection of count data. Count data cannot be collected with any other type of data.

`-I` *directory*

Specify a directory for `bit` instrumentation. This option is available only on Oracle Solaris systems, and is meaningful only when the `-c` option is also specified.

`-N` *library_name*

Specify a library to be excluded from `bit` instrumentation, whether the library is linked into the executable or loaded with `dlopen()()`. This option is available only on Oracle Solaris systems, and is meaningful only when the `-c` option is also specified. You can specify multiple `-N` options.

`-r` *option*

Collect data for data race detection or deadlock detection for the Thread Analyzer. The allowed values are:

- `race` - Collect data for data race detection
- `deadlock` - Collect deadlock and potential-deadlock data
- `all` - Collect data for data race detection and deadlock detection
- `off` - Turn off thread analyzer data

For more information about the `collect -r` command and Thread Analyzer, see the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* and the `tha`(1) man page.

Experiment Control Options

These options control aspects of how the experiment data is collected.

`-F` *option*

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

- `on` – Record experiments only on descendant processes that are created by functions `fork`, `exec`, and their variants.
- `all` – Record experiments on all descendant processes.
- `off` – Do not record experiments on descendant processes.
- `=` *regexp* – Record experiments on all descendant processes whose name or lineage matches the specified regular expression.

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

For MPI experiments, descendants are also followed by default.

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

If you specify the -F '= *regexp*' option, the Collector follows all descendant processes. The Collector creates a subexperiment when the descendant name or subexperiment name matches the specified regular expression. See the regexp(5) man page for information about regular expressions.

When you collect data on descendant processes, the Collector opens a new experiment for each descendant process inside the founder experiment. These new experiments are named by adding an underscore, a letter, and a number to the experiment suffix, as follows:

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

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

For MPI experiments, the founder experiment is named using the same rules as for other experiments, but subexperiment names take the form M_r*m*.er, where *m* is the MPI rank.

The Analyzer and the er_print utility automatically read experiments for descendant processes when the founder experiment is read, and show descendants in the data display.

To select the data of a particular subexperiment for display from the command line, specify the subexperiment path name explicitly as an argument to the er_print or analyzer commands. The specified path must include the founder experiment name, and descendant experiment name inside the founder directory.

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

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

```
analyzer test.1.er/_f3.er
```

Alternatively, you can prepare an experiment group file with the explicit names of the descendant experiments in which you are interested. See "Experiment Groups" on page 55 for more information.

To examine particular descendant processes in the Analyzer, load the founder experiment and select Manage Filters from the View menu. The Manage Filters dialog box displays a list of experiments with the founder and descendent experiments checked. Uncheck all experiments except the descendant experiments of interest.

Note – If the founder process exits while descendant processes are being followed, collection of data from descendants that are still running will continue. The founder experiment directory continues to grow accordingly.

You can also collect data on scripts and follow descendant processes of scripts. See "Collecting Data From Scripts" on page 83 for more information.

`-j` *option*

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

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

You must use `-j on` to obtain profiling data if the target is a JVM machine.

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

If you cannot define the path to the `java` executable in the `JDK_HOME` or `JAVA_PATH` environment variables, or if you want to disable the recognition of methods compiled by the Java HotSpot virtual machine you can use the `-j` option. If you use this option, the *program* specified on the `collect` command line must be a Java virtual machine whose version is not earlier than JDK 6, Update 18. The `collect` command validates the version of the JVM specified for Java profiling.

`-J` *java_argument*

Specify additional arguments to be passed to the JVM used for profiling. If you specify the `-J` option, but do not specify Java profiling, an error is generated, and no experiment is run. The *java_argument* must be enclosed in quotation marks if it contains more than one argument. It must consist of a set of tokens separated by blanks or tabs. Each token is passed as a separate argument to the JVM. Most arguments to the JVM must begin with a "-" character.

`-l` *signal*

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

You can specify the signal by the full signal name, by the signal name without the initial letters SIG, or by the signal number. Do not use a signal that is used by the program or that would terminate execution. Suggested signals are SIGUSR1 and SIGUSR2. SIGPROF can be used, even when clock-profiling is specified. Signals can be delivered to a process by the kill command.

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

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

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

-t *duration*

Specify a time range for data collection.

The *duration* can be specified as a single number, with an optional m or s suffix, to indicate the time in minutes or seconds at which the experiment should be terminated. By default, the duration is in seconds. The *duration* can also be specified as two such numbers separated by a hyphen, which causes data collection to pause until the first time elapses, and at that time data collection begins. When the second time is reached, data collection terminates. If the second number is a zero, data will be collected after the initial pause until the end of the program's run. Even if the experiment is terminated, the target process is allowed to run to completion.

-x

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

-y *signal* [,r]

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

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

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

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

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

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

Output Options

These options control aspects of the experiment produced by the Collector.

`-o` *experiment_name*

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

If you do not specify the `-o` option, give the experiment a name of the form *stem*`.`*n*`.er`, where *stem* is a string, and *n* is a number. If you have specified a group name with the `-g` option, set *stem* to the group name without the `.erg` suffix. If you have not specified a group name, set *stem* to the string `test`.

If you are invoking the `collect` command from one of the commands used to run MPI jobs, for example, `mpirun`, but without the `-M` *MPI-version* option and the `-o` option, take the value of *n* used in the name from the environment variable used to define the MPI rank of that process. Otherwise, set *n* to one greater than the highest integer currently in use.

If the name is not specified in the form *stem*`.`*n*`.`*er*, and the given name is in use, an error message is displayed and the experiment is not run. If the name is of the form *stem*`.`*n*`.`*er* and the name supplied is in use, the experiment is recorded under a name corresponding to one greater than the highest value of *n* that is currently in use. A warning is displayed if the name is changed.

`-d` *directory-name*

Place the experiment in directory *directory-name*. This option only applies to individual experiments and not to experiment groups. If the directory does not exist, the `collect` utility prints an error message and exits. If a group is specified with the `-g` option, the group file is also written to *directory-name*.

For the lightest-weight data collection, it is best to record data to a local file, using the `-d` option to specify a directory in which to put the data. However, for MPI experiments on a cluster, the founder experiment must be available at the same path for all processes to have all data recorded into the founder experiment.

Experiments written to long-latency file systems are especially problematic, and might progress very slowly, especially if Sample data is collected (`-S on` option, the default). If you must record over a long-latency connection, disable Sample data.

-g *group-name*

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

-A *option*

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

- `off` – do not archive load objects into the experiment.
- `on` – archive load objects into the experiment.
- `copy` – copy and archive load objects (the target and any shared objects it uses) into the experiment.

If you expect to copy experiments to a different machine from which they were recorded, or to read the experiments from a different machine, specify `- A copy`. Using this option does not copy any source files or object (`.o`) files into the experiment. Ensure that those files are accessible and unchanged from the machine on which you are examining the experiment.

-L *size*

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

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

To impose a limit of approximately 2 Gbytes, for example, specify `-L 2000`. The size specified must be greater than zero.

By default, there is no limit on the amount of data recorded.

-O *file*

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

Other Options

These `collect` command options are used for miscellaneous purposes.

`-P` *process_id*

Write a script for `dbx` to attach to the process with the given *process_id*, collect data from it, and then invoke `dbx` on the script. You can specify only profiling data, not tracing data, and timed runs (`-t` option) are not supported.

`-C comment`

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

`-n`

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

`-R`

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

`-V`

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

`-v`

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

Collecting Data From a Running Process Using the `collect` Utility

On Oracle Solaris platforms only, the `-P` *pid* option can be used with the `collect` utility to attach to the process with the specified PID, and collect data from the process. The other options to the `collect` command are translated into a script for `dbx`, which is then invoked to collect the data. Only clock-based profile data (`-p` option) and hardware counter overflow profile data (`-h` option) can be collected. Tracing data is not supported.

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

▼ To Collect Data From a Running Process Using the collect Utility

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

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

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

2 **Use the collect command to enable data collection on the process, and set any optional parameters.**

```
% collect -P pid collect-options
```

The collector options are described in "Data Collection Options" on page 59. For information about clock-based profiling, see "-p *option*" on page 60. For information about hardware clock profiling, see -h option.

Collecting Data Using the dbx collector Subcommands

This section shows how to run the Collector from dbx, and then explains each of the subcommands that you can use with the collector command within dbx.

▼ To Run the Collector From dbx:

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

```
% dbx program
```

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

```
(dbx) collector subcommand
```

To get a listing of available collector subcommands, type:

```
(dbx) help collector
```

You must use one collector command for each subcommand.

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

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

Data Collection Subcommands

The following subcommands can be used with the collector command within dbx to control the types of data that are collected by the Collector. They are ignored with a warning if an experiment is active.

profile *option*

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

- on – Enables clock-based profiling with the default profiling interval of 10 ms.
- off – Disables clock-based profiling.
- timer *interval* - Sets the profiling interval. The allowed values of *interval* are
  - on – Use the default profiling interval of approximately 10 milliseconds.
  - lo[w] – Use the low-resolution profiling interval of approximately 100 milliseconds.
  - hi[gh] – Use the high-resolution profiling interval of approximately 1 millisecond. See “Limitations on Clock-Based Profiling” on page 51 for information on enabling high-resolution profiling.
  - *value* - Set the profiling interval to *value*. The default units for *value* are milliseconds. You can specify *value* as an integer or a floating-point number. The numeric value can optionally be followed by the suffix m to select millisecond units or u to select microsecond units. The value should be a multiple of the clock resolution. If the value is larger than the clock resolution but not a multiple it is rounded down. If the value is smaller than the clock resolution it is set to the clock resolution. In both cases a warning message is printed.

    The default setting is approximately 10 milliseconds.

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

hwprofile *option*

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

- on – Turns on hardware counter overflow profiling. The default action is to collect data for the cycles counter at the normal overflow value.

- off – Turns off hardware counter overflow profiling.
- list – Returns a list of available counters. See "Hardware Counter Lists" on page 27 for a description of the list. If your system does not support hardware counter overflow profiling, dbx returns a warning message.
- counter *counter_definition*... [, *counter_definition*] – A counter definition takes the following form.

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

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

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

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

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

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

    See also "Limitations on Hardware Counter Overflow Profiling" on page 52.

synctrace *option*

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

- on – Enable synchronization wait tracing with the default threshold.
- off – Disable synchronization wait tracing.
- threshold *value* - Sets the threshold for the minimum synchronization delay. The allowed values for *value* are:
  - all – Use a zero threshold. This option forces all synchronization events to be recorded.

- `calibrate` – Set the threshold value by calibration at runtime. (Equivalent to `on`.)
- `off` – Turn off synchronization wait tracing.
- `on` – Use the default threshold, which is to set the value by calibration at runtime. (Equivalent to `calibrate`.)
- *number* - Set the threshold to *number*, given as a positive integer in microseconds. If `value` is 0, all events are traced.

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

`heaptrace` *option*

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

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

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

`tha` *option*

Collect data for data race detection or deadlock detection for the Thread Analyzer. The allowed values are:

- `off` – Turn off thread analyzer data collection
- `all` – Collect all thread analyzer data
- `race` - Collect data-race-detection data
- `deadlock` – Collect deadlock and potential-deadlock data

For more information about the Thread Analyzer, see the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* and the `tha.1` man page.

`sample` *option*

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

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

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

`dbxsample{on|off}`

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

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

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

Experiment Control Subcommands

The following subcommands can be used with the `collector` command within `dbx` to control the collection of experiment data by the Collector. The subcommands are ignored with a warning if an experiment is active.

`disable`

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

`enable`

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

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

`pause`

Suspends the collection of data, but leaves the experiment open. Sample points are not recorded while the Collector is paused. A sample is generated prior to a pause, and another sample is generated immediately following a resume. This subcommand is ignored if data collection is already paused.

`resume`

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

`sample record` *name*

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

Output Subcommands

The following subcommands can be used with the `collector` command within `dbx` to define storage options for the experiment. The subcommands are ignored with a warning if an experiment is active.

`archive` *mode*

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

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

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

`limit` *value*

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

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

By default, the amount of data recorded is unlimited.

`store` *option*

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

- `directory` *directory-name* – Sets the directory where the experiment and any experiment group is stored. This subcommand is ignored with a warning if the directory does not exist.
- `experiment` *experiment-name* – Sets the name of the experiment. If the experiment name does not end in `.er`, the subcommand is ignored with a warning. See "Where the Data Is Stored" on page 55 for more information on experiment names and how the Collector handles them.
- `group` *group-name* – Sets the name of the experiment group. If the group name does not end in `.erg`, the subcommand is ignored with a warning. If the group already exists, the experiment is added to the group. If the directory name has been set using the `store directory` subcommand and the group name is not an absolute path, the group name is prefixed with the directory name.

Information Subcommands

The following subcommands can be used with the `collector` command within `dbx` to get reports about Collector settings and experiment status.

show

Shows the current setting of every Collector control.

status

Reports on the status of any open experiment.

Collecting Data From a Running Process With dbx on Oracle Solaris Platforms

On Oracle Solaris platforms, the Collector allows you to collect data from a running process. If the process is already under the control of dbx, you can pause the process and enable data collection using the methods described in previous sections. Starting data collection on a running process is not supported on Linux platforms.

If the process is not under the control of dbx, the collect –P *pid* command can be used to collect data from a running process, as described in "Collecting Data From a Running Process Using the collect Utility" on page 71. You can also attach dbx to it, collect performance data, and then detach from the process, leaving it to continue. If you want to collect performance data for selected descendant processes, you must attach dbx to each process.

▼ To Collect Data From a Running Process That is Not Under the Control of dbx

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

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

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

2. **Attach to the process.**

   From dbx, type the following.

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

   If dbx is not already running, type the following.

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

   Attaching to a running process pauses the process.

   See the manual *Oracle Solaris Studio 12.3: Debugging a Program With dbx* for more information about attaching to a process.

3 **Start data collection.**

From dbx, use the collector command to set up the data collection parameters and the cont command to resume execution of the process.

4 **Detach from the process.**

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

From dbx, type the following.

```
(dbx) detach
```

Collecting Tracing Data From a Running Program

If you want to collect any kind of tracing data, you must preload the Collector library, libcollector.so , before you run your program. To collect heap tracing data or synchronization wait tracing data, you must also preload er_heap.so and er_sync.so, respectively. These libraries provide wrappers to the real functions that enable data collection to take place. In addition, the Collector adds wrapper functions to other system library calls to guarantee the integrity of performance data. If you do not preload the libraries, these wrapper functions cannot be inserted. See "Using System Libraries" on page 45 for more information on how the Collector interposes on system library functions.

To preload libcollector.so, you must set both the name of the library and the path to the library using environment variables, as shown in the table below. Use the environment variable LD_PRELOAD to set the name of the library. Use the environment variables LD_LIBRARY_PATH, LD_LIBRARY_PATH_32, or LD_LIBRARY_PATH_64 to set the path to the library. LD_LIBRARY_PATH is used if the _32 and _64 variants are not defined. If you have already defined these environment variables, add new values to them.

TABLE 3–2 Environment Variable Settings for Preloading libcollector.so, er_sync.so, and er_heap.so

Environment Variable	Value
LD_PRELOAD	libcollector.so
LD_PRELOAD	er_heap.so
LD_PRELOAD	er_sync.so
LD_LIBRARY_PATH	/opt/solarisstudio12.3/lib/analyzer/runtime
LD_LIBRARY_PATH_32	/opt/solarisstudio12.3/lib/analyzer/runtime
LD_LIBRARY_PATH_64	/opt/solarisstudio12.3/lib/analyzer/v9/runtime
LD_LIBRARY_PATH_64	/opt/solarisstudio12.3/lib/analyzer/amd64/runtime

If your Oracle Solaris Studio software is not installed in /opt/solarisstudio12.3, ask your system administrator for the correct path. You can set the full path in LD_PRELOAD, but doing this can create complications when using SPARC V9 64-bit architecture.

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

Collecting Data From MPI Programs

The Collector can collect performance data from multi-process programs that use the Message Passing Interface (MPI).

The Collector supports the Oracle Message Passing Toolkit 8 (formerly known as Sun HPC ClusterTools 8) and its updates. The Collector can recognize other versions of MPI; the list of valid MPI versions is shown when you run collect -h with no additional arguments.

The Oracle Message Passing Toolkit MPI software is available at http://www.oracle.com/us/products/tools/message-passing-toolkit-070499.html for installing on Oracle Solaris 10 and Linux systems.

The Oracle Message Passing Toolkit is made available as part of the Oracle Solaris 11 release. If it is installed on your system, you can find it in /usr/openmpi. If it is not already installed on your Oracle Solaris 11 system, you can search for the package with the command pkg search openmpi if a package repository is configured for the system. See the manual *Adding and Updating Oracle Solaris 11 Software Packages* in the Oracle Solaris 11 documentation library for more information about installing software in Oracle Solaris 11.

For information about MPI and the MPI standard, see the MPI web site http://www.mcs.anl.gov/mpi/ . For more information about Open MPI, see the web site http://www.open-mpi.org/ .

To collect data from MPI jobs, you must use the collect command; the dbx collector subcommands cannot be used to start MPI data collection. Details are provided in "Running the collect Command for MPI" on page 80.

Running the collect Command for MPI

The collect command can be used to trace and profile MPI applications.

To collect data, use the following syntax:

collect [*collect-arguments*] **mpirun** [*mpirun-arguments*] **--** *program-name* [*program-arguments*]

For example, the following command runs MPI tracing and profiling on each of the 16 MPI processes, storing the data in a single MPI experiment:

```
collect -M OMPT mpirun -np 16 -- a.out 3 5
```

The -M OMPT option indicates MPI profiling is to be done and Oracle Message Passing Toolkit is the MPI version.

The initial collect process reformats the mpirun command to specify running the collect command with appropriate arguments on each of the individual MPI processes.

The -- argument immediately before the *program_name* is required for MPI profiling. If you do not include the -- argument, the collect command displays an error message and no experiment is collected.

Note – The technique of using the mpirun command to spawn explicit collect commands on the MPI processes is no longer supported for collecting MPI trace data. You can still use this technique for collecting other types of data.

Storing MPI Experiments

Because multiprocessing environments can be complex, you should be aware of some issues about storing MPI experiments when you collect performance data from MPI programs. These issues concern the efficiency of data collection and storage, and the naming of experiments. See "Where the Data Is Stored" on page 55 for information on naming experiments, including MPI experiments.

Each MPI process that collects performance data creates its own subexperiment. While an MPI process creates an experiment, it locks the experiment directory; all other MPI processes must wait until the lock is released before they can use the directory. Store your experiments on a file system that is accessible to all MPI processes.

If you do not specify an experiment name, the default experiment name is used. Within the experiment, the Collector will create one subexperiment for each MPI rank. The Collector uses the MPI rank to construct a subexperiment name with the form M_*rm*.er, where *m* is the MPI rank.

If you plan to move the experiment to a different location after it is complete, then specify the -A copy option with the collect command. To copy or move the experiment, do not use the UNIX cp or mv command; instead, use the er_cp or er_mv command as described in Chapter 8, "Manipulating Experiments."

MPI tracing creates temporary files in /tmp/a.*.z on each node. These files are removed during the MPI_finalize() function call. Make sure that the file systems have enough space for the experiments. Before collecting data on a long running MPI application, do a short duration trial run to verify file sizes. Also see "Estimating Storage Requirements" on page 57 for information on how to estimate the space needed.

MPI profiling is based on the open source VampirTrace 5.5.3 release. It recognizes several supported VampirTrace environment variables, and a new one, VT_STACKS, which controls whether or not call stacks are recorded in the data. For further information on the meaning of these variables, see the VampirTrace 5.5.3 documentation.

The default value of the environment variable VT_BUFFER_SIZE limits the internal buffer of the MPI API trace collector to 64 Mbytes. After the limit has been reached for a particular MPI process, the buffer is flushed to disk, if the VT_MAX_FLUSHES limit has not been reached. By default VT_MAX_FLUSHES is set to 0. This setting causes the MPI API trace collector to flush the buffer to disk whenever the buffer is full. If you set VT_MAX_FLUSHES to a positive number, you limit the number of flushes allowed. If the buffer fills up and cannot be flushed, events are no longer written into the trace file for that process. The result can be an incomplete experiment, and in some cases, the experiment might not be readable.

To change the size of the buffer, use the environment variable VT_BUFFER_SIZE. The optimal value for this variable depends on the application that is to be traced. Setting a small value will increase the memory available to the application but will trigger frequent buffer flushes by the MPI API trace collector. These buffer flushes can significantly change the behavior of the application. On the other hand, setting a large value, like 2 Gbytes, will minimize buffer flushes by the MPI API trace collector, but decrease the memory available to the application. If not enough memory is available to hold the buffer and the application data this might cause parts of the application to be swapped to disk leading also to a significant change in the behavior of the application.

Another important variable is VT_VERBOSE, which turns on various error and status messages. Set this variable to 2 or higher if problems arise.

Normally, MPI trace output data is post-processed when the mpirun target exits; a processed data file is written to the experiment, and information about the post-processing time is written into the experiment header. MPI post-processing is not done if MPI tracing is explicitly disabled with -m off. In the event of a failure in post-processing, an error is reported, and no MPI Tabs or MPI tracing metrics are available.

If the mpirun target does not actually invoke MPI, an experiment is still recorded, but no MPI trace data is produced. The experiment reports an MPI post-processing error, and no MPI Tabs or MPI tracing metrics will be available.

If the environment variable VT_UNIFY is set to 0, the post-processing routines are not run by collect. They are run the first time er_print or analyzer are invoked on the experiment.

Note – If you copy or move experiments between computers or nodes, you cannot view the annotated source code or source lines in the annotated disassembly code unless you have access to the source files or a copy with the same timestamp. You can put a symbolic link to the original source file in the current directory in order to see the annotated source. You can also use settings in the Set Data Presentation dialog box: the Search Path tab (see "Search Path Tab" on page 110) lets you manage a list of directories to be used for searching for source files, the Pathmaps tab (see "Pathmaps Tab" on page 111) enables you to map the leading part of a file path from one location to another.

Collecting Data From Scripts

You can specify a script as the target for the `collect` command. When the target is a script, `collect` by default collects data on the program that is launched to execute the script, and on all descendant processes.

To collect data only on a specific process, use the `-F` option to specify the name of the executable to follow.

For example, to profile the script `start.sh`, but collect data primarily from the executable `myprogram`, use the following command.

```
$ collect -F =myprogram start.sh
```

Data is collected on the founder process that is launched to execute the `start.sh` script, and on all `myprogram` processes that are spawned from the script, but not collected for other processes.

Using `collect` With `ppgsz`

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

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

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

CHAPTER 4

The Performance Analyzer Tool

The Performance Analyzer is a graphical data-analysis tool that analyzes performance data collected by the Collector. The Collector can be started from a Performance Analyzer menu option, or by using the `collect` command, or the `collector` commands in `dbx`. The Collector gathers performance information to create an experiment during the execution of a process, as described in Chapter 3, "Collecting Performance Data." The Performance Analyzer reads in such experiments, analyzes the data, and displays the data in tabular and graphical displays. A command-line tool, the `er_print` utility, is also available for displaying the experiment data in ASCII text form. See Chapter 5, "The `er_print` Command Line Performance Analysis Tool," for more information.

This chapter covers the following topics:

Starting the Performance Analyzer

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

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

The *experiment-list* command argument is a blank-separated list of experiment names, experiment group names, or both. If you do not provide an experiment list, the Analyzer starts and automatically opens the Open Experiment dialog box so you can navigate to an experiment and open it.

You can specify multiple experiments or experiment groups on the command line. If you specify an experiment that has descendant experiments inside it, all descendant experiments are automatically loaded. The data from the initial founder process and all the descendants is aggregated. To load individual descendant experiments you must specify each experiment explicitly or create an experiment group. You can also put an `en_desc` directive in an `.er.rc` file (see "`en_desc { on | off | =`*regexp*`}`" on page 148).

To create an experiment group, you can use the `-g` argument to the `collect` utility. To manually create an experiment group, create a plain text file whose first line is as follows:

```
#analyzer experiment group
```

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

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

When the Analyzer displays multiple experiments, data from all the experiments is aggregated by default. The data is combined and viewed as if the data is from one experiment. However, you can also choose to compare the experiments instead of aggregating the data. See "Comparing Experiments" on page 118.

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

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

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

The Analyzer starts up with the Collect window showing the named target and its arguments, and settings for collecting an experiment. See "Recording Experiments from Analyzer" on page 116 for details.

You can also open a "live" experiment – an experiment that is still being collected. When you open a live experiment, you see only the data that had already been collected when you opened the experiment. The experiment is not automatically updated as new data comes in. To update, you can open the experiment again.

Analyzer Command Options

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

- Java options
- Control options
- Information options

Java Options

These options specify settings for the JVM that runs the Analyzer.

-j | --jdkhome *jvm-path*

Specify the path to the JVM software for running the Analyzer. When the `-j` option is not specified, the default path is taken first by examining environment variables for a path to the JVM, in the order `JDK_HOME` and then `JAVA_PATH`. If neither environment variable is set, the JVM found on your PATH is used. Use the `-j` option to override all the default paths.

-J*jvm-options*

Specify the JVM options. You can specify multiple options. For example:

- To run the 64–bit Analyzer, type:

  ```
  analyzer -J-d64
  ```
- To run the Analyzer with a maximum of JVM memory of 2 Gbytes, type:

  ```
  analyzer -J-Xmx2G
  ```
- To run the 64–bit Analyzer with a maximum JVM memory of 8 Gbytes, type:

  ```
  analyzer -J-d64 -J-Xmx8G
  ```

Control Options

These options control the font size of the GUI, and display the version and runtime information before starting the Analyzer.

-f | --fontsize *size*

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

-v | --verbose

Print version information and Java runtime arguments before starting.

Information Options

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

-V | --version

Print version information and exit.

-? | --h | --help

Print usage information and exit.

Analyzer Default Settings

The Analyzer uses resource files named `.er.rc` to determine default values for various settings upon startup. The system wide `er.rc` defaults file is read first, then an `.er.rc` file in the user's home directory, if present, then an `.er.rc` file in the current directory, if present. Defaults from the `.er.rc` file in your home directory override the system defaults, and defaults from the `.er.rc` file in the current directory override both home and system defaults. The `.er.rc` files are used by the Analyzer and the `er_print` utility. Any settings in `.er.rc` that apply to source and disassembly compiler commentary are also used by the `er_src` utility.

See the sections "Default Settings for Analyzer" on page 117 for more information about the `.er.rc` files. See "Commands That Set Defaults" on page 146 and "Commands That Set Defaults Only For the Performance Analyzer" on page 148 for information about setting defaults with `er_print` commands.

Performance Analyzer GUI

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

The Menu Bar

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

The File menu is for opening, adding, and dropping experiments and experiment groups. The File menu allows you to collect data for an experiment using the Performance Analyzer GUI. For details on using the Performance Analyzer to collect data, refer to "Recording Experiments from Analyzer" on page 116. From the File menu, you can also create a new Analyzer window and print the data that is currently displayed in the Analyzer to a file or printer.

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

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

The Toolbar

The toolbar provides sets of icons as shortcuts, a View Mode list you can use to change the way data is displayed for some types of experiments, and a Find function to help you locate text or highlighted lines in some tabs. For more details about the Find function, refer to "Finding Text and Data" on page 112.

Analyzer Data Displays

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

The tabs in the left pane work together so that when you select an item such as a function in one tab, the item is also selected in other tabs. Most of the tabs in the left pane have a context menu that you open by right-clicking on an item in the tab. You can use the context menu to add filters or to perform other activities related to that tab. When you apply filters in one tab, the data is filtered in all the tabs that can be filtered.

The tabs in the right pane show more information about the items that you select in the left pane, and in some cases provide tools for manipulating the information in the tab in the left pane.

Data Tabs in Left Pane

Two factors determine whether a tab is displayed in the left pane of the Analyzer window when you open an experiment:

- The `.er.rc` files, which are read when you start the Analyzer, contain a `tabs` directive that specifies the default tabs to display.
- The type of data in the experiment determines what other tabs should be displayed. For example, if an experiment contains OpenMP data, the tabs for OpenMP are automatically opened to display the data.

You can use the Tabs tab in the Set Data Presentation dialog box (see "Tabs Tab" on page 111) to select the tabs you want to display in the current Analyzer session. If you want to change the default tabs, see "Default Settings for Analyzer" on page 117.

The left pane displays tabs for the principal Analyzer displays in the order in which they appear:

- "The MPI Timeline Tab" on page 90
- "MPI Chart Tab" on page 91
- "The Races Tab" on page 91
- "The Deadlocks tab" on page 91
- "The Functions Tab" on page 91
- "The Callers-Callees Tab" on page 92
- "The Call Tree Tab" on page 93
- "The Dual-Source Tab" on page 94
- "The Source/Disassembly Tab" on page 94
- "The Source Tab" on page 94
- "The Lines Tab" on page 95
- "The Disassembly Tab" on page 96
- "The PCs Tab" on page 96
- "The OpenMP Parallel Region Tab" on page 97
- "The OpenMP Task Tab" on page 97
- "The Timeline Tab" on page 98
- "The LeakList Tab" on page 99
- "The DataObjects Tab" on page 99
- "The DataLayout Tab" on page 100
- "The Inst-Freq Tab" on page 100
- "The Statistics Tab" on page 100
- "The Experiments Tab" on page 100
- "The Index Objects Tabs" on page 101
- "The Threads Tab" on page 101
- "The Samples Tab" on page 102
- "The CPUs Tab" on page 102
- "The Seconds Tab" on page 103
- "The MemoryObjects Tabs" on page 104

The MPI Timeline Tab

The MPI Timeline tab shows a set of horizontal bars, one for each process in the MPI experiment, with diagonal lines connecting them indicating messages. Each bar has regions colored according to the MPI function they are in, or indicating that the process is not within MPI (that is, it is elsewhere in the application code).

Selecting a region of a bar or a message line shows detailed information about the selection in the MPI Timeline Controls tab.

Dragging the mouse causes the MPI Timeline tab to zoom in on the horizontal (time) axis or the vertical (process) axis, depending on the predominant direction of the drag.

You can print an image of the MPI Timeline to a printer or a `.jpg` file. Choose File → Print and select Print or File, then specify the printer or filename.

See *Oracle Solaris Studio 12.3: Performance Analyzer MPI Tutorial* for more information about using the MPI Timeline.

MPI Chart Tab

The MPI Chart tab shows charts of the MPI tracing data displayed in the MPI Timeline tab. It displays plots of data concerning MPI execution. Changing the controls in the MPI Chart tab and clicking Redraw causes a new chart to be displayed. Selecting an element from a chart shows more detailed information about that element in the MPI Chart Controls tab.

Dragging the mouse causes the MPI Chart tab to zoom in on the rectangular area defined by the drag.

You can print an image of the MPI chart to a printer or a `.jpg` file. Choose File → Print and select Print or File, then specify the printer or filename.

See *Oracle Solaris Studio 12.3: Performance Analyzer MPI Tutorial* for more information about using MPI charts.

The Races Tab

The Races tab shows a list of all the data races detected in a data-race experiment. For more information, see *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide*.

The Deadlocks tab

The Deadlocks tab shows a list of all the deadlocks detected in a deadlock experiment. For more information, see *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide*.

The Functions Tab

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

The list of available metrics for each kind of data collected is given in the `collect`(1) man page and in Chapter 2, "Performance Data."

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

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

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

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

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

To search for a function, use the Find tool. For further details about the Find tool, refer to "Finding Text and Data" on page 112.

To view the source code for a function, double-click on it to open the Source tab at the correct line in the source code.

To select a single function, click on that function.

To select several functions that are displayed contiguously in the tab, select the first function of the group, then Shift-click on the last function of the group.

To select several functions that are not displayed contiguously in the tab, select the first function of the group, then select the additional functions by Ctrl-clicking on each function.

You can also right-click in the Functions tab to open a context menu and select a predefined filter for the selected functions. See the Analyzer help for details about filtering.

The Callers-Callees Tab

The Callers-Callees tab shows the calling relationships between the functions in your code, along with performance metrics. The Callers-Callees tab lets you examine metrics for code branches in detail by building call stack fragments one call at a time.

The tab shows three separate panels: the Callers panel at the top, the Stack Fragment panel in the center, and the Callees panel at the bottom. When you first open the Callers-Callees tab, the function in the Stack Fragment panel is the function that you selected in one of the other Analyzer tabs, such as the Function tab or Source tab. The Callers panel lists functions that call the function in the Stack Fragment panel, and the Callees panel lists functions that are called by the function in the Stack Fragment panel.

You can construct a call stack fragment around the center function, one call at a time, by adding callers and callees to the call stack.

You can add a call to the stack fragment by double-clicking a function in the Callers pane or Callees pane, or by selecting a function and clicking the Add button. You can remove a function call in a similar way, by double-clicking the function at the top or bottom of the call stack fragment, or selecting the top or bottom function and clicking Remove. The Add and Remove tasks can also be performed through the context menu by right-clicking a function and selecting the appropriate command.

You can set a function as the head (top), center, or tail (bottom) of the call stack fragment by selecting the function and clicking Set Head, Set Center, or Set Tail. This new ordering causes other functions currently in the call stack fragment to move to the Callers area or Callees area, to their appropriate location in relation to the new location of the selected function in the stack fragment.

You can use the Back and Forward buttons located above the Stack Fragment panel to go through the history of your changes to the call stack fragment.

As you add and remove functions in the stack fragment, the metrics are computed for the entire fragment and displayed next to the last function in the fragment.

You can select a function in any panel of the Callers-Callees tab and then right-click to open a context menu and select filters. The data is filtered according to your selection in this tab and all the Analyzer data tabs. See the online help for details about using context filters.

The Callers-Callees tab shows attributed metrics:

- For the call stack fragment in the Stack Fragment panel the attributed metric represents the exclusive metric for that call stack fragment.
- For the callees, the attributed metric represents the portion of the callee's metric that is attributable to calls from the call stack fragment. The sum of attributed metrics for the callees and the call stack fragment should add up to the metric for the call stack fragment.
- For the callers, the attributed metrics represent the portion of the call stack fragment's metric that is attributable to calls from the callers. The sum of the attributed metrics for all callers should also add up to the metric for the call stack fragment.

For more information about metrics, see “Function-Level Metrics: Exclusive, Inclusive, and Attributed” on page 36.

The Call Tree Tab

The Call Tree tab displays a dynamic call graph of the program as a tree with each function call shown as a node that you can expand and collapse. An expanded function node shows all the function calls made by the function, plus performance metrics for those function calls. When you select a node, the Summary tab on the right displays metrics for the function call and its callees. The percentages given for attributed metrics are the percentages of the total program metrics. The default root of the tree is `<Total>`, which is not a function, but represents 100% of the performance metrics of all the functions of the program.

The Call Tree tab allows you to dig down into specific call traces and analyze which traces have the greatest performance impact. You can navigate through the structure of your program, searching for high metric values.

Tip – To easily find the branch that is consuming the most time, right click any node and select Expand Hottest Branch.

You can right-click in the Call Tree tab to open a context menu and add a predefined filter for the selected branch or selected functions. By filtering in this way you can screen out data in all the Analyzer tabs for areas you are not interested in.

The Dual-Source Tab

The Dual-Source tab shows the two source contexts involved in the selected data race or deadlock. The tab is shown only if data-race-detection or deadlock experiments are loaded. See *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information.

The Source/Disassembly Tab

The Source/Disassembly tab shows the annotated source in an upper pane, and the annotated disassembly in a lower pane. The panes are coordinated so that when you select lines in one pane the related lines in the other pane are also selected. The tab is not visible by default. Use the Set Data Presentation option on the View menu to add the Source/Disassembly tab. Click Save to always show the tab.

The Source Tab

If the source code is available, the Source tab shows the file containing the source code of the selected function, annotated with performance metrics in columns to the left of each source line. High metrics are highlighted in yellow to indicate source lines that are hot areas of resource usage. A yellow navigation marker is also shown in a margin next to the scrollbar on the right for each hot source line. Non-zero metrics that are below the hot threshold are not highlighted, but are flagged with yellow navigation markers. To quickly navigate to source lines with metrics, you can click the yellow markers in the right margin to jump to the lines with metrics. You can also right-click the metrics themselves and select an option such as Next Hot Line or Next Non-Zero Metric Line to jump to the next line with metrics.

The threshold for highlighting metrics can be set in the Set Data Presentation dialog box, in the Source/Disassembly tab. The default threshold can be set in a `.er.rc` defaults file. See "Default Settings for Analyzer" on page 117 for more information about the `.er.rc` file.

The Source tab shows the full paths to the source file and the corresponding object file, and the name of the load object in the column heading for the source code. In the rare case where the same source file is used to compile more than one object file, the Source tab shows the performance data for the object file containing the selected function.

See “How the Tools Find Source Code” on page 191 for a description of the process used to find an experiment's source code.

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

<Function: f_name>

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

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

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

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

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

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

For detailed information about the content of the Source tab, see “Performance Analyzer Source Tab Layout” on page 193.

The Lines Tab

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

The Disassembly Tab

The Disassembly tab shows a disassembly listing of the object file containing the selected function, annotated with performance metrics for each instruction. You might need to select Machine from the View Mode list in the toolbar to see the disassembly listing.

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

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

The classes of commentary shown can be set in the Set Data Presentation dialog box. The default classes can be set in a `.er.rc` defaults file by clicking the Save button in the dialog box.

The Analyzer highlights hot lines, which are lines with metrics that are equal to or exceed a metric-specific threshold, to make it easier to find the important lines. You can set the threshold in the Set Data Presentation dialog box. You can set the default threshold in a `.er.rc` defaults file by clicking the Save button in the dialog box..

As with the Source tab, yellow navigation markers are shown in a margin next to the scrollbar on the right for each source line with metrics. Non-zero metrics that are below the hot threshold are not highlighted, but are flagged with yellow navigation markers. To quickly navigate to source lines with metrics, you can click the yellow markers in the right margin to jump to the lines with metrics. You can also right-click the metrics themselves and select an option such as Next Hot Line or Next Non-Zero Metric Line to jump to the next line with metrics.

For detailed information about the content of the Disassembly tab, see "Annotated Disassembly Code" on page 199.

The PCs Tab

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

See the section "Call Stacks and Program Execution" on page 167 for more information about PCs.

The OpenMP Parallel Region Tab

The OpenMP Parallel Region tab is applicable only to experiments that were recorded with the OpenMP 3.0 collector, for programs that use OpenMP tasks compiled with Oracle Solaris Studio compilers. See "Limitations on OpenMP Profiling" on page 53 for more information.

The tab lists all parallel areas encountered during the program's execution along with metric values computed from the same profiling data. Exclusive metrics are computed for the current parallel region. Inclusive metrics reflect nested parallelism. They are attributed to the current parallel region, and the parent parallel region from which it was created, and further on recursively up to the topmost Implicit OpenMP Parallel Region, representing the serial execution of the program (outside any parallel region). If there are no nested parallel regions in a program, the exclusive and inclusive metrics have the same values.

If a function containing a parallel region is called many times, all instances of the parallel region will be aggregated together and presented as one line item in the corresponding tab.

The tab is useful for navigation. You can select an item of interest, such as the parallel region with the highest OpenMP Wait time, analyze its source or select a context filter to include only the data related to the selected item, and then analyze how the data is represented by other program objects using other tabs: Functions, Timeline, Threads, and so on.

The OpenMP Task Tab

The OpenMP Task tab shows the list of OpenMP tasks with their metrics. The tab is applicable only to experiments that were recorded with the OpenMP 3.0 collector, for programs that use OpenMP tasks compiled with Oracle Solaris Studio compilers. See "Limitations on OpenMP Profiling" on page 53 for more information.

The tab lists tasks encountered during the program's execution, along with metric values computed from the profiling data. Exclusive metrics apply to the current task only. Inclusive metrics include metrics for OpenMP tasks plus those of their child tasks, with their parent-child relationship established at the task creation time. The OpenMP Task from Implicit Parallel Region represents the serial execution of the program.

If a function containing a task is called many times, all instances of the parallel region will be aggregated together and presented as one line item in the corresponding tab.

The tab is useful for navigation. You can select an item of interest, such as the task with the highest OpenMP Wait time, analyze its source by clicking the Source tab. You can also select an item, right-click to select a context filter to include only the data related to the selected item and then analyze how it's represented by other program objects using other tabs: Functions, Timeline, Threads, and so on.

The Timeline Tab

The Timeline tab shows a chart of the events and the sample points recorded by the Collector as a function of time. Data is displayed in horizontal bars. For each experiment there is a bar for sample data at the top, and a set of bars follows for each thread. The set of bars for a thread includes a bar for each data type recorded: clock-based profiling, hardware counter overflow profiling, synchronization tracing, heap tracing, and MPI tracing. For hardware counter overflow profiling, there is a bar for each hardware counter. The bar for each data type is displayed only when loaded experiments contain data of that type.

The timing metrics for events shown by the Performance Analyzer correspond to the relative amount of time spent in each state. The samples bar displays a summary of the timing metrics for all the threads in the experiment. Click a sample to display the timing metrics for that sample in the Timeline Details tab on the right. You can also display timing metrics for individual events in the threads by displaying event state charts.

The profiling data or tracing data bars show an event marker for each event recorded. The event markers consist of a color-coded representation of the call stack recorded with the event.

The profiling data or tracing data bars show an event marker for each event recorded. The event markers consist of a color-coded representation of the call stack recorded with the event. Clicking an event marker selects the corresponding call stack for the event and displays the data for that event and the call stack functions in the Timeline Details tab. In the Call Stack for Selected Event area of the Timeline Details tab you can select the individual function calls to see detailed information about them in the Details for Selected Event area of the tab. If you then click the Source tab or Disassembly tab, the line corresponding to that frame in the call stack is highlighted.

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

The Timeline tab of the Set Data Presentation dialog box allows you to change the types of event-specific data that are shown; to select the display of event-specific data for threads, LWPs, CPUs, or experiments; to choose to align the call stack representation at the root or at the leaf; and to choose the number of levels of the call stack that are displayed. You can also select to display event frequency charts and event state charts in some timeline bars.

An *event frequency chart* is a line chart that displays frequency of events as a function of time. To enable event frequency charts, select Event Frequency Chart in the Timeline tab of the Set Data Presentation dialog box. For each enabled data type, an event frequency chart is then displayed immediately below the associated timeline data bar.

An *event state chart* is a bar chart that shows the distribution of application time spent in various states as a function of time. For clock profiling data recorded on Oracle Solaris, the event state chart shows Oracle Solaris microstates. The color coding for the event state chart is the same as for the timeline samples bar. To enable event state charts, select Event State Chart in the Timeline tab of the Set Data Presentation dialog box. Event state charts are then shown below the associated data bars. If event frequency charts are also enabled, the state chart is displayed below the frequency chart.

For details about using the Timeline tab, refer to the online help.

The LeakList Tab

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

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

You can display the LeakList tab by selecting it in the Tabs tab of the Set Data Presentation dialog box (see "Tabs Tab" on page 111). You can make the LeakList tab visible only if one or more of the loaded experiments contains heap trace data.

The DataObjects Tab

The DataObjects tab shows the list of data objects with their metrics. The tab applies only to experiments that include dataspace profiling, which is an extension of hardware counter overflow profiling. See "Dataspace Profiling and Memoryspace Profiling" on page 165 for more information.

You can display the tab by selecting it in the Tabs tab of the Set Data Presentation dialog box (see "Tabs Tab" on page 111). You can make the DataObjects tab visible only if one or more of the loaded experiments contains a dataspace profile.

The tab shows hardware counter memory operation metrics against the various data structures and variables in the program.

To select a single data object, click on that object.

To select several objects that are displayed contiguously in the tab, select the first object, then press Shift while clicking on the last object.

To select several objects that are not displayed contiguously in the tab, select the first object, then select the additional objects by pressing Ctrl while clicking on each object.

The DataLayout Tab

The DataLayout tab shows the annotated data object layouts for all program data objects with data-derived metric data. The tab is applicable only to experiments that include dataspace profiling, which is an extension of hardware counter overflow profiling. See "Dataspace Profiling and Memoryspace Profiling" on page 165 for more information.

The layouts appear in the tab sorted by the data sort metrics values for the structure as a whole. The tab shows each aggregate data object with the total metrics attributed to it, followed by all of its elements in offset order. Each element, in turn, has its own metrics and an indicator of its size and location in 32–byte blocks.

The DataLayout tab can be displayed by selecting it in the Tabs tab of the Set Data Presentation dialog box (see "Tabs Tab" on page 111). As with the DataObjects tab, you can make the DataLayout tab visible only if one or more of the loaded experiments contains a dataspace profile.

To select a single data object, click on that object.

To select several objects that are displayed contiguously in the tab, select the first object, then press the Shift key while clicking on the last object.

To select several objects that are not displayed contiguously in the tab, select the first object, then select the additional objects by pressing the Ctrl key while clicking on each object.

The Inst-Freq Tab

The Inst-Freq, or instruction-frequency, tab shows a summary of the frequency with which each type of instruction was executed in a count-data experiment, which is collected with `collect -c`. The tab also shows data about the frequency of execution of load, store, and floating-point instructions. In addition, the tab includes information about annulled instructions and instructions in a branch delay slot.

The Statistics Tab

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

The Experiments Tab

The Experiments tab is divided into two panels. The top panel contains a tree that includes nodes for the load objects in all the loaded experiments, and for each experiment loaded. When you expand the Load Objects node, a list of all load objects is displayed with various messages about their processing. When you expand the node for an experiment, two areas are displayed: a Notes area and an Info area.

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

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

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

The Index Objects Tabs

Each Index Objects tab shows the metric values from data attributed to various index objects, such as Threads, Cpus, and Seconds. Inclusive and Exclusive metrics are not shown, since Index objects are not hierarchical. Only a single metric of each type is shown.

Several Index Objects tabs are predefined: Threads, Cpus, Samples, Seconds, Processes, and Experiment IDs. These tabs are described separately below.

You can define a custom index object by clicking on the Add Custom Index Tab button in the Set Data Presentation dialog box to open the Add Index Objects dialog box. You can also define an index object with an `indxobj_define` directive in an `.er.rc` file (see "`indxobj_define` *indxobj_type index_exp*" on page 136).

The Threads Tab

The Threads tab is an Index Object tab that is not displayed by default. You can select it in the Tabs tab in the Set Data Presentation dialog box. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The Threads tab shows a list of threads and their metrics. The threads are represented by a thread number and show the User CPU time by default for clock-profiling experiments. Other metrics might also be displayed by default if the metrics are present in the loaded experiments.

You can use radio buttons to select the type of display: Text (the default), Graphical, or Chart. The Text display is similar to the Functions tab but shows only exclusive metrics attributed to each thread. The Graphical display shows bar graphs of the relative values for each thread with a separate histogram for each metric. The histogram is sorted by the data sort metric that is in effect in the Text display mode. For the Text and Graphical displays, you can click with the mouse or use arrow keys to navigate the tab and see more information in Summary tab on the right about the currently selected item.

The Chart display shows charts of threads versus other metrics, such as User CPU time or Sync Time. By default, a Load Imbalance chart is shown, which helps detect threads that used much higher CPU time.

The Load Imbalance chart shows the amount of real time lost due to uneven thread loading, expressed in seconds and percent in the Threads Chart Controls tab on the right. If the threads were perfectly balanced, there would be a performance improvement close to these values. Better load balance might be achieved by dividing the task being performed by the heavily loaded threads into shorter tasks and distributing them among available threads.

The Lock Contention chart shows the distribution of the Sync Wait Time across threads, which helps to see how much time is lost because of synchronization between threads, and which threads lost more than others.

To change the type of chart, select the Chart Name in the Threads Chart Controls tab on the right and click Redraw.

Click the bars in the chart or click the Up and Down buttons in the navigation bar in the Threads Chart Controls tab to select particular threads and view metric information in the Threads Chart Controls tab.

You can also right-click and select navigation options from the context menu.

The Samples Tab

The Samples tab is an Index Object tab that is not displayed by default. You can select it in the Tabs tab in the Set Data Presentation dialog box. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The Samples tab shows a list of sample points and their metrics. The Samples are represented by Sample numbers and show the User CPU time by default for clock-profiling experiments. Other metrics might also be displayed if the metrics are present in the loaded experiments.

You can use radio buttons to select the type of display: Text or Graphical. The Text mode is displayed by default and shows a display similar to the Functions tab. The metrics values reflect the microstates recorded at each sample point in the loaded experiments.

The Graphical display shows bar graphs of the relative values for each Sample with a separate histogram for each metric. The histogram is sorted by the data sort metric that is in effect in the Text display mode.

You can also right-click to add context filters to filter the data shown in this and other Analyzer tabs.

The CPUs Tab

The CPUs tab is an Index Object tab that is not displayed by default. You can select it in the Tabs tab in the Set Data Presentation dialog box. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The CPUs tab shows a list of CPUs that the experiment ran on, along with their associated metrics. The CPUs are represented by a CPU number and show the User CPU time by default for clock-profiling experiments. Other metrics might also be displayed by default if the metrics are present in the loaded experiments. If you have selected other metrics in the Set Data Presentation dialog box, they are also displayed.

You can use radio buttons to select the type of display: Text or Graphical. The Text mode is displayed by default and shows a display similar to the Functions tab. The values reflect the value or percentage of the metrics that were recorded on each CPU in the loaded experiments.

The Graphical display shows bar graphs of the relative values for each CPU with a separate histogram for each metric. The histogram is sorted by the data sort metric that is in effect in the Text display mode.

You can right-click to add context filters to filter the data shown in this and other Analyzer tabs.

The Seconds Tab

The Seconds tab is an Index Object tab that is not displayed by default. You can select it in the Tabs tab in the Set Data Presentation dialog box. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The Seconds tab shows each second of the program run that was captured in the experiment, along with a metrics collected in that second. Seconds differ from Samples in that they are periodic samples that occur every second beginning at 0 and the interval cannot be changed. The Seconds tab lists the seconds of execution with the User CPU time by default for clock-profiling experiments. Other metrics might be displayed if the metrics are present in the loaded experiments.

You can use radio buttons to select the type of display: Text or Graphical. The Text mode is displayed by default and shows a display similar to the Functions tab. The metric values reflect the microstates recorded at each second in the loaded experiments.

The Graphical display shows bar graphs of the relative values for each second with a separate histogram for each metric. The histogram is sorted by the data sort metric that is in effect in the Text display mode.

You can right-click to add context filters to filter the data shown in this and other Analyzer tabs.

The Processes Tab

The Processes tab is an Index Object tab that is not displayed by default. You can select it in the Tabs tab in the Set Data Presentation dialog box. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The Processes tab shows a list of processes that the target program ran, along with their associated metrics. The processes are listed by their PIDs and show the User CPU time by default for clock-profiling experiments. Other metrics might also be displayed by default if the metrics are present in the loaded experiments. If you have selected other metrics in the Set Data Presentation dialog box, they are also displayed.

The Processes tab enables you find the processes that used the most resources. If there is a particular set of processes that you want to isolate and explore using other tabs, you can filter out other processes using the filters available in the context menu.

You can use radio buttons to select the type of display: Text or Graphical. The Text mode is displayed by default and shows a display similar to the Functions tab. The values reflect the value or percentage of the metrics that were recorded on each process in the loaded experiments.

The Graphical display shows bar graphs of the relative values for each PID with a separate histogram for each metric. The histogram is sorted by the data sort metric that is in effect in the Text display mode.

The Experiment IDs Tab

The Experiment IDs tab is an Index Object tab that is not displayed by default. You can select it in the Tabs tab in the Set Data Presentation dialog box. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The Experiment IDs tab shows a list of experiments that are loaded, along with their associated metrics. You can filter out other experiments using the filters available in the context menu.

You can use radio buttons to select the type of display: Text or Graphical. The Text mode is displayed by default and shows a display similar to the Functions tab. The values reflect the value or percentage of the metrics that were recorded on each process in the loaded experiments.

The Graphical display shows bar graphs of the relative values for each PID with a separate histogram for each metric. The histogram is sorted by the data sort metric that is in effect in the Text display mode.

The MemoryObjects Tabs

Each MemoryObjects tab shows the metric values for dataspace metrics attributed to the various memory objects such as pages. If one or more of the loaded experiments contains a dataspace profile, you can select the memory objects for which you want to display tabs in the Tabs tab of the Set Data Presentation dialog box. Any number of MemoryObjects tabs can be displayed.

Various MemoryObjects tabs are predefined. Memory objects are predefined for virtual and physical pages with names such as Vpage_8K, Ppage_8K, Vpage_64K, and so on. You can define a custom memory object by clicking the Add Custom Object button in the Set Data Presentation dialog box to open the Add Memory Objects dialog box. You can also define a memory object with a `mobj_define` directive in an `.er.rc` file (see "`mobj_define` *mobj_type index_exp*" on page 135).

A radio button on each MemoryObjects tab lets you select either a Text display or a Graphical display. The Text display is similar to the display in the DataObjects tab and uses the same metric settings. The Graphical display shows a graphical representation of the relative values for each memory object, with a separate histogram for each metric sorted by the data sort metric.

Tabs in Right Pane

The right pane contains the Summary tab, the Timeline Details tab, the Threads Chart Controls tab, the MPI Timeline Controls tab, the MPI Chart Controls tab, the Race Detail tab, Deadlock Detail tab, and Leak tab. By default the Summary tab is displayed.

The MPI Timeline Controls Tab

The MPI Timeline Controls tab supports zoom, pan, event-step, and filtering for the MPI Timeline tab. It includes a control to adjust the percentage of MPI messages shown on MPI Timeline tab.

Filtering causes data outside the current field of view to be eliminated from the data set shown in the MPI Timeline tab and the MPI Chart tab. A filter is applied by clicking the Filter button. The back-filter button is used to undo the last filter; the forward-filter button is used to reapply a filter. Filters are shared between the MPI Timeline tab and the MPI Chart tab, but are not currently applied to other tabs.

The message slider can be adjusted to control the percentage of messages displayed. When you select less than 100%, priority is given to the most costly messages. Cost is defined as the time spent in the message's send and receive functions.

The MPI Timeline Controls tab is also used to show the details for a function or message selection from the MPI Timeline tab.

The MPI Chart Controls Tab

The MPI Chart Controls tab has a set of drop-down lists to control the type of chart, the parameters for the X and Y axes, and the Metric and Operator used to aggregate the data. Clicking Redraw causes a new graph to be drawn.

Filtering causes data outside the current field of view to be eliminated from the data set shown in the MPI Timeline tab and MPI Chart tab. A filter is applied by clicking the Filter button. The back-filter button is used to undo the last filter; the forward-filter button is used to reapply a filter.

The MPI Chart Controls tab is also used to show the details for a selection from the MPI Chart tab.

The Summary Tab

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

The Timeline Details Tab

The Timeline Details tab shows detailed data for the event that is selected in the Timeline tab, including the event type, leaf function, LWP ID, thread ID, and CPU ID. Below the data panel the call stack is displayed. The program counter (PC) for each frame in the call stack is displayed as a function plus an offset. The color coding that is used for the functions in the event markers is shown to the left of the PCs. Clicking a function in the call stack makes it the selected function, so it is also selected in the main Analyzer tabs such as the Function tab. Double-clicking a function in the call stack brings up the Function Color Chooser which enables you to change the colors used for functions in the Timeline.

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

The Threads Chart Controls Tab

The Threads Chart Controls tab is displayed when you select a Chart display in the Threads tab.

By default, a Load Imbalance chart is shown, which helps detect threads that used much higher CPU time. To view a different type of chart, select the Chart Name in the Threads Chart Controls tab on the right and click Redraw.

The Load Imbalance chart shows the amount of real time lost due to uneven thread loading, expressed in seconds and percent, displayed in the Details area. If the threads were perfectly balanced, there would be a performance improvement close to these values. Better load balance might be achieved by dividing the task being performed by the heavily loaded threads into shorter tasks and distributing them among available threads.

You can use the Threads Chart Controls tab to navigate among the threads in the chart by clicking the Up and Down buttons in the navigation bar. When you select a thread, metric information is displayed in the Details area of the Threads Chart Controls tab.

You can click the Center button in the navigation bar to redisplay the metrics for the load imbalance in the Details area.

The Lock Contention chart shows the distribution of the Sync Wait Time across threads, which helps to see how much time is lost because of synchronization between threads, and which threads lost more than others.

The Leak Tab

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

The Race Detail Tab

The Race Detail tab shows detailed data for the selected data race in the Races tab. See the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information.

The Deadlock Detail Tab

The Deadlock Detail tab shows detailed data for the selected Deadlock in the Deadlocks tab. See the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information.

Setting Data Presentation Options

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

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

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

The OK button in this dialog box applies the changes you made for the current session, and closes the dialog box. The Apply button applies the changes for the current session, but keeps the dialog box open so you can make more changes.

The Save button stores the current settings including the tabs displayed, and any custom-defined memory objects, to a `.er.rc` file in your home directory or the current working directory. Saving the settings causes the changes you made to apply to future Analyzer sessions as well as the current session.

Note – The `.er.rc` file provides the default settings for the Analyzer, the `er_print` utility, and the `er_src` utility. When you save changes in the Set Data Preferences dialog box, the changes update the `.er.rc` file which affects the output from all three utilities. See "Default Settings for Analyzer" on page 117 for more information about `.er.rc` file.

Metrics Tab

The Metrics tab allows you to choose the metrics that are displayed in most of the Analyzer tabs including Functions, Callers-Callees, Source, Disassembly, and others. Some metrics can be displayed in your choice of time or percentage, while others are displayed as a value. The list of metrics includes all metrics that are available in any of the experiments that are loaded.

For each metric, check boxes are provided for Time and Percentage, or Value. Select the check boxes for the types of metrics that you want the Analyzer to display. If you want to select or deselect all the metrics in a particular column, select the check boxes in the bottom row of the dialog box and then click the Apply to all metrics button.

Note – You can only choose to display exclusive and inclusive metrics. Attributed metrics are always displayed in the Call Tree tab if either the exclusive metric or the inclusive metric is displayed.

Sort Tab

The Sort tab shows the order of the metrics presented, and the choice of metric to sort by. Double-click on the metric to use for sorting, and use the Move Up and Move Down buttons to change the order of the metrics. The choices you make here determine the default presentation of the metrics in the Analyzer data tabs. While viewing data tabs, you can change the sort order by clicking the column heading of the metric you want to sort by. You can change the order of the columns of metrics by dragging the column headings.

Source/Disassembly Tab

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

- The compiler commentary that is shown in the Source and Disassembly tabs.
- The threshold for highlighting important lines in the Source and Disassembly tabs.

  The highlight threshold is the percentage of the largest metric value attributed to any line in the file whose source or disassembly is displayed. The threshold is applied to each metric independently.
- The display of source code in the Disassembly tab.

  If you display source code in the Disassembly tab, the compiler commentary is also displayed.

- The metrics on the source lines in the Disassembly tab.

  If you display metrics for source lines in the disassembly listing, the Find tool finds high-metric items on source lines as well as on disassembly lines. If you want to limit the search to high-metric items on disassembly lines, clear the Metrics for Source Lines checkbox.
- The display of instructions in hexadecimal in the Disassembly tab.

Formats Tab

The Formats tab presents a choice for the long form, short form, or mangled form of C++ function names and Java method names. Mangled function names are compiler-generated names created when using compiler optimization, which the Analyzer “demangles” when you use the long form or short form function names.

If you select the Append SO name to Function name checkbox, the name of the shared object in which the function or method is located is appended to the function name or method name.

If you select the Compare Experiments option, when you load multiple experiments the data of the experiments is shown in separate columns. By default the data is aggregated when multiple experiments are loaded. See “Comparing Experiments” on page 118 for more information.

The Formats tab also presents a choice for View Mode of User, Expert, or Machine to set the default mode for viewing experiments. You can switch the current view using the View Mode list in the toolbar. The View Mode setting controls the view of only Java experiments and OpenMP experiments.

For Java experiments:

- User view mode shows each method by name, with data for interpreted and HotSpot-compiled methods aggregated together; it also suppresses data for non-user-Java threads.
- Expert view mode separates HotSpot-compiled methods from interpreted methods, and shows non-user Java threads.
- Machine view mode shows data for interpreted Java methods against the JVM machine as it does the interpreting, while data for methods compiled with the Java HotSpot virtual machine is reported for named methods. All threads are shown.

See “Java Profiling View Modes” on page 172 for more detailed descriptions of the view modes for Java experiments.

For OpenMP experiments:

- User view mode shows slave threads as if they were cloned from the master thread, with call stacks matching those from the master thread. Frames in the call stack that come from the OpenMP runtime code (`libmtsk.so`) are suppressed. Special functions with names of the form `<OMP-*>` are shown when the OpenMP runtime is performing certain operations.
- Expert view mode shows master and slave threads differently, and shows functions generated by the compiler. Frames in the call stack that come from `libmtsk.so` are suppressed.
- Machine view mode shows native call stacks for all threads and outline functions generated by the compiler.

See "Overview of OpenMP Software Execution" on page 173 for more detailed descriptions of the view modes for OpenMP experiments.

For all other experiments, all three modes show the same data.

Timeline Tab

The Timeline tab of the Set Data Presentation dialog box presents choices for the information displayed in the Timeline tab in the main Analyzer panel. You can select the types of event-specific data that are shown, the display of event-specific data for threads, LWP, CPUs, or experiments; the alignment of the call stack representation at the root or at the leaf; and the number of levels of the call stack that are displayed. You can also select Event Frequency Charts and Event State Charts, which are explained in "The Timeline Tab" on page 98.

Click the Help button in the Set Data Presentation dialog box for more detailed information about the Timeline tab.

Search Path Tab

The Search Path tab sets the path used for finding the loaded experiment's associated source and object files for displaying annotated source data in the Source and Disassembly tabs. The search path is also used to locate the `.jar` files for the Java Runtime Environment on your system. The special directory name `$expts` refers to the set of current experiments, in the order in which they were loaded. Only the founder experiment is looked at when searching `$expts`, no descendant experiments are examined.

By default the search path is set to `$expts` and `.` (the current directory). You can add other paths to search by typing or browsing to the path and clicking Append. To edit paths in the list, select a path, edit it in the Paths field, and click Update. To change the search order, select a path in the list and click the Move Up/ Move Down buttons.

See "How the Tools Find Source Code" on page 191 for more information about how the search path is used.

Pathmaps Tab

The Pathmaps tab enables you to map the leading part of a file path from one location to another to help the Analyzer locate source files. A path map is useful for an experiment that has been moved from the original location it was recorded or is being viewed from a different machine with a different path to the files on a network. When the source can be found the Analyzer can display annotated data in the Source and Disassembly tabs.

For example, if the experiment contains paths specified as `/a/b/c/d/sourcefile` and `soucefile` is now located in `/x`, you can use the Pathmaps tab to map `/a/b/c/d/` to `/x/`. Multiple path maps can be specified, and each is tried in order to find a file.

See "How the Tools Find Source Code" on page 191 for more information about how the path maps are used.

Tabs Tab

You can use the Tabs tab of the Set Data Presentation dialog box to select the tabs to be displayed in the Analyzer window. If you click the Save button in this dialog box, the selections are saved as defaults to your `.er.rc` file.

The Tabs tab lists the applicable tabs for the current experiment. The standard tabs are listed in the left column. The Index Objects tabs are listed in the center column, and the defined Memory Objects tabs are listed in the right column.

In the left column, click the checkboxes to select or deselect standard tabs for display.

In the center column, click the check boxes to select or deselect Index Objects tabs for display. The predefined Index Objects tabs are Threads, Cpus, Samples, and Seconds. To add a tab for another index object, click the Add Custom Index Tab button to open the Add Index Object dialog. In the Object name text box, type the name of the new object. In the Formula text box, type an index expression to be used to map the recorded physical address or virtual address to the object index. For information on the rules for index expressions, see "`indxobj_define` *indxobj_type index_exp*" on page 136

In the right column, click the check boxes to select or deselect Memory Object tabs for display. These tabs have data if the experiment contains hardware counter overflow profiling data. Memory objects represent components in the memory subsystem, such as cache lines, pages, and memory banks. Memory objects are predefined for virtual and physical pages, for sizes of 8KB, 64KB, 512KB, and 4MB. To add a custom object, click the Add Custom Object button to open the Add Memory Object dialog box. In the Object name text box, type the name of the new custom memory object. In the Formula text box, type an index expression to be used to map the recorded physical address or virtual address to the object index. For information on the rules for index expressions, see "`mobj_define` *mobj_type index_exp*" on page 135

When you have added a custom index object or memory object, a checkbox for that object is added to the Tabs tab and is selected by default.

Finding Text and Data

The Analyzer toolbar includes a Find tool with two options for search targets that are given in a drop-down list. You can search for text in the Name column of the Functions tab or Callers-Callees tabs and in the code column of the Source tab, Disassembly tab, and Source/Disassembly tab.

In the Source tab, Disassembly tab, and Source/Disassembly tab, you can also select High Metric Value from the Find drop-down list and use Next and Previous buttons to search for a high-metric item. The metric values on the lines containing high-metric items are highlighted in yellow.

Showing or Hiding Functions

By default, all functions in each load object are shown in the Functions tab and Callers-Callees tab. You can hide all the functions in a load object or show only those function representing the API into the load object using the Show/Hide/API-only Functions dialog box; see the Analyzer help for details.

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

When only the API functions in a load object are shown, only those functions representing calls into the library are shown, and all calls below those functions, whether within that load object, or into other load objects, including callbacks, are not shown. The Callers-Callees tab will never show callees from such functions.

The settings for load objects can be preset with command in a `.er.rc` file.

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

Filtering Data

The Performance Analyzer offers several ways to filter data so you can focus on areas of interest in your program. When you filter, you specify criteria for the data you want to view. When the filter is applied, the data that does not meet the criteria is removed from view in all the Analyzer tabs that support filtering.

When you first open experiments in the Performance Analyzer, you see data for all functions in all loaded experiments, and for all threads and CPUs for each experiment, and for the complete time period covered by the experiment.

You can filter by the following:

- Experiments - select which of the currently loaded experiments whose metrics you want to see
- Threads - select the data collected from specific thread IDs
- CPUs - select the data collected from specific CPU IDs
- Samples - select the data collected at particular sample points
- Call stacks - select the data collected from call stacks that include a particular function
- Call tree branches - select the data collected from call stacks from a particular branch of a call tree
- Time range - select the data collected during a particular time range
- Labels - select the data specified by a label that identifies a particular period of time

You can filter from multiple locations:

- In data tabs such as Functions, Callers-Callees, and Timeline, you can select a predefined filter from a popup menu by right-clicking on data such as a function name in the tab. The filters are called *context filters*.
- Using the Experiments and General tabs of the Manage Filters dialog box, you can select experiments, threads, CPUs, or samples whose data you want to display.
- Using the Custom tab of the Manage Filters dialog box, you can edit filter expressions to create custom filters to precisely define the data to be displayed.

When you set a filter, the data is filtered in all the Analyzer tabs. You can combine filters to display metrics from highly specific areas of your program's run.

Note – The filters described here are independent of the MPI filtering described in "The MPI Timeline Controls Tab" on page 105 and "The MPI Chart Controls Tab" on page 105. These filters do not affect the MPI Timeline tab and the MPI Chart tab.

Using Context Filters

Context filters are context-specific filters that are available in several data tabs in the Performance Analyzer. You access them by right-clicking with the mouse or by pressing Shift-F10 on the keyboard. When you select a context filter, the data is immediately filtered.

In general, you use context filters by selecting one or more items in the tab that you want to focus on, right-clicking the mouse, and selecting the appropriate filter. For most data tabs the filters enable you to include or *not* include the data that meets the criteria named in the filter. This enables you to use the filters to either focus on data from a particular area of your program, or exclude data from a particular area.

Some ways you can use context filters include:

- Add multiple filters to narrow down the data. The filters are combined in a logical AND relationship, which requires the data to match all of the filters.
- Add a filter in one tab and then view the filtered data in another tab. For example, in the Call Tree tab you can find the hottest branch, then select "Add Filter: Include only stacks containing the selected branch", and then go to the Functions tab to view metrics for the functions called in that code branch.
- Add multiple filters from multiple tabs to create a very specific set of data.
- Use context filters as a basis for creating custom filters. See "Using Custom Filters" on page 114.

Managing Filters

You can use the Manage Filters dialog box to perform simple filtering on experiments, threads, CPUs, and samples. This filtering can be used together with context filters.

The Manage Filters dialog box can be accessed in multiple ways:

- Choose View ⇒ Manage Filters
- Click the Manage Filters button in the toolbar
- Right-click in a data tab such as the Functions tab, and select Manage Filters from the context menu
- Right-click in the Experiments tab and select Filter Experiments

The Manage Filters dialog box enables you to do general filtering to select data to display from particular experiments, sample points, threads, and CPUs. You use the Experiments tab and the General tab together to specify the items whose data you want to display. The selections made in the Experiments and General tabs are combined for filtering using a logical AND (the && operator). Data must match all the selections made in these tabs to be included in the data tabs.

The Analyzer help includes instructions for using the Manage Filters dialog box.

The dialog box also provides a Custom tab that you can use to customize filters. See the following section for more information.

Using Custom Filters

The Custom tab enables you to type in your own filter expressions or edit previously applied filters to create custom filters. The filters in the Custom tab are separate from the filters applied through the Experiments and General tabs of the Manage Filters dialog box.

When you select context filters in Performance Analyzer data tabs, filter expressions are generated and are immediately applied to filter the data. The generated filter expressions are also added to a text box in the Custom tab, where you can use them as a beginning point for creating custom filters.

You can edit the filters, and use the arrow buttons above the Filter Specification panel to undo and redo your edits. You can also press Ctrl-Z to undo and Shift-Ctrl-Z to redo as you would in a text editor.

Filter expressions use standard C relational operators (==, >=, &&, `||`, and so on) along with keywords that are specific to the experiment. To see the keywords that you can use in an experiment that is open in the Analyzer, click the Show Keywords button in the Custom tab.

The filter expression syntax is the same as that used for filtering with `er_print`. See "Expression Grammar" on page 150 for information about filter expressions.

The edited filters do not affect the data tabs until you click OK or Apply. If you click Apply, the Custom tab remains open. If you select context filters in the Analyzer data tabs, you can see the filter expressions generated by your selections as they are added to the Filter Specification text box. Each new filter is placed on a new line beginning with &&, the logical "and" operator. Experiment data must match the first filter *and* the second filter *and* the third filter, and so on, in order to be displayed. You can change && to `||` if you want data to match the first filter *or* the second filter, for example.

You can also use the Custom tab to filter using labels as explained in the following section.

Using Labels for Filtering

Labels are names you can assign to a portion of an experiment. Using the `er_label` command, you can assign a label name to a period of time in an experiment and the label persists with the experiment. You can use the label to filter the experiment data with the `er_print` command or the Performance Analyzer to include or exclude the data collected during the labelled time period.

See "Labeling Experiments" on page 212 for information about how to use the `er_label` utility to create labels.

In the Performance Analyzer you can filter data from the labeled time period in the Custom tab of the Manage Filters dialog box. Type the label name in the Filter Specification panel and click Apply to filter the data specified by the label. You do not need to use any numerical comparison because the label acts as a nickname for a filter expression that uses a numerical comparison with the `TSTAMP` keyword. You can combine the label with other filters in the Custom tab by adding it on a separate line preceded by &&.

You can see if there are labels assigned to an experiment that is open in the Performance Analyzer by clicking the Show Keywords button in the Custom tab. You can also use the `er_print -describe` command to see the same information. Labels are listed first in the display and include the actual filter expression with the `TSTAMP` keyword that is implemented by the label.

After applying a label filter, you can click the Timeline tab to see data is removed in the intervals that were defined by the label. The data is also filtered in other tabs that support filtering.

Recording Experiments from Analyzer

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

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

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

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

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

A set of buttons allows the following operations:

- Running the experiment
- Terminating the run
- Sending Pause, Resume, and Sample signals to the process during the run (enabled if the corresponding signals are specified
- Closing the window

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

Default Settings for Analyzer

The default settings for Analyzer are controlled through .er.rc defaults files. The Analyzer processes directives from several of these files, in the following order:

- er.rc file located in the lib directory of your installed Oracle Solaris Studio software. For example, in the default Oracle Solaris installation, the file is at /opt/solarisstudio12.3/lib/.er.rc.
- .er.rc file in your home directory, if the file exists
- .er.rc file in the current directory, if the file exists

Some settings are accumulated from all .er.rc files, including pathmap and addpath. For other settings, the last .er.rc file read takes precedence. The .er.rc settings in the current directory take precedence over the .er.rc settings in your home directory, which take precedence over the .er.rc settings in the system-wide version of the file.

Saving Performance Analyzer Settings

In the Performance Analyzer, you can create and update an .er.rc file by clicking the Save button in the Set Data Presentation dialog, which you can open from the View menu. Saving the settings from the Set Data Presentation dialog box not only affects subsequent invocations of the Analyzer, but also the er_print utility and er_src utility because the utilities all use the .er.rc file to determine default settings.

Settings in the .er.rc File

The .er.rc files can contain settings for the following:

- Specifying which tabs are visible when you load an experiment into the Analyzer. The Analyzer tab names match the er_print command for the corresponding report, except for the Experiments tab and the Timeline tab.
- Definitions for custom MemoryObjects and IndexObjects.
- Default settings for metrics, for sorting, and for specifying compiler commentary options.
- Thresholds for highlighting metrics in source and disassembly output.
- Default settings for the Timeline tab, and for name formatting, and setting View mode.
- Specifying the search path or pathmaps for source files and object files.
- Showing and hiding functions from load objects.

- Specifying whether to load descendant experiments when the founder experiment is read. The setting for `en_desc` may be `on`, `off`, or =*regexp* to specifying reading and loading all descendants, no descendants, or reading and loading those descendants whose lineage or executable name match the given regular expression, respectively. By default `en_desc` is `on` so that all descendants are loaded.

For details about the commands you can use in the `.er.rc` files, see "Commands That Set Defaults" on page 146 and "Commands That Set Defaults Only For the Performance Analyzer" on page 148.

Comparing Experiments

You can load multiple experiments or experiment groups simultaneously in the Analyzer. By default, when multiple experiments on the same executable are loaded, the data is aggregated and presented as if it were one experiment. You can also look at the data separately, so you can compare the data in the experiments.

To compare two experiments in the Analyzer, you can open the first experiment in the normal way, and then select File > Add Experiment to load the second experiment. To compare them, right-click in a tab that supports comparing and select Enable Experiment Comparison.

The tabs that support comparing experiments are Functions, Callers-Callees, Source, Disassembly, Lines, and PCs. In comparison mode, the data from the experiments or groups is shown in adjacent columns on these tabs. The columns are shown in the order of the loading of the experiments or groups, with an additional header line giving the experiment or group name.

Enabling Comparison Mode By Default

You can enable comparison mode by default by setting `compare on` in your `.er.rc` file. Alternatively, you can enable comparison mode in the Analyzer's Set Data Presentation dialog by selecting the Compare Experiments option in the Formats tab. To make this the default, click the Save button to add `compare on` to your `.er.rc` file.

You can also compare experiments using the `er_print compare` command. See "`compare { on | off }`" on page 145 for more information.

CHAPTER 5

The er_print Command Line Performance Analysis Tool

This chapter explains how to use the er_print utility for performance analysis. The er_print utility prints an ASCII version of the various displays supported by the Performance Analyzer. The information is written to standard output unless you redirect it to a file. You must give the er_print utility the name of one or more experiments or experiment groups generated by the Collector as arguments.

The er_print utility only works on experiments that were recorded with Sun Studio 12 Update 1, Oracle Solaris Studio 12.2, and this release, Oracle Solaris Studio 12.3. An error is reported if you use an experiment recorded with any other version. If you have older experiments, you must use the version of er_print from the release with which the experiment was recorded.

You can use the er_print utility to display the performance metrics for functions, for callers and callees; the source code listing and disassembly listing; sampling information; dataspace data; thread analysis data, and execution statistics.

When invoked on more than one experiment or experiment groups, er_print aggregates the experiment data by default, but can also be used to compare the experiments. See "compare { on | off }" on page 145 for more information.

This chapter covers the following topics.

- "Commands for the OpenMP Index Objects" on page 136
- "Commands That Support the Thread Analyzer" on page 137
- "Commands That List Experiments, Samples, Threads, and LWPs" on page 138
- "Commands That Control Filtering of Experiment Data" on page 139
- "Commands That Control Load Object Expansion and Collapse" on page 141
- "Commands That List Metrics" on page 143
- "Commands That Control Output" on page 143
- "Commands That Print Other Information" on page 145
- "Commands That Set Defaults" on page 146
- "Commands That Set Defaults Only For the Performance Analyzer" on page 148
- "Miscellaneous Commands" on page 150
- "Expression Grammar" on page 150
- "`er_print` command Examples" on page 154

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

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

`er_print` Syntax

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

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

The options for the `er_print` utility are:

`-`	Read `er_print` commands entered from the keyboard.
`-script` *script*	Read commands from the file *script*, which contains a list of `er_print` commands, one per line. If the `-script` option is not present, `er_print` reads commands from the terminal or from the command line.
-*command* [*argument*]	Process the given command.
`-V`	Display version information and exit.

Multiple options can appear on the `er_print` command line. They are processed in the order they appear. You can mix scripts, hyphens, and explicit commands in any order. The default action if you do not supply any commands or scripts is to enter interactive mode, in which commands are entered from the keyboard. To exit interactive mode type `quit` or Ctrl-D.

After each command is processed, any error messages or warning messages arising from the processing are printed. You can print summary statistics on the processing with the `procstats` command.

The commands accepted by the `er_print` utility are listed in the following sections.

You can abbreviate any command with a shorter string as long as the command is unambiguous. You can split a command into multiple lines by terminating a line with a backslash, \. Any line that ends in \ will have the \ character removed, and the content of the next line appended before the line is parsed. There is no limit, other than available memory, on the number of lines you can use for a command

You must enclose arguments that contain embedded blanks in double quotes. You can split the text inside the quotes across lines.

Metric Lists

Many of the `er_print` commands use a list of metric keywords. The syntax of the list is:

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

For dynamic metrics, those based on measured data, a metric keyword consists of three parts: a metric flavor string, a metric visibility string, and a metric name string. These are joined with no spaces, as follows.

flavorvisibilityname

For static metrics, those based on the static properties of the load objects in the experiment (name, address, and size), a metric keyword consists of a metric name, optionally preceded by a metric visibility string, joined with no spaces:

`[`*visibility*`]`*name*

The metric *flavor* and metric *visibility* strings are composed of flavor and visibility characters.

The allowed metric flavor characters are given in Table 5–1. A metric keyword that contains more than one flavor character is expanded into a list of metric keywords. For example, `ie.user` is expanded into `i.user:e.user`.

TABLE 5–1 Metric Flavor Characters

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

The allowed metric visibility characters are given in Table 5–2. The order of the visibility characters in the visibility string does not matter: it does not affect the order in which the corresponding metrics are displayed. For example, both `i%.user` and `i.%user` are interpreted as `i.user:i%user`.

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

TABLE 5–2 Metric Visibility Characters

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

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

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

You can use the visibility character exclamation point (`!`) to override the built-in visibility defaults for each flavor of metric.

If the same metric appears multiple times in the metric list, only the first appearance is processed and subsequent appearances are ignored. If the named metric is not on the list, it is appended to the list.

Table 5–3 lists the available `er_print` metric name strings for timing metrics, synchronization delay metrics, memory allocation metrics, MPI tracing metrics, and the two common hardware counter metrics. For other hardware counter metrics, the metric name string is the same as the counter name. You can get a list of all the available metric name strings for the loaded experiments with the `metric_list` command. A list of counter names can be obtained by using the `collect -h` command with no additional arguments. See "Hardware Counter Overflow Profiling Data" on page 26 for more information on hardware counters.

TABLE 5–3 Metric Name Strings

Category	String	Description
Timing metrics	`user`	User CPU time
	`wall`	Wall-clock time
	`total`	Total thread time
	`system`	System CPU time
	`wait`	CPU wait time
	`ulock`	User lock time
	`text`	Text-page fault time
	`data`	Data-page fault time
	`owait`	Other wait time
Clock-based profiling metrics	`mpiwork`	Time spent inside the MPI runtime doing work, such as processing requests or messages
	`mpiwait`	Time spent inside the MPI runtime, but waiting for an event, buffer, or message
	`ompwork`	Time spent doing work either serially or in parallel
	`ompwait`	Time spent when OpenMP runtime is waiting for synchronization
Synchronization delay metrics	`sync`	Synchronization wait time
	`syncn`	Synchronization wait count
MPI tracing metrics	`mpitime`	Time spent in MPI calls
	`mpisend`	Number of MPI point-to-point sends started
	`mpibytessent`	Number of bytes in MPI Sends
	`mpireceive`	Number of MPI point-to-point receives completed
	`mpibytesrecv`	Number of bytes in MPI Receives
	`mpiother`	Number of calls to other MPI functions
Memory allocation metrics	`alloc`	Number of allocations
	`balloc`	Bytes allocated
	`leak`	Number of leaks

TABLE 5–3 Metric Name Strings *(Continued)*

Category	String	Description
	`bleak`	Bytes leaked
Hardware counter overflow metrics	`cycles`	CPU cycles
	`insts`	Instructions issued
Thread Analyzer metrics	`raccesses`	Data race accesses
	`deadlocks`	Deadlocks

In addition to the name strings listed in Table 5–3, two name strings can only be used in default metrics lists. These are `hwc`, which matches any hardware counter name, and `any`, which matches any metric name string. Also note that `cycles` and `insts` are common to SPARC platforms and x86 platforms, but other flavors also exist that are architecture-specific.

To see the metrics available from the experiments you have loaded, use the `metric_list` command.

Commands That Control the Function List

The following commands control how the function information is displayed.

`functions`

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

You can limit the number of lines written by using the `limit` command (see “Commands That Control Output” on page 143).

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

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

- User mode shows each method by name, with data for interpreted and HotSpot-compiled methods aggregated together; it also suppresses data for non-user-Java threads.

- Expert mode separates HotSpot-compiled methods from interpreted methods, and does not suppress non-user Java threads.
- Machine mode shows data for interpreted Java methods against the Java Virtual Machine (JVM) software as it does the interpreting, while data for methods compiled with the Java HotSpot virtual machine is reported for named methods. All threads are shown.

In all three modes, data is reported in the usual way for any C, C++, or Fortran code called by a Java target.

`metrics` *metric_spec*

Specify a selection of function-list metrics. The string *metric_spec* can either be the keyword `default`, which restores the default metric selection, or a list of metric keywords, separated by colons. The following example illustrates a metric list.

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

This command instructs the `er_print` utility to display the following metrics:

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

By default, the metric setting used is based on the `dmetrics` command, processed from `.er.rc` files, as described in “Commands That Set Defaults” on page 146. If a `metrics` command explicitly sets *metric_spec* to `default`, the default settings are restored as appropriate to the data recorded.

When metrics are reset, the default sort metric is set in the new list.

If *metric_spec* is omitted, the current metrics setting is displayed.

In addition to setting the metrics for the function list, the `metrics` command sets metrics for callers-callees, for data-derived output, and for index objects. The callers-callees metrics show the attributed metrics that correspond to those metrics in the functions list whose inclusive or exclusive metrics are shown, as well as the static metrics.

The dataspace metrics show the dataspace metrics for which data is available and that correspond to those metrics in the function list whose inclusive or exclusive metrics are shown, as well as the static metrics.

The index objects metrics show the index-object metrics corresponding to those metrics in the function list whose inclusive or exclusive metrics are shown, as well as the static metrics.

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

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

For information on the syntax of metric lists, see "Metric Lists" on page 121. To see a listing of the available metrics, use the `metric_list` command.

If a `metrics` command has an error, it is ignored with a warning, and the previous settings remain in effect.

`sort` *metric_spec*

Sort the function list on *metric_spec*. The *visibility* in the metric name does not affect the sort order. If more than one metric is named in the *metric_spec*, use the first one that is visible. If none of the metrics named are visible, ignore the command. You can precede the *metric_spec* with a minus sign (-) to specify a reverse sort.

By default, the metric sort setting is based on the `dsort` command, processed from `.er.rc` files, as described in "Commands That Set Defaults" on page 146. If a `sort` command explicitly sets *metric_spec* to `default`, the default settings are used.

The string *metric_spec* is one of the metric keywords described in "Metric Lists" on page 121, as shown in this example.

```
sort i.user
```

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

`fsummary`

Write a summary panel for each function in the function list. You can limit the number of panels written by using the `limit` command (see "Commands That Control Output" on page 143).

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

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

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

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

Commands That Control the Callers-Callees List

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

`callers-callees`

Print the callers-callees panel for each of the functions, in the order specified by the function sort metric (`sort`).

Within each caller-callee report, the callers and callees are sorted by the caller-callee sort metrics (`csort`). You can limit the number of panels written by using the `limit` command (see "Commands That Control Output" on page 143). The selected (center) function is marked with an asterisk, as shown in this example.

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

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

`csingle` *function_name* `[`*N*`]`

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

`cprepend` *function-name [N | ADDR]*

When building a call stack, prepend the named function to the current call stack fragment. The optional parameter is needed where the function name is ambiguous; see “`source|src` { *filename* | *function_name* } [*N*]” on page 130 for more information about specifying the parameter.

`cappend` *function-name [N | ADDR]*

When building a call stack, append the named function to the current call stack fragment. The optional parameter is needed where the function name is ambiguous; see “`source|src` { *filename* | *function_name* } [*N*]” on page 130 for more information about specifying the parameter.

`crmfirst`

When building a call stack, remove the top frame from the call stack segment.

`crmlast`

When building a call stack, remove the bottom frame from the call stack segment.

Commands That Control the Call Tree List

This section describes the commands for the call tree.

`calltree`

Display the dynamic call graph from the experiment, showing the hierarchical metrics at each level.

Commands That Control the Leak and Allocation Lists

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

`leaks`

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

`allocs`

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

Commands That Control the Source and Disassembly Listings

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

`pcs`

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

`psummary`

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

`lines`

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

`lsummary`

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

`source|src` { *filename* | *function_name* } [*N*]

Write out annotated source code for either the specified file or the file containing the specified function. The file in either case must be in a directory in your path. If the source was compiled with the GNU Fortran compiler, you must add two underscore characters after the function name as it appears in the source.

Use the optional parameter *N* (a positive integer) only in those cases where the file or function name is ambiguous; in this case, the *N*th possible choice is used. If you give an ambiguous name without the numeric specifier the `er_print` utility prints a list of possible object-file names; if the name you gave was a function, the name of the function is appended to the object-file name, and the number that represents the value of *N* for that object file is also printed.

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

```
<Function: f_name>
```

The default source context for any function is defined as the source file to which the first instruction in that function is attributed. It is normally the source file compiled to produce the object module containing the function. Alternate source contexts consist of other files that contain instructions attributed to the function. Such contexts include instructions coming from include files and instructions from functions inlined into the named function. If there are any alternate source contexts, include a list of extended index lines at the beginning of the default source context to indicate where the alternate source contexts are located in the following form:

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

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

Normally when the default source context is used, metrics are shown for all functions from that file. Referring to the file explicitly shows metrics only for the named function.

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

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

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

scc *com_spec*

Specify the classes of compiler commentary that are shown in the annotated source listing. The class list is a colon-separated list of classes, containing zero or more of the following message classes.

TABLE 5–4 Compiler Commentary Message Classes

Class	Meaning
`b[asic]`	Show the basic level messages.
`v[ersion]`	Show version messages, including source file name and last modified date, versions of the compiler components, compilation date and options.
`pa[rallel]`	Show messages about parallelization.
`q[uery]`	Show questions about the code that affect its optimization.
`l[oop]`	Show messages about loop optimizations and transformations.
`pi[pe]`	Show messages about pipelining of loops.
`i[nline]`	Show messages about inlining of functions.
`m[emops]`	Show messages about memory operations, such as load, store, prefetch.
`f[e]`	Show front-end messages.
`co[degen]`	Show code generator messages.
`cf`	Show compiler flags at the bottom of the source.
`all`	Show all messages.
`none`	Do not show any messages.

The classes `all` and `none` cannot be used with other classes.

If no `scc` command is given, the default class shown is `basic`. If the `scc` command is given with an empty *class-list*, compiler commentary is turned off. The `scc` command is normally used only in an `.er.rc` file.

`sthresh` *value*

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

`dcc` *com_spec*

Specify the classes of compiler commentary that are shown in the annotated disassembly listing. The class list is a colon-separated list of classes. The list of available classes is the same as the list of classes for annotated source code listing shown in Table 5–4. You can add the following options to the class list.

TABLE 5–5 Additional Options for the `dcc` Command

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

`dthresh` *value*

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

`cc` *com_spec*

Specify the classes of compiler commentary that are shown in the annotated source and disassembly listing. The class list is a colon-separated list of classes. The list of available classes is the same as the list of classes for annotated source code listing shown in Table 5–4.

Commands That Control Searching For Source Files

The `er_print` utility looks for the source files and load object files referenced in an experiment. You can use the directives described in this section to help `er_print` find the files referenced by your experiment.

See "How the Tools Find Source Code" on page 191 for a description of the process used to find an experiment's source code, including how these directives are used.

`setpath` *path_list*

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

The default path is: `$expts:..` which is the directories of the loaded experiments and the current working directory.

Use `setpath` with no argument to display the current path.

`addpath` *path_list*

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

`pathmap` *old-prefix new-prefix*

If a file cannot be found using the *path_list* set by `addpath` or `setpath`, you can specify one or more path remappings with the `pathmap` command. In any pathname for a source file, object file, or shared object that begins with the prefix specified with *old-prefix*, the old prefix is replaced by the prefix specified with *new-prefix*. The resulting path is then used to find the file. Multiple `pathmap` commands can be supplied, and each is tried until the file is found.

Commands That Control Hardware Counter Dataspace and Memory Object Lists

For experiments collected with hardware counter profiling and dataspace profiling, you can view metrics related to the following:

- Data objects - program constants, variables, arrays and aggregates such as structures and unions, along with distinct aggregate elements, described in source code.
- Memory objects - components in the memory subsystem, such as cache lines, pages, and memory banks. The object is determined from an index computed from the virtual or physical address as recorded. Memory objects are predefined for virtual and physical pages, for sizes of 8KB, 64KB, 512KB, and 4MB. You can define others with the `mobj_define` command.

This data can only be collected on Solaris objects compiled with the `-xhwcprof` compiler option on SPARC architectures.

See "Hardware Counter Overflow Profiling Data" on page 26 and for more information about these types of data. See "`-h` *counter_definition_1*`...[,`*counter_definition_n*`]`" on page 60 for information about the command line used to perform hardware counter overflow profiling.

For information about the `-xhwcprof` compiler option, see the *Oracle Solaris Studio 12.3: Fortran User's Guide*, the *Oracle Solaris Studio 12.3: C User's Guide*, or the *Oracle Solaris Studio 12.3: C++ User's Guide*.

`data_objects`

Write the list of data objects with their metrics.

`data_single` *name* `[`*N*`]`

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

`data_layout`

Write the annotated data object layouts for all program data objects with data-derived metric data, sorted by the current data sort metric values for the structures as a whole. Each aggregate data object is shown with the total metrics attributed to it, followed by all of its elements in offset order, each with their own metrics and an indicator of its size and location relative to 32-byte blocks.

memobj *mobj_type*

Write the list of the memory objects of the given type with the current metrics. Metrics used and sorting as for the data space list. You can also use the name *mobj_type* directly as the command.

mobj_list

Write the list of known types of memory objects, as used for *mobj_type* in the `memobj` command.

mobj_define *mobj_type index_exp*

Define a new type of memory objects with a mapping of VA/PA to the object given by the *index_exp*. The syntax of the expression is described in "Expression Grammar" on page 150.

The *mobj_type* must not already be defined. Its name must be entirely composed of alphanumeric characters or the '_' character, and begin with an alphabetic character.

The *index_exp* must be syntactically correct. If it is not syntactically correct, an error is returned and the definition is ignored.

The `<Unknown>` memory object has an index of `-1`, and the expression used to define a new memory object should support recognizing `<Unknown>`. For example, for `VADDR`-based objects, the expression should be of the following form:

```
VADDR>255?expression :-1
```

and for `PADDR`-based objects, the expression should be of the following form:

```
PADDR>0?expression:-1
```

Commands That Control Index Object Lists

Index objects commands are applicable to all experiments. An index object list is a list of objects for whom an index can be computed from the recorded data. Index objects are predefined for Threads, Cpus, Samples, and Seconds. You can define other index objects with the `indxobj_define` command.

The following commands control the index-object lists.

`indxobj` *indxobj_type*

Write the list of the index objects that match the given type, along with their metrics. Metrics and sorting for index objects is the same as those for the function list, but containing exclusive metrics only. The name *indxobj_type* can also be used directly as the command.

`indxobj_list`

Write the list of known types of index objects, as used for *indxobj_type* in the `indxobj` command.

`indxobj_define` *indxobj_type index_exp*

Define a new type of index object with a mapping of packets to the object given by the *index_exp*. The syntax of the expression is described in "Expression Grammar" on page 150.

The *indxobj_type* must not already be defined. Its name is case-insensitive, must be entirely composed of alphanumeric characters or the '_' character, and begin with an alphabetic character.

The *index_exp* must be syntactically correct, or an error is returned and the definition is ignored. If the *index_exp* contains any blanks, it must be surrounded by double quotes (").

The `<Unknown>` index object has an index of `-1`, and the expression used to define a new index object should support recognizing `<Unknown>`.

For example, for index objects based on virtual or physical PC, the expression should be of the following form:

```
VIRTPC>0?VIRTPC:-1
```

Commands for the OpenMP Index Objects

The following commands let you print information for OpenMP index objects.

`OMP_preg`

Print a list of the OpenMP parallel regions executed in the experiment with their metrics. This command is available only for experiments with OpenMP 3.0 performance data.

`OMP_task`

Print a list of the OpenMP tasks executed in the experiment with their metrics. This command is available only for experiments with OpenMP 3.0 performance data.

Commands That Support the Thread Analyzer

The following commands are in support of the Thread Analyzer. See the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information about the data captured and shown.

`races`

Writes a list of all data races in the experiments. Data-race reports are available only from experiments with data-race-detection data.

`rdetail` *race_id*

Writes the detailed information for the given *race_id*. If the *race_id* is set to `all`, detailed information for all data races is shown. Data race reports are available only from experiments with data race detection data.

`deadlocks`

Writes a list of all detected real and potential deadlocks in the experiments. Deadlock reports are available only from experiments with deadlock-detection data.

`ddetail` *deadlock_id*

Writes the detailed information for the given *deadlock_id*. If the *deadlock_id* is set to `all`, detailed information for all deadlocks is shown. Deadlock reports are available only from experiments with deadlock-detection data.

Commands That List Experiments, Samples, Threads, and LWPs

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

experiment_list

Display the full list of experiments loaded with their ID number. Each experiment is listed with an index, which is used when selecting samples, threads, or LWPs, and a PID, which can be used for advanced filtering.

The following example shows an experiment list.

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

sample_list

Display the list of samples currently selected for analysis.

The following example shows a sample list.

```
(er_print) sample_list
Exp Sel     Total
=== ======= =====
  1 1-6        31
  2 7-10,15    31
```

lwp_list

Display the list of LWPs currently selected for analysis.

thread_list

Display the list of threads currently selected for analysis.

cpu_list

Display the list of CPUs currently selected for analysis.

Commands That Control Filtering of Experiment Data

You can specify filtering of experiment data in two ways:

- By specifying a filter expression, which is evaluated for each data record to determine whether or not the record should be included
- By selecting experiments, samples, threads, CPUs, and LWPs for filtering

Specifying a Filter Expression

You can specify a filter expression with the `filters` command.

`filters` *filter_exp*

filter_exp is an expression that evaluates as true for any data record that should be included, and false for records that should not be included. The grammar of the expression is described in "Expression Grammar" on page 150.

Listing Keywords for a Filter Expression

You can see a list of operands or keywords that you can use in a filter expression on your experiment.

`describe`

Print the list of keywords that can be used to build a filter expression. Some keywords and the grammar of a filter expression is described in "Expression Grammar" on page 150.

In the Performance Analyzer, you can see the same information by selecting View ⇒ Manage Filters and clicking the Show Keywords button in the Custom tab.

Selecting Samples, Threads, LWPs, and CPUs for Filtering

Selection Lists

The syntax of a selection is shown in the following example. This syntax is used in the command descriptions.

```
[experiment-list:]selection-list[+[
experiment-list:]selection-list ... ]
```

Each selection list can be preceded by an experiment list, separated from it by a colon and no spaces. You can make multiple selections by joining selection lists with a + sign.

The experiment list and the selection list have the same syntax, which is either the keyword `all` or a list of numbers or ranges of numbers (*n-m*) separated by commas but no spaces, as shown in this example.

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

The experiment numbers can be determined by using the `experiment_list` command.

Some examples of selections are as follows.

```
1:1-4+2:5,6
all:1,3-6
```

In the first example, objects 1 through 4 are selected from experiment 1 and objects 5 and 6 are selected from experiment 2. In the second example, objects 1 and 3 through 6 are selected from all experiments. The objects may be LWPs, threads, or samples.

Selection Commands

The commands to select LWPs, samples, CPUs, and threads are not independent. If the experiment list for a command is different from that for the previous command, the experiment list from the latest command is applied to all three selection targets, LWPs, samples, and threads, in the following way.

- Existing selections for experiments not in the latest experiment list are turned off.
- Existing selections for experiments in the latest experiment list are kept.
- Selections are set to `all` for targets for which no selection has been made.

`sample_select` *sample_spec*

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

`lwp_select` *lwp_spec*

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

`thread_select` *thread_spec*

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

cpu_select *cpu_spec*

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

Commands That Control Load Object Expansion and Collapse

These commands determine how load objects are displayed by the er_print utility.

object_list

Display a two-column list showing the status and names of all load objects. The show/hide/api status of each load object is shown in the first column, and the name of the object is shown in the second column. The name of each load object is preceded either by a show that indicates that the functions of that object are shown in the function list (expanded), by a hide that indicates that the functions of that object are not shown in the function list (collapsed), or by API-only if only those functions representing the entry point into the load object are shown. All functions for a collapsed load object map to a single entry in the function list representing the entire load object.

The following is an example of a load object list.

```
(er_print) object_list
Sel  Load Object
==== ==================
hide <Unknown>
show <Freeway>
show <libCstd_isa.so.1>
show <libnsl.so.1>
show <libmp.so.2>
show <libc.so.1>
show <libICE.so.6>
show <libSM.so.6>
show <libm.so.1>
show <libCstd.so.1>
show <libX11.so.4>
show <libXext.so.0>
show <libCrun.so.1>
show <libXt.so.4>
show <libXm.so.4>
show <libsocket.so.1>
show <libgen.so.1>
show <libcollector.so>
show <libc_psr.so.1>
show <ld.so.1>
show <liblayout.so.1>
```

`object_show` *object1,object2,...*

Set all named load objects to show all their functions. The names of the objects can be either full path names or the basename. If the name contains a comma character, the name must be surrounded by double quotation marks. If the string "all" is used to name the load object, functions are shown for all load objects.

`object_hide` *object1,object2,...*

Set all named load objects to hide all their functions. The names of the objects can be either full path names or the basename. If the name contains a comma character, the name must be surrounded by double quotation marks. If the string "all" is used to name the load object, functions are shown for all load objects.

`object_api` *object1,object2,...*

Set all named load objects to show all only the functions representing entry points into the library. The names of the objects can be either full path names or the basename. If the name contains a comma character, the name must be surrounded by double quotation marks. If the string "all" is used to name the load object, functions are shown for all load objects.

`objects_default`

Set all load objects according to the initial defaults from `.er.rc` file processing.

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

Select the load objects for which you want to display information about the functions in the load object. Functions from all named load objects are shown; functions from all others are hidden. *object-list* is a list of load objects, separated by commas but no spaces. If functions from a load object are shown, all functions that have non-zero metrics are shown in the function list. If a functions from a load object are hidden, its functions are collapsed, and only a single line with metrics for the entire load object instead of its individual functions is displayed.

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

Commands That List Metrics

The following commands list the currently selected metrics and all available metric keywords.

`metric_list`

Display the currently selected metrics in the function list and a list of metric keywords that you can use in other commands (for example, `metrics` and `sort`) to reference various types of metrics in the function list.

`cmetric_list`

Display the currently selected caller-callee attributed metrics and the metric currently used for sorting.

`data_metric_list`

Display the currently selected data-derived metrics and a list of metrics and keyword names for all data-derived reports. Display the list in the same way as the output for the `metric_list` command, but include only those metrics that have a data-derived flavor and static metrics.

`indx_metric_list`

Display the currently selected index-object metrics and a list of metrics and keyword names for all index object reports. Display the list in the same way as the `metric_list` command, but include only those metrics that have an exclusive flavor, and static metrics.

Commands That Control Output

The following commands control `er_print` display output.

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

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

`appendfile` *filename*

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

`limit` *n*

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

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

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

`viewmode { user| expert | machine }`

Set the mode to one of the following:

`user` For Java experiments, show the Java call stacks for Java threads, and do not show housekeeping threads. The function list includes a function `<JVM-System>` representing aggregated time from non-Java threads. When the JVM software does not report a Java call stack, time is reported against the function `<no Java callstack recorded>`.

For OpenMP experiments, show reconstructed call stacks similar to those obtained when the program is compiled without OpenMP. Add special functions, with the names of form `<OMP-*>`, when the OpenMP runtime is performing certain operations.

`expert` For Java experiments, show the Java call stacks for Java threads when the user's Java code is being executed, and machine call stacks when JVM code is being executed or when the JVM software does not report a Java call stack. Show the machine call stacks for housekeeping threads.

For OpenMP experiments, show compiler generated functions representing parallelized loops, tasks, and such, which are aggregated with user functions in user mode. Add special functions, with the names of form `<OMP-*>`, when the OpenMP runtime is performing certain operations. Functions from the OpenMP runtime code `libmtsk.so` are suppressed.

`machine` For Java experiments and OpenMP experiments, show the actual native call stacks for all threads.

For all experiments other than Java experiments and OpenMP experiments, all three modes show the same data.

`compare { on | off }`

Set comparison mode on or off. The default is off so when multiple experiments on the same executable are read, the data is aggregated. If comparison mode is enabled by setting `compare on` in your `.er.rc` file, and multiple experiments on the same executable are loaded, separate columns of metrics are shown for the data from each experiment. You can also compare experiments using the `er_print compare` command.

In comparison mode, the data from the experiments or groups is shown in adjacent columns on the Functions list, the Callers-callees list, and the Source and Disassembly lists. The columns are shown in the order of the loading of the experiments or groups, with an additional header line giving the experiment or group name.

Commands That Print Other Information

The following `er_print` subcommands display miscellaneous information about the experiment.

`header` *exp_id*

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

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

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

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

`ifreq`

Write a list of instruction frequency from the measured count data. The instruction frequency report can only be generated from count data. This command applies only on Oracle Solaris.

`objects`

List the load objects with any error or warning messages that result from the use of the load object for performance analysis. The number of load objects listed can be limited by using the `limit` command (see "Commands That Control Output" on page 143).

`overview` *exp_id*

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

`statistics` *exp_id*

Write out execution statistics, aggregated over the current sample set for the specified experiment. For information on the definitions and meanings of the execution statistics that are presented, see the `getrusage`(3C) and `proc`(4) man pages. The execution statistics include statistics from system threads for which the Collector does not collect any data.

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

Commands That Set Defaults

You can use the following commands in your `.er.rc`file to set the defaults for `er_print`, `er_src`, and the Performance Analyzer. You can use these commands only for setting defaults: they cannot be used as input for the `er_print` utility. The commands can only be included in a defaults file named `.er.rc`.Commands that apply only to defaults for the Performance Analyzer are described in "Commands That Set Defaults Only For the Performance Analyzer" on page 148. See "Default Settings for Analyzer" on page 117 for information about how the Analyzer uses the `.er.rc` file.

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

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

The defaults file can also include the `scc`, `sthresh` , `dcc`, `dthresh`, `cc`, `setpath`, `addpath`, `pathmap`, `name`, `mobj_define`, `object_show`, `object_hide`, `object_api`, `indxobj_define`, `tabs`, `rtabs`, and `viewmode` commands. You can include multiple `dmetrics`, `dsort`, `addpath`, `pathmap`, `mobj_define`, and `indxobj_define` commands in a defaults file, and the commands from all `.er.rc` files are concatenated. For all other commands, the first appearance of the command is used and subsequent appearances are ignored.

`dmetrics` *metric_spec*

Specify the default metrics to be displayed or printed in the function list. The syntax and use of the metric list is described in the section "Metric Lists" on page 121. The order of the metric keywords in the list determines the order in which the metrics are presented and the order in which they appear in the Metric chooser in the Performance Analyzer.

Default metrics for the Callers-Callees list are derived from the function list default metrics by adding the corresponding attributed metric before the first occurrence of each metric name in the list.

`dsort` *metric_spec*

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

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

The syntax and use of the metric list is described in the section "Metric Lists" on page 121.

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

`en_desc { on | off | =`*regexp*`}`

Set the mode for reading descendant experiments to `on` (enable all descendants) or `off` (disable all descendants). If the =*regexp* is used, enable data from those experiments whose lineage or executable name matches the regular expression. The default setting is `on` to follow all descendants.

Commands That Set Defaults Only For the Performance Analyzer

You can use the following commands in your `.er.rc`file to set some additional defaults for the Performance Analyzer.

`tabs` *tab_spec*

Set the default set of tabs to be visible in the Analyzer. The tabs are named by the `er_print` command that generates the corresponding reports. In addition, `mpi_timeline` specifies the MPI Timeline tab, `mpi_chart` specifies the MPI Chart tab, `timeline` specifies the Timeline tab, and `headers` specifies the Experiments tab.

Only those tabs that are supported by the data in the loaded experiments are shown.

`rtabs` *tab_spec*

Set the default set of tabs to be visible when the Analyzer is invoked with the `tha` command, for examining Thread Analyzer experiments. Only those tabs that are supported by the data in the loaded experiments are shown.

`tlmode` *tl_mode*

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

TABLE 5–6 Timeline Display Mode Options

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

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

`tldata` *tl_data*

Select the default data types shown in the Timeline tab of the Performance Analyzer. The value of *tl_data* is a list of types separated by colons. The allowed types are listed in the following table.

TABLE 5–7 Timeline Display Data Types

Type	Meaning
`sa[mple]`	Display sample data
`c[lock]`	Display clock profiling data
`hw[c]`	Display hardware counter profiling data
`sy[nctrace]`	Display thread synchronization tracing data
`mp[itrace]`	Display MPI tracing data
`he[aptrace]`	Display heap tracing data

Miscellaneous Commands

The following commands perform miscellaneous tasks in the `er_print` utility.

`procstats`

Print the accumulated statistics from processing data.

`script` *file*

Process additional commands from the script file *file*.

`version`

Print the current release number of the `er_print` utility

`quit`

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

`help`

Print a list of `er_print` commands.

Expression Grammar

A common grammar is used for an expression defining a filter and an expression used to compute a memory object index.

The grammar specifies an expression as a combination of operators and operands or keywords. For filters, if the expression evaluates to true, the packet is included; if the expression evaluates to false, the packet is excluded. For memory objects or index objects, the expression is evaluated to an index that defines the particular memory object or index object referenced in the packet.

Operands in an expression can be labels, constants, or fields within a data record, as listed with the `describe` command. The operands include `THRID`, `LWPID`, `CPUID` , `USTACK`, `XSTACK`, `MSTACK`, `LEAF`, `VIRTPC`, `PHYSPC`, `VADDR`, `PADDR`, `DOBJ`, `TSTAMP`, `SAMPLE`, `EXPID`, `PID`, or the name of a memory object. Operand names are case-insensitive.

USTACK, XSTACK, and MSTACK represent the function call stacks in user view, expert view, and machine view, respectively.

VIRTPC, PHYSPC, VADDR, and PADDR are non-zero only when “+” is specified for Hardware-counter-profiling or clock-profiling. Furthermore, VADDR is less than 256 when the real virtual address could not be determined. PADDR is zero if VADDR could not be determined, or if the virtual address could not be mapped to a physical address. Likewise, VIRTPC is zero if backtracking failed, or was not requested, and PHYSPC is zero if either VIRTPC is zero, or the VIRTPC could not be mapped to a physical address.

Operators include the usual logical operators and arithmetic (including shift) operators, in C notation, with C precedence rules, and an operator for determining whether an element is in a set (IN) or whether any or all of a set of elements is contained in a set (SOME IN or IN, respectively). An additional operator ORDERED IN is for determining if all elements from the left operand appear in the same sequence in the right operand. Note that the IN operator requires all elements from the left operand to appear in the right operand but does not enforce the order. If-then-else constructs are specified as in C, with the ? and : operators. Use parentheses to ensure proper parsing of all expressions. On the er_print command lines, the expression cannot be split across lines. In scripts or on the command line, the expression must be inside double quotes if it contains blanks.

Filter expressions evaluate to a boolean value, true if the packet should be included, and false if it should not be included. Thread, CPU, experiment-id, process-pid, and sample filtering are based on a relational expression between the appropriate keyword and an integer, or using the IN operator and a comma-separated list of integers.

Time-filtering is used by specifying one or more relational expressions between TSTAMP and a time, given in integer nanoseconds from the start of the experiment whose packets are being processed. Times for samples can be obtained using the overview command. Times in the overview command are given in seconds, and must be converted to nanoseconds for time-filtering. Times can also be obtained from the Timeline display in the Analyzer.

Function filtering can be based either on the leaf function, or on any function in the stack. Filtering by leaf function is specified by a relational expression between the LEAF keyword and an integer function id, or using the IN operator and the construct FNAME("*regexp*"), where *regexp* is a regular expression as specified on the regexp(5) man page. The entire name of the function, as given by the current setting of *name*, must match.

Filtering based on any function in the call stack is specified by determining if any function in the construct FNAME("*regexp*") is in the array of functions represented by the keyword USTACK using the expression (FNAME("myfunc") SOME IN USTACK). FNAME can also be used to filter the machine view of the stack (MSTACK) and the expert view (XSTACK) in the same way.

Data object filtering is analogous to stack function filtering, using the DOBJ keyword and the construct DNAME("*regexp*") enclosed in parentheses.

Memory object filtering is specified using the name of the memory object, as shown in the `mobj_list` command, and the integer index of the object, or the indices of a set of objects. (The `<Unknown>` memory object has index -1.)

Index object filtering is specified using the name of the index object, as shown in the `indxobj_list` command, and the integer index of the object, or the indices of a set of objects. (The `<Unknown>` index object has index -1.)

Data object filtering and memory object filtering are meaningful only for hardware counter packets with dataspace data; all other packets are excluded under such filtering.

Direct filtering of virtual addresses or physical addresses is specified with a relational expression between `VADDR` or `PADDR`, and the address.

Memory object definitions (see "`mobj_define` *mobj_type index_exp*" on page 135) use an expression that evaluates to an integer index, using either the `VADDR` keyword or `PADDR` keyword. The definitions are applicable only to hardware counter packets for memory counters and dataspace data. The expression should return an integer, or -1 for the `<Unknown>` memory object.

Index object definitions (see "`indxobj_define` *indxobj_type index_exp*" on page 136) use an expression that evaluates to an integer index. The expression should return an integer, or -1 for the `<Unknown>` index object.

Example Filter Expressions

This section shows examples of filter expressions that can be used with the `er_print -filters` command, and in the Custom tab of the Performance Analyzer's Manage Filter dialog box.

With the `er_print -filters` command, the filter expression is enclosed in single quotes, similar to the following:

```
er_print -filters ’FNAME("myfunc") SOME IN USTACK’ -functions test.1.er
```

EXAMPLE 5–1 Filter functions by name and stack

To filter functions named `myfunc` from the user function stack:

```
FNAME("myfunc") SOME IN USTACK
```

EXAMPLE 5–2 Filter events by thread and CPU

To see events from thread 1 when it was running on CPU 2 only, use:

```
THRID == 1 && CPUID == 2
```

EXAMPLE 5–3 Filter events by index object

If an index object, THRCPU, is defined as "CPUID<<16|THRID", the following filter is equivalent to the one above to see events from thread 1 when it was running on CPU 2:

```
THRCPU == 0x10002
```

EXAMPLE 5–4 Filter events occurring in specified time period

To filter events from experiment 2 that occurred during the period between second 5 and second 9:

```
EXPID==2 && TSTAMP >= 5000000000 && TSTAMP < 9000000000
```

EXAMPLE 5–5 Filter events from a particular Java class

To filter events that have any method from a particular Java class in the stack (in user view mode):

```
FNAME("myClass.*") SOME IN USTACK
```

EXAMPLE 5–6 Filter events by internal function ID and call sequence

If function IDs are known (as shown in the Analyzer), to filter events that contain a particular call sequence in the machine call stack:

```
(314,272) ORDERED IN MSTACK
```

EXAMPLE 5–7 Filter events by state or duration

If the describe command lists the following properties for a clock profiling experiment:

```
MSTATE    UINT32  Thread state
NTICK     UINT32  Duration
```

you can select events that are in a particular state using the following filter:

```
MSTATE == 1
```

Alternatively, you can use the following filter to select events that are in a particular state and whose duration is longer than 1 clock tick:

```
MSTATE == 1 && NTICK > 1
```

er_print command Examples

This section provides some examples for using the er_print command.

EXAMPLE 5–8 Show summary of how time is spent in functions

```
er_print -functions test.1.er
```

EXAMPLE 5–9 Show caller-callee relationships

```
er_print -callers-callees test.1.er
```

EXAMPLE 5–10 Show which source lines are hot

Source-line information assumes the code was compiled and linked with -g. Append a trailing underscore to the function name for Fortran functions and routines. The 1 after the function name is used to distinguish between multiple instances of myfunction.

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

EXAMPLE 5–11 Filter functions named myfunc from the user function stack:

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

EXAMPLE 5–12 Generate output similar to gprof

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

```
er_print -outfile  er_print.out -metrics e.%user -sort e.user \
-limit 100 -func -callers-callees test.1.er
```

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

```
er_print -metrics  e.%user -limit 100  -functions test.1.er

er_print -metrics  e.%user -callers-callees test.1.er
```

EXAMPLE 5–13 Show only the compiler commentary

It is not necessary to run your program in order to use this command.

```
er_src -myfile.o
```

EXAMPLE 5–14 Use wall-clock profiling to list functions and callers-callees

```
er_print -metrics  ei.%wall -functions test.1.er

er_print -metrics aei.%wall  -callers-callees test.1.er
```

EXAMPLE 5–15 Run a script containing er_print commands

```
er_print -script myscriptfile test.1.er
```

The myscriptfile script contains er_print commands. A sample of the script file contents follows:

```
## myscriptfile

## Send script output to standard output
outfile -

## Display descriptive information about the experiments
header

## Write out the sample data for all experiments
overview

## Write out execution statistics, aggregated over
## the current sample set for all experiments
statistics

## List functions
functions

## Display status and names of available load objects
object_list


## Write out annotated disassembly code for systime,
## to file disasm.out
outfile disasm.out
disasm systime


## Write out annotated source code for synprog.c
## to file source.out
outfile source.out
source synprog.c

## Terminate processing of the script
quit
```

CHAPTER 6

Understanding the Performance Analyzer and Its Data

The Performance Analyzer reads the event data that is collected by the Collector and converts it into performance metrics. The metrics are computed for various elements in the structure of the target program, such as instructions, source lines, functions, and load objects. In addition to a header containing a timestamp, thread id, LWP id, and CPU id, the data recorded for each event collected has two parts:

- Some event-specific data that is used to compute metrics
- A call stack of the application that is used to associate those metrics with the program structure

The process of associating the metrics with the program structure is not always straightforward, due to the insertions, transformations, and optimizations made by the compiler. This chapter describes the process and discusses the effect on what you see in the Performance Analyzer displays.

This chapter covers the following topics:

How Data Collection Works

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

Experiment Format

All experiments must have three files:

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

In addition, experiments have binary data files representing the profile events in the life of the process. Each data file has a series of events, as described below under "Interpreting Performance Metrics" on page 161. Separate files are used for each type of data, but each file is shared by all threads in the target.

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

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

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

The `archives` Directory

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

Subexperiments

Subexperiments are created when multiple processes are profiled, such as when you follow descendent processes, collect an MPI experiment, or profile the kernel with user processes.

Descendant processes write their experiments into subdirectories within the founder-process' experiment directory. These new subexperiments are named to indicate their lineage as follows:

- An underscore is appended to the creator's experiment name.
- One of the following code letters is added: `f` for fork, `x` for exec, and `c` for other descendants.
- A number to indicate the index of the fork or exec is added after the code letter. This number is applied whether the process was started successfully or not.
- The experiment suffix, `.er` is appended to complete the experiment name.

For example, if the experiment name for the founder process is `test.1.er`, the experiment for the child process created by its third fork is `test.1.er/_f3.er`. If that child process executes a new image, the corresponding experiment name is `test.1.er/_f3_x1.er`. Descendant experiments consist of the same files as the parent experiment, but they do not have descendant experiments (all descendants are represented by subdirectories in the founder experiment), and they do not have archive subdirectories (all archiving is done into the founder experiment).

Data for MPI programs are collected by default into `test.1.er`, and all the data from the MPI processes are collected into subexperiments, one per rank. The Collector uses the MPI rank to construct a subexperiment name with the form `M_r`*m*`.er`, where *m* is the MPI rank. For example, MPI rank 1 would have its experiment data recorded in the `test.1.er/M_r1.er` directory.

Experiments on the kernel by default are named `ktest.1.er` rather than `test.1.er`. When data is also collected on user processes, the kernel experiment contains subexperiments for each user process being followed. The kernel subexperiments are named using the format *_process-name*`_PID_`*process-id*`.1.er`. For example an experiment run on a `sshd` process running under process ID 1264 would be named `ktest.1.er/_sshd_PID_1264.1.er`.

Dynamic Functions

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

Java Experiments

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

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

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

Recording Experiments

You can record an experiment on a user-mode target in three different ways:

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

The Performance Collect window in the Analyzer GUI runs a collect experiment.

collect Experiments

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

libcollector.so and associated modules are responsible for writing all experiment files.

dbx Experiments That Create a Process

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

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

dbx Experiments on a Running Process

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

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

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

Interpreting Performance Metrics

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

In addition to the common data, each event generates specific raw data, which is described in the following sections. Each section also contains a discussion of the accuracy of the metrics derived from the raw data and the effect of data collection on the metrics.

Clock-Based Profiling

The event-specific data for clock-based profiling consists of an array of profiling interval counts. On Oracle Solaris, an interval counter is provided. At the end of the profiling interval, the appropriate interval counter is incremented by 1, and another profiling signal is scheduled. The array is recorded and reset only when the Solaris thread enters CPU user mode. Resetting the array consists of setting the array element for the User-CPU state to 1, and the array elements for all the other states to 0. The array data is recorded on entry to user mode before the array is reset. Thus, the array contains an accumulation of counts for each microstate that was entered since the previous entry into user mode, for each of the ten microstates maintained by the kernel for each Solaris thread. On the Linux operating system, microstates do not exist; the only interval counter is User CPU Time.

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

The metrics to which each of the microstates contributes on Oracle Solaris are shown in Table 6–1.

TABLE 6–1 How Kernel Microstates Contribute to Metrics

Kernel Microstate	Description	Metric Name
`LMS_USER`	Running in user mode	User CPU Time
`LMS_SYSTEM`	Running in system call or page fault	System CPU Time
`LMS_TRAP`	Running in any other trap	System CPU Time
`LMS_TFAULT`	Asleep in user text page fault	Text Page Fault Time
`LMS_DFAULT`	Asleep in user data page fault	Data Page Fault Time
`LMS_KFAULT`	Asleep in kernel page fault	Other Wait Time
`LMS_USER_LOCK`	Asleep waiting for user-mode lock	User Lock Time
`LMS_SLEEP`	Asleep for any other reason	Other Wait Time
`LMS_STOPPED`	Stopped (`/proc`, job control, or `lwp_stop`)	Other Wait Time
`LMS_WAIT_CPU`	Waiting for CPU	Wait CPU Time

Accuracy of Timing Metrics

Timing data is collected on a statistical basis, and is therefore subject to all the errors of any statistical sampling method. For very short runs, in which only a small number of profile packets is recorded, the call stacks might not represent the parts of the program which consume the most resources. Run your program for long enough or enough times to accumulate hundreds of profile packets for any function or source line you are interested in.

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

- When a thread is created, the time spent before the first profile packet is recorded is less than the profiling interval, but the entire profiling interval is ascribed to the microstate recorded in the first profile packet. If many threads are created, the error can be many times the profiling interval.
- When a thread is destroyed, some time is spent after the last profile packet is recorded. If many threads are destroyed, the error can be many times the profiling interval.

- Rescheduling of threads can occur during a profiling interval. As a consequence, the recorded state of the thread might not represent the microstate in which it spent most of the profiling interval. The errors are likely to be larger when there are more threads to run than there are processors to run them.
- A program can behave in a way that is correlated with the system clock. In this case, the profiling interval always expires when the thread is in a state that might represent a small fraction of the time spent, and the call stacks recorded for a particular part of the program are overrepresented. On a multiprocessor system, the profiling signal can induce a correlation: processors that are interrupted by the profiling signal while they are running threads for the program are likely to be in the Trap-CPU microstate when the microstate is recorded.
- The kernel records the microstate value when the profiling interval expires. When the system is under heavy load, that value might not represent the true state of the process. On Oracle Solaris, this situation is likely to result in overaccounting of the Trap-CPU or Wait-CPU microstate.
- When the system clock is being synchronized with an external source, the time stamps recorded in profile packets do not reflect the profiling interval but include any adjustment that was made to the clock. The clock adjustment can make it appear that profile packets are lost. The time period involved is usually several seconds, and the adjustments are made in increments.
- Experiments recorded on machines that dynamically change their operating clock frequency may induce inaccuracies in profiling.

In addition to the inaccuracies just described, timing metrics are distorted by the process of collecting data. The time spent recording profile packets never appears in the metrics for the program, because the recording is initiated by the profiling signal. (This is another instance of correlation.) The user CPU time spent in the recording process is distributed over whatever microstates are recorded. The result is an underaccounting of the User CPU Time metric and an overaccounting of other metrics. The amount of time spent recording data is typically less than a few percent of the CPU time for the default profiling interval.

Comparisons of Timing Metrics

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

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

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

The microstate timings that appear in the Statistics tab of the Performance Analyzer and the er_print statistics display are based on process file system /proc usage reports, for which the times spent in the microstates are recorded to high accuracy. See the proc (4) man page for more information. You can compare these timings with the metrics for the <Total> function, which represents the program as a whole, to gain an indication of the accuracy of the aggregated timing metrics. However, the values displayed in the Statistics tab can include other contributions that are not included in the timing metric values for <Total>. These contributions come from the periods of time in which data collection is paused.

User CPU time and hardware counter cycle time differ because the hardware counters are turned off when the CPU mode has been switched to system mode. For more information, see "Traps" on page 169.

Hardware Counter Overflow Profiling

Hardware counter overflow profiling data includes a counter ID and the overflow value. The value can be larger than the value at which the counter is set to overflow, because the processor executes some instructions between the overflow and the recording of the event. The value is especially likely to be larger for cycle and instruction counters, which are incremented much more frequently than counters such as floating-point operations or cache misses. The delay in recording the event also means that the program counter address recorded with call stack does not correspond exactly to the overflow event. See "Attribution of Hardware Counter Overflows" on page 202 for more information. See also the discussion of "Traps" on page 169. Traps and trap handlers can cause significant differences between reported User CPU time and time reported by the cycle counter.

Experiments recorded on machines that dynamically change their operating clock frequency will show inaccuracies in the conversion of cycle-based count to time.

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

- The amount of time spent collecting data can be a substantial fraction of the execution time of the program. The collection run might spend most of its time handling overflows and writing data instead of running the program.
- A substantial fraction of the counts can come from the collection process. These counts are attributed to the collector function collector_record_counters . If you see high counts for this function, the overflow value is too small.

- The collection of data can alter the behavior of the program. For example, if you are collecting data on cache misses, the majority of the misses could come from flushing the collector instructions and profiling data from the cache and replacing it with the program instructions and data. The program would appear to have a lot of cache misses, but without data collection there might in fact be very few cache misses.

Dataspace Profiling and Memoryspace Profiling

Dataspace profiling is an extension to hardware counter profiling that is used for memory references. Hardware counter profiling can attribute metrics to user functions, source lines, and instructions, but not to data objects that are being referenced. By default, the Collector only captures the user instruction addresses. When dataspace profiling is enabled, the Collector also captures data addresses. *Backtracking* is the technique used on some machines to get the performance information that supports dataspace profiling. When backtracking is enabled, the Collector looks back at the load or store instructions that occurred before a hardware counter event to find a candidate instruction that could cause the event. On some systems, counters are precise and no backtracking is needed. Such counters are indicated by the word `precise` in the output of the `collect -h` command.

A dataspace profile is a data collection in which memory-related events, such as cache misses, are reported against the data object references that cause the events rather than just the instructions where the memory-related events occur.

A memoryspace profile is similar to a dataspace profile except that in a memoryspace profile the events are reported against components of the memory subsystem such as cache-lines or pages, rather than data objects in the program. Memoryspace profiling occurs when you prepend a + sign to a precise counter that is related to memory.

To allow dataspace profiling, the target must be a C, C++, or Fortran program, compiled for the SPARC architecture, with the `-xhwcprof` flag and `-xdebugformat=dwarf -g` flags. Furthermore, the data collected must be hardware counter profiles for memory-related counters and the + sign must be prepended to the counter name. If the optional + is prepended to one memory-related counter, but not all, the counters without the + report dataspace data against the `<Unknown>` data object, with subtype `Dataspace data not requested during data collection`.

On machines with precise interrupts no backtracking is required, and memoryspace profiling does not require the `-xhwcprof` flag and `-xdebugformat=dwarf -g` flags for compilation. Dataspace profiling, even on such machines, does require the flags.

When an experiment includes a dataspace or memoryspace profile, the `er_print` utility allows three additional commands: `data_objects`, `data_single`, and `data_layout`, as well as various commands relating to memory objects. See "Commands That Control Hardware Counter Dataspace and Memory Object Lists" on page 134 for more information.

In addition, the Performance Analyzer includes two tabs related to dataspace profiling, the DataObjects tab and the DataLayout tab, and various tabs for memory objects. See "The DataObjects Tab" on page 99 and "The DataLayout Tab" on page 100 and "The MemoryObjects Tabs" on page 104.

Running `collect -h` with no additional arguments lists hardware counters, and specifies whether they are load, store, or load-store related and whether they are precise. See "Hardware Counter Overflow Profiling Data" on page 26.

Synchronization Wait Tracing

The Collector collects synchronization delay events by tracing calls to the functions in the threads library, `libthread.so`, or to the real time extensions library, `librt.so`. The event-specific data consists of high-resolution timestamps for the request and the grant (beginning and end of the call that is traced), and the address of the synchronization object (the mutex lock being requested, for example). The thread and LWP IDs are the IDs at the time the data is recorded. The wait time is the difference between the request time and the grant time. Only events for which the wait time exceeds the specified threshold are recorded. The synchronization wait tracing data is recorded in the experiment at the time of the grant.

The waiting thread cannot perform any other work until the event that caused the delay is completed. The time spent waiting appears both as Synchronization Wait Time and as User Lock Time. User Lock Time can be larger than Synchronization Wait Time because the synchronization delay threshold screens out delays of short duration.

The wait time is distorted by the overhead for data collection. The overhead is proportional to the number of events collected. You can minimize the fraction of the wait time spent in overhead by increasing the threshold for recording events.

Heap Tracing

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

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

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

MPI Tracing

MPI tracing is based on a modified VampirTrace data collector. For more information, see the VampirTrace User Manual on the Technische Universität Dresden web site.

Call Stacks and Program Execution

A *call stack* is a series of program counter addresses (PCs) representing instructions from within the program. The first PC, called the *leaf PC*, is at the bottom of the stack, and is the address of the next instruction to be executed. The next PC is the address of the call to the function containing the leaf PC; the next PC is the address of the call to that function, and so forth, until the top of the stack is reached. Each such address is known as a return address. The process of recording a call stack involves obtaining the return addresses from the program stack and is referred to as *unwinding the stack*. For information on unwind failures, see "Incomplete Stack Unwinds" on page 179.

The leaf PC in a call stack is used to assign exclusive metrics from the performance data to the function in which that PC is located. Each PC on the stack, including the leaf PC, is used to assign inclusive metrics to the function in which it is located.

Most of the time, the PCs in the recorded call stack correspond in a natural way to functions as they appear in the source code of the program, and the Performance Analyzer's reported metrics correspond directly to those functions. Sometimes, however, the actual execution of the program does not correspond to a simple intuitive model of how the program would execute, and the Performance Analyzer's reported metrics might be confusing. See "Mapping Addresses to Program Structure" on page 180 for more information about such cases.

Single-Threaded Execution and Function Calls

The simplest case of program execution is that of a single-threaded program calling functions within its own load object.

When a program is loaded into memory to begin execution, a context is established for it that includes the initial address to be executed, an initial register set, and a stack (a region of memory used for scratch data and for keeping track of how functions call each other). The initial address is always at the beginning of the function `_start()`, which is built into every executable.

When the program runs, instructions are executed in sequence until a branch instruction is encountered, which among other things could represent a function call or a conditional statement. At the branch point, control is transferred to the address given by the target of the branch, and execution proceeds from there. (Usually the next instruction after the branch is already committed for execution: this instruction is called the branch delay slot instruction. However, some branch instructions annul the execution of the branch delay slot instruction).

When the instruction sequence that represents a call is executed, the return address is put into a register, and execution proceeds at the first instruction of the function being called.

In most cases, somewhere in the first few instructions of the called function, a new frame (a region of memory used to store information about the function) is pushed onto the stack, and the return address is put into that frame. The register used for the return address can then be used when the called function itself calls another function. When the function is about to return, it pops its frame from the stack, and control returns to the address from which the function was called.

Function Calls Between Shared Objects

When a function in one shared object calls a function in another shared object, the execution is more complicated than in a simple call to a function within the program. Each shared object contains a Program Linkage Table, or PLT, which contains entries for every function external to that shared object that is referenced from it. Initially the address for each external function in the PLT is actually an address within `ld.so`, the dynamic linker. The first time such a function is called, control is transferred to the dynamic linker, which resolves the call to the real external function and patches the PLT address for subsequent calls.

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

Signals

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

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

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

Traps

Traps can be issued by an instruction or by the hardware, and are caught by a trap handler. System traps are traps that are initiated from an instruction and trap into the kernel. All system calls are implemented using trap instructions. Some examples of hardware traps are those issued from the floating point unit when it is unable to complete an instruction, or when the instruction is not implemented in the hardware.

When a trap is issued, the kernel enters system mode. On Oracle Solaris, the microstate is usually switched from User CPU state to Trap state then to System state. The time spent handling the trap can show as a combination of System CPU time and User CPU time, depending on the point at which the microstate is switched. The time is attributed to the instruction in the user's code from which the trap was initiated (or to the system call).

For some system calls, it is considered critical to provide as efficient handling of the call as possible. The traps generated by these calls are known as *fast traps*. Among the system functions that generate fast traps are `gethrtime` and `gethrvtime`. In these functions, the microstate is not switched because of the overhead involved.

In other circumstances it is also considered critical to provide as efficient handling of the trap as possible. Some examples of these are TLB (translation lookaside buffer) misses and register window spills and fills, for which the microstate is not switched.

In both cases, the time spent is recorded as User CPU time. However, the hardware counters are turned off because the CPU mode has been switched to system mode. The time spent handling these traps can therefore be estimated by taking the difference between User CPU time and Cycles time, preferably recorded in the same experiment.

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

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

Tail-Call Optimization

The compiler can do one particular optimization whenever the last thing a particular function does is to call another function. Rather than generating a new frame, the callee reuses the frame from the caller, and the return address for the callee is copied from the caller. The motivation for this optimization is to reduce the size of the stack, and, on SPARC platforms, to reduce the use of register windows.

Suppose that the call sequence in your program source looks like this:

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

When B and C are tail-call optimized, the call stack looks as if function A calls functions B, C, and D directly.

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

That is, the call tree is flattened. When code is compiled with the -g option, tail-call optimization takes place only at a compiler optimization level of 4 or higher. When code is compiled without the- g option, tail-call optimization takes place at a compiler optimization level of 2 or higher.

Explicit Multithreading

A simple program executes in a single thread. Multithreaded executables make calls to a thread creation function, to which the target function for execution is passed. When the target exits, the thread is destroyed.

Oracle Solaris supports two thread implementations: Solaris threads and POSIX threads (Pthreads). Beginning with Oracle Solaris 10, both thread implementations are included in libc.so.

With Solaris threads, newly-created threads begin execution at a function called _thread_start(), which calls the function passed in the thread creation call. For any call stack involving the target as executed by this thread, the top of the stack is _thread_start(), and there is no connection to the caller of the thread creation function. Inclusive metrics associated with the created thread therefore only propagate up as far as _thread_start() and the <Total> function. In addition to creating the threads, the Solaris threads implementation also creates LWPs on Solaris to execute the threads. Each thread is bound to a specific LWP.

Pthreads is available in Oracle Solaris as well as in Linux for explicit multithreading.

In both environments, to create a new thread, the application calls the Pthread API function pthread_create(), passing a pointer to an application-defined start routine as one of the function arguments.

On Solaris versions before Oracle Solaris 10 , when a new pthread starts execution, it calls the _lwp_start() function. Beginning with Oracle Solaris 10, _lwp_start() calls an intermediate function _thrp_setup(), which then calls the application-defined start routine that was specified in pthread_create().

On the Linux operating system, when the new pthread starts execution, it runs a Linux-specific system function, clone(), which calls another internal initialization function, pthread_start_thread(), which in turn calls the application-defined start routine that was specified in pthread_create() . The Linux metrics-gathering functions available to the Collector are thread-specific. Therefore, when the collect utility runs, it interposes a metrics-gathering function, named `collector_root()`, between `pthread_start_thread()` and the application-defined thread start routine.

Overview of Java Technology-Based Software Execution

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

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

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

During execution of a Java technology-based application, most methods are interpreted by the JVM software; these methods are referred to as *interpreted methods*. The Java HotSpot virtual machine monitors performance as it interprets the bytecode to detect methods that are frequently executed. Methods that are repeatedly executed might then be compiled by the Java HotSpot virtual machine to generate machine code for those methods. The resulting methods are referred to as *compiled methods*. The virtual machine executes the more efficient compiled methods thereafter, rather than interpret the original bytecode for the methods. Compiled methods are loaded into the data space of the application, and may be unloaded at some later point in time. In addition, other code is generated in the data space to execute the transitions between interpreted and compiled code.

Code written in the Java programming language might also call directly into native-compiled code, either C, C++, or Fortran; the targets of such calls are referred to as native methods.

Applications written in the Java programming language are inherently multithreaded, and have one JVM software thread for each thread in the user's program. Java applications also have several housekeeping threads used for signal handling, memory management, and Java HotSpot virtual machine compilation.

Data collection is implemented with various methods in the JVMTI in J2SE.

Java Call Stacks and Machine Call Stacks

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

Clock-based Profiling and Hardware Counter Overflow Profiling

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

Java Profiling View Modes

The Performance Analyzer provides three view modes for displaying performance data for applications written in the Java programming language: the User mode, the Expert mode, and the Machine mode. The User mode is shown by default where the data supports it. The following section summarizes the main differences between these three view modes.

User View Mode of Java Profiling Data

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

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

The function list in the User mode shows metrics against the Java methods and any native methods called. The Callers-Callees tab shows the calling relationships in the User mode.

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

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

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

Expert View Mode of Java Profiling Data

The Expert mode is similar to the User mode, except that some details of the JVM internals that are suppressed in the User mode are exposed in the Expert mode. With the Expert mode, the Timeline shows all threads; the call stack for housekeeping threads is a native call stack.

Machine View Mode of Java Profiling Data

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

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

The Timeline in the Machine view mode shows bars for all threads, LWPs, or CPUs, and the call stack in each is the Machine mode of the call stack.

Overview of OpenMP Software Execution

The actual execution model of OpenMP applications is described in the OpenMP specifications (See, for example, OpenMP Application Program Interface, Version 3.0, section 1.3.) The specification, however, does not describe some implementation details that may be important to users, and the actual implementation from Oracle is such that directly recorded profiling information does not easily allow the user to understand how the threads interact.

As any single-threaded program runs, its call stack shows its current location, and a trace of how it got there, starting from the beginning instructions in a routine called `_start`, which calls `main`, which then proceeds and calls various subroutines within the program. When a subroutine contains a loop, the program executes the code inside the loop repeatedly until the loop exit criterion is reached. The execution then proceeds to the next sequence of code, and so forth.

When the program is parallelized with OpenMP (or by autoparallelization), the behavior is different. An intuitive model of the parallelized program has the main, or master, thread executing just as a single-threaded program. When it reaches a parallel loop or parallel region, additional slave threads appear, each a clone of the master thread, with all of them executing the contents of the loop or parallel region, in parallel, each for different chunks of work. When all chunks of work are completed, all the threads are synchronized, the slave threads disappear, and the master thread proceeds.

The actual behavior of the parallelized program is not so straightforward. When the compiler generates code for a parallel region or loop (or any other OpenMP construct), the code inside it is extracted and made into an independent function, called an *mfunction* in the Oracle implementation. (It may also be referred to as an outlined function, or a loop-body-function.) The name of the mfunction encodes the OpenMP construct type, the name of the function from which it was extracted, and the line number of the source line at which the construct appears. The names of these functions are shown in the Analyzer's Expert mode and Machine mode in the following form, where the name in brackets is the actual symbol-table name of the function:

```
bardo_ -- OMP parallel region from line 9 [_$p1C9.bardo_]
atomsum_ -- MP doall from line 7 [_$d1A7.atomsum_]
```

There are other forms of such functions, derived from other source constructs, for which the `OMP parallel region` in the name is replaced by `MP construct`, `MP doall`, or `OMP sections`. In the following discussion, all of these are referred to generically as *parallel regions*.

Each thread executing the code within the parallel loop can invoke its mfunction multiple times, with each invocation doing a chunk of the work within the loop. When all the chunks of work are complete, each thread calls synchronization or reduction routines in the library; the master thread then continues, while the slave threads become idle, waiting for the master thread to enter the next parallel region. All of the scheduling and synchronization are handled by calls to the OpenMP runtime.

During its execution, the code within the parallel region might be doing a chunk of the work, or it might be synchronizing with other threads or picking up additional chunks of work to do. It might also call other functions, which may in turn call still others. A slave thread (or the master thread) executing within a parallel region, might itself, or from a function it calls, act as a master thread, and enter its own parallel region, giving rise to nested parallelism.

The Analyzer collects data based on statistical sampling of call stacks, and aggregates its data across all threads and shows metrics of performance based on the type of data collected, against functions, callers and callees, source lines, and instructions. The Analyzer presents information on the performance of OpenMP programs in one of three view modes: User mode , Expert mode, and Machine mode.

For more detailed information about data collection for OpenMP programs, see An OpenMP Runtime API for Profiling (`http://www.compunity.org/futures/omp-api.html`) at the OpenMP user community web site.

User View Mode of OpenMP Profile Data

The User mode presentation of the profile data attempts to present the information as if the program really executed according to the intuitive model described in "Overview of OpenMP Software Execution" on page 173. The actual data, shown in the Machine mode, captures the implementation details of the runtime library, `libmtsk.so` , which does not correspond to the model. The Expert mode shows a mix of data altered to fit the model, and the actual data.

In User mode, the presentation of profile data is altered to match the model better, and differs from the recorded data and Machine mode presentation in three ways:

- Artificial functions are constructed representing the state of each thread from the point of view of the OpenMP runtime library.
- Call stacks are manipulated to report data corresponding to the model of how the code runs, as described above.
- Two additional metrics of performance are constructed for clock-based profiling experiments, corresponding to time spent doing useful work and time spent waiting in the OpenMP runtime. The metrics are OpenMP Work and OpenMP Wait.
- For OpenMP 3.0 programs, a third metric OpenMP Overhead is constructed.

Artificial Functions

Artificial functions are constructed and put onto the User mode and Expert mode call stacks reflecting events in which a thread was in some state within the OpenMP runtime library.

The following artificial functions are defined:

`<OMP-overhead>`	Executing in the OpenMP library
`<OMP-idle>`	Slave thread, waiting for work
`<OMP-reduction>`	Thread performing a reduction operation
`<OMP-implicit_barrier>`	Thread waiting at an implicit barrier
`<OMP-explicit_barrier>`	Thread waiting at an explicit barrier
`<OMP-lock_wait>`	Thread waiting for a lock
`<OMP-critical_section_wait>`	Thread waiting to enter a critical section
`<OMP-ordered_section_wait>`	Thread waiting for its turn to enter an ordered section
`<OMP-atomic_section_wait>`	Thread waiting on an OpenMP atomic construct.

When a thread is in an OpenMP runtime state corresponding to one of the artificial functions, the artificial function is added as the leaf function on the stack. When a thread's actual leaf function is anywhere in the OpenMP runtime, it is replaced by `<OMP-overhead>` as the leaf function. Otherwise, all PCs from the OpenMP runtime are omitted from the user-mode stack.

For OpenMP 3.0 programs, the `<OMP-overhead>` artificial function is not used. The artificial function is replaced by an OpenMP Overhead metric.

User Mode Call Stacks

For OpenMP experiments, User mode shows reconstructed call stacks similar to those obtained when the program is compiled without OpenMP. The goal is to present profile data in a manner that matches the intuitive understanding of the program rather than showing all the details of the actual processing. The call stacks of the master thread and slave threads are reconciled and the artificial `<OMP-*>` functions are added to the call stack when the OpenMP runtime library is performing certain operations.

OpenMP Metrics

When processing a clock-profile event for an OpenMP program, two metrics corresponding to the time spent in each of two states in the OpenMP system are shown: OpenMP Work and OpenMP Wait.

Time is accumulated in OpenMP Work whenever a thread is executing from the user code, whether in serial or parallel. Time is accumulated in OpenMP Wait whenever a thread is waiting for something before it can proceed, whether the wait is a busy-wait (spin-wait), or sleeping. The sum of these two metrics matches the Total Thread metric in the clock profiles.

The OpenMP Wait and OpenMP Work metrics are shown in User mode, Expert mode, and Machine mode.

Expert View Mode of OpenMP Profiling Data

When you look at OpenMP experiments in Expert view mode you see the artificial functions of the form `<OMP-*>` when the OpenMP runtime is performing certain operations, similar to User view mode. However, Expert view mode separately shows compiler-generated mfunctions that represent parallelized loops, tasks, and so on. In User mode, these compiler-generated mfunctions are aggregated with user functions.

Machine View Mode of OpenMP Profiling Data

Machine mode shows native call stacks for all threads and outline functions generated by the compiler.

The real call stacks of the program during various phases of execution are quite different from the ones portrayed above in the intuitive model. The Machine mode shows the call stacks as measured, with no transformations done, and no artificial functions constructed. The clock-profiling metrics are, however, still shown.

In each of the call stacks below, `libmtsk` represents one or more frames in the call stack within the OpenMP runtime library. The details of which functions appear and in which order change from release to release of OpenMP, as does the internal implementation of code for a barrier, or to perform a reduction.

1. Before the first parallel region

   Before the first parallel region is entered, there is only the one thread, the master thread. The call stack is identical to that in User mode and Expert mode.

Master
`foo`
`main`
`_start`

2. During execution in a parallel region

Master	Slave 1	Slave 2	Slave 3
`foo-OMP...`			
`libmtsk`			
`foo`	`foo-OMP...`	`foo-OMP...`	`foo-OMP...`
`main`	`libmtsk`	`libmtsk`	`libmtsk`
`_start`	`_lwp_start`	`_lwp_start`	`_lwp_start`

   In Machine mode, the slave threads are shown as starting in `_lwp_start` , rather than in `_start` where the master starts. (In some versions of the thread library, that function may appear as `_thread_start` .) The calls to `foo-OMP...` represent the mfunctions that are generated for parallelized regions.

3. At the point at which all threads are at a barrier

Master	Slave 1	Slave 2	Slave 3
`libmtsk`			
`foo-OMP...`			
`foo`	`libmtsk`	`libmtsk`	`libmtsk`
`main`	`foo-OMP...`	`foo-OMP...`	`foo-OMP...`

Master	Slave 1	Slave 2	Slave 3
_start	_lwp_start	_lwp_start	_lwp_start

Unlike when the threads are executing in the parallel region, when the threads are waiting at a barrier there are no frames from the OpenMP runtime between `foo` and the parallel region code, `foo-OMP...`. The reason is that the real execution does not include the OMP parallel region function, but the OpenMP runtime manipulates registers so that the stack unwind shows a call from the last-executed parallel region function to the runtime barrier code. Without it, there would be no way to determine which parallel region is related to the barrier call in Machine mode.

4. After leaving the parallel region

Master	Slave 1	Slave 2	Slave 3
foo			
main	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start

In the slave threads, no user frames are on the call stack.

5. When in a nested parallel region

Master	Slave 1	Slave 2	Slave 3	Slave 4
	bar-OMP...			
foo-OMP...	libmtsk			
libmtsk	bar			
foo	foo-OMP...	foo-OMP...	foo-OMP...	bar-OMP...
main	libmtsk	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start	_lwp_start

Incomplete Stack Unwinds

Stack unwind is defined in "Call Stacks and Program Execution" on page 167.

Stack unwind might fail for a number of reasons:

- If the stack has been corrupted by the user code; if so, the program might core dump, or the data collection code might core dump, depending on exactly how the stack was corrupted.
- If the user code does not follow the standard ABI conventions for function calls. In particular, on the SPARC platform, if the return register, %o7, is altered before a save instruction is executed.

  On any platform, hand-written assembler code might violate the conventions.
- If the leaf PC is in a function after the callee's frame is popped from the stack, but before the function returns.
- If the call stack contains more than about 250 frames, the Collector does not have the space to completely unwind the call stack. In this case, PCs for functions from `_start` to some point in the call stack are not recorded in the experiment. The artificial function `<Truncated-stack>` is shown as called from `<Total>` to tally the topmost frames recorded.
- If the Collector fails to unwind the frames of optimized functions on x86 platforms.

Intermediate Files

If you generate intermediate files using the `-E` or `-P` compiler options, the Analyzer uses the intermediate file for annotated source code, not the original source file. The `#line` directives generated with `-E` can cause problems in the assignment of metrics to source lines.

The following line appears in annotated source if there are instructions from a function that do not have line numbers referring to the source file that was compiled to generate the function:

function_name `-- <instructions without line numbers>`

Line numbers can be absent under the following circumstances:

- You compiled without specifying the `-g` option.
- The debugging information was stripped after compilation, or the executables or object files that contain the information are moved or deleted or subsequently modified.
- The function contains code that was generated from `#include` files rather than from the original source file.
- At high optimization, if code was inlined from a function in a different file.
- The source file has `#line` directives referring to some other file; compiling with the `-E` option, and then compiling the resulting `.i` file is one way in which this happens. It may also happen when you compile with the `-P` flag.
- The object file cannot be found to read line number information.

- The compiler used generates incomplete line number tables.

Mapping Addresses to Program Structure

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

The Process Image

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

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

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

Load Objects and Functions

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

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

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

Aliased Functions

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

Under various circumstances, a function can be known by several different names. A very common example of this is the use of so-called weak and strong symbols for the same piece of code. A strong name is usually the same as the corresponding weak name, except that it has a leading underscore. Many of the functions in the threads library also have alternate names for pthreads and Solaris threads, as well as strong and weak names and alternate internal symbols. In all such cases, only one name is used in the function list of the Analyzer. The name chosen is the last symbol at the given address in alphabetic order. This choice most often corresponds to the name that the user would use. In the Summary tab, all the aliases for the selected function are shown.

Non-Unique Function Names

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

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

Static Functions From Stripped Shared Libraries

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

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

Fortran Alternate Entry Points

Fortran provides a way of having multiple entry points to a single piece of code, allowing a caller to call into the middle of a function. When such code is compiled, it consists of a prologue for the main entry point, a prologue to the alternate entry point, and the main body of code for the function. Each prologue sets up the stack for the function's eventual return and then branches or falls through to the main body of code.

The prologue code for each entry point always corresponds to a region of text that has the name of that entry point, but the code for the main body of the subroutine receives only one of the possible entry point names. The name received varies from one compiler to another.

The prologues rarely account for any significant amount of time, and the functions corresponding to entry points other than the one that is associated with the main body of the subroutine rarely appear in the Analyzer. Call stacks representing time in Fortran subroutines with alternate entry points usually have PCs in the main body of the subroutine, rather than the prologue, and only the name associated with the main body appears as a callee. Likewise, all calls from the subroutine are shown as being made from the name associated with the main body of the subroutine.

Cloned Functions

The compilers have the ability to recognize calls to a function for which extra optimization can be performed. An example of such calls is a call to a function for which some of the arguments are constants. When the compiler identifies particular calls that it can optimize, it creates a copy of the function, which is called a clone, and generates optimized code. The clone function name is a mangled name that identifies the particular call. The Analyzer demangles the name, and presents each instance of a cloned function separately in the function list. Each cloned function has a different set of instructions, so the annotated disassembly listing shows the cloned functions separately. Each cloned function has the same source code, so the annotated source listing sums the data over all copies of the function.

Inlined Functions

An inlined function is a function for which the instructions generated by the compiler are inserted at the call site of the function instead of an actual call. There are two kinds of inlining, both of which are done to improve performance, and both of which affect the Analyzer.

- C++ inline function definitions. The rationale for inlining in this case is that the cost of calling a function is much greater than the work done by the inlined function, so it is better to simply insert the code for the function at the call site, instead of setting up a call. Typically, access functions are defined to be inlined, because they often only require one instruction. When you compile with the `-g` option, inlining of functions is disabled; compilation with `-g0` permits inlining of functions, and is recommended.
- Explicit or automatic inlining performed by the compiler at high optimization levels (4 and 5). Explicit and automatic inlining is performed even when `-g` is turned on. The rationale for this type of inlining can be to save the cost of a function call, but more often it is to provide more instructions for which register usage and instruction scheduling can be optimized.

Both kinds of inlining have the same effect on the display of metrics. Functions that appear in the source code but have been inlined do not show up in the function list, nor do they appear as callees of the functions into which they have been inlined. Metrics that would otherwise appear as inclusive metrics at the call site of the inlined function, representing time spent in the called function, are actually shown as exclusive metrics attributed to the call site, representing the instructions of the inlined function.

Note – Inlining can make data difficult to interpret, so you might want to disable inlining when you compile your program for performance analysis.

In some cases, even when a function is inlined, a so-called out-of-line function is left. Some call sites call the out-of-line function, but others have the instructions inlined. In such cases, the function appears in the function list but the metrics attributed to it represent only the out-of-line calls.

Compiler-Generated Body Functions

When a compiler parallelizes a loop in a function, or a region that has parallelization directives, it creates new body functions that are not in the original source code. These functions are described in "Overview of OpenMP Software Execution" on page 173.

In user mode, the Analyzer does not show these functions. In expert and machine mode, the Analyzer shows these functions as normal functions, and assigns a name to them based on the function from which they were extracted, in addition to the compiler-generated name. Their exclusive metrics and inclusive metrics represent the time spent in the body function. In addition, the function from which the construct was extracted shows inclusive metrics from each of the body functions. The means by which this is achieved is described in "Overview of OpenMP Software Execution" on page 173.

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

Note – The names of compiler-generated body functions can only be demangled for modules compiled with `-g`.

Outline Functions

Outline functions can be created during feedback-optimized compilations. They represent code that is not normally executed, specifically code that is not executed during the training run used to generate the feedback for the final optimized compilation. A typical example is code that performs error checking on the return value from library functions; the error-handling code is never normally run. To improve paging and instruction-cache behavior, such code is moved elsewhere in the address space, and is made into a separate function. The name of the outline function encodes information about the section of outlined code, including the name of the function from which the code was extracted and the line number of the beginning of the section in the source code. These mangled names can vary from release to release. The Analyzer provides a readable version of the function name.

Outline functions are not really called, but rather are jumped to; similarly they do not return, they jump back. In order to make the behavior more closely match the user's source code model, the Analyzer imputes an artificial call from the main function to its outline portion.

Outline functions are shown as normal functions, with the appropriate inclusive and exclusive metrics. In addition, the metrics for the outline function are added as inclusive metrics in the function from which the code was outlined.

For further details on feedback-optimized compilations, refer to the description of the `-xprofile` compiler option in one of the following manuals:

- Appendix B, "C Compiler Options Reference," in *Oracle Solaris Studio 12.3: C User's Guide*

- Appendix A, "C++ Compiler Options," in *Oracle Solaris Studio 12.3: C++ User's Guide*
- Chapter 3, "Fortran Compiler Options," in *Oracle Solaris Studio 12.3: Fortran User's Guide*

Dynamically Compiled Functions

Dynamically compiled functions are functions that are compiled and linked while the program is executing. The Collector has no information about dynamically compiled functions that are written in C or C++, unless the user supplies the required information using the Collector API functions. See "Dynamic Functions and Modules" on page 50 for information about the API functions. If information is not supplied, the function appears in the performance analysis tools as `<Unknown>`.

For Java programs, the Collector obtains information on methods that are compiled by the Java HotSpot virtual machine, and there is no need to use the API functions to provide the information. For other methods, the performance tools show information for the JVM software that executes the methods. In the Java representation, all methods are merged with the interpreted version. In the machine representation, each HotSpot-compiled version is shown separately, and JVM functions are shown for each interpreted method.

The `<Unknown>` Function

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

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

- When a function written in C or C++ is dynamically generated, and information about the function is not provided to the Collector using the Collector API functions. See "Dynamic Functions and Modules" on page 50 for more information about the Collector API functions.
- When a Java method is dynamically compiled but Java profiling is disabled.
- When the PC corresponds to an address in the data section of the executable or a shared object. One case is the SPARC V7 version of `libc.so`, which has several functions (`.mul` and `.div`, for example) in its data section. The code is in the data section so that it can be dynamically rewritten to use machine instructions when the library detects that it is executing on a SPARC V8 or SPARC V9 platform.
- When the PC corresponds to a shared object in the address space of the executable that is not recorded in the experiment.
- When the PC is not within any known load object. The most likely cause is an unwind failure, where the value recorded as a PC is not a PC at all, but rather some other word. If the PC is the return register, and it does not seem to be within any known load object, it is ignored, rather than attributed to the `<Unknown>` function.

- When a PC maps to an internal part of the JVM software for which the Collector has no symbolic information.

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

OpenMP Special Functions

Artificial functions are constructed and put onto the User mode call stacks reflecting events in which a thread was in some state within the OpenMP runtime library. The following artificial functions are defined:

<OMP-overhead>	Executing in the OpenMP library
<OMP-idle>	Slave thread, waiting for work
<OMP-reduction>	Thread performing a reduction operation
<OMP-implicit_barrier>	Thread waiting at an implicit barrier
<OMP-explicit_barrier>	Thread waiting at an explicit barrier
<OMP-lock_wait>	Thread waiting for a lock
<OMP-critical_section_wait>	Thread waiting to enter a critical section
<OMP-ordered_section_wait>	Thread waiting for its turn to enter an ordered section

The <JVM-System> Function

In the User representation, the <JVM-System> function represents time used by the JVM software performing actions other than running a Java program. In this time interval, the JVM software is performing tasks such as garbage collection and HotSpot compilation. By default, <JVM-System> is visible in the Function list.

The <no Java callstack recorded> Function

The <no Java callstack recorded> function is similar to the <Unknown> function, but for Java threads, in the Java representation only. When the Collector receives an event from a Java thread, it unwinds the native stack and calls into the JVM software to obtain the corresponding Java stack. If that call fails for any reason, the event is shown in the Analyzer with the artificial function <no Java callstack recorded>. The JVM software might refuse to report a call stack either to avoid deadlock, or when unwinding the Java stack would cause excessive synchronization.

The `<Truncated-stack>` Function

The size of the buffer used by the Analyzer for recording the metrics of individual functions in the call stack is limited. If the size of the call stack becomes so large that the buffer becomes full, any further increase in size of the call stack will force the analyzer to drop function profile information. Since in most programs the bulk of exclusive CPU time is spent in the leaf functions, the Analyzer drops the metrics for functions the less critical functions at the bottom of the stack, starting with the entry functions `_start()` and `main()`. The metrics for the dropped functions are consolidated into the single artificial `<Truncated-stack>` function. The `<Truncated-stack>` function may also appear in Java programs.

The `<Total>` Function

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

Functions Related to Hardware Counter Overflow Profiling

The following functions are related to hardware counter overflow profiling:

- `collector_not_program_related`: The counter does not relate to the program.
- `collector_hwcs_out_of_range`: The counter appears to have exceeded the overflow value without generating an overflow signal. The value is recorded and the counter reset.
- `collector_hwcs_frozen`: The counter appears to have exceeded the overflow value and been halted but the overflow signal appears to be lost. The value is recorded and the counter reset.
- `collector_hwc_ABORT`: Reading the hardware counters has failed, typically when a privileged process has taken control of the counters, resulting in the termination of hardware counter collection.
- `collector_record_counter`: The counts accumulated while handling and recording hardware counter events, partially accounting for hardware counter overflow profiling overhead. If this corresponds to a significant fraction of the `<Total>` count, a larger overflow interval (that is, a lower resolution configuration) is recommended.

Mapping Performance Data to Index Objects

Index objects represent sets of things whose index can be computed from the data recorded in each packet. Index-object sets that are predefined include: Threads, Cpus, Samples, and Seconds. Other index objects may be defined either through the `er_print` `indxobj_define` command, issued directly or in a `.er.rc` file. In the Analyzer, you can define index objects by selecting Set Data Presentation from the View menu, selecting the Tabs tab, and clicking the Add Custom Index Object button.

For each packet, the index is computed and the metrics associated with the packet are added to the Index Object at that index. An index of `-1` maps to the `<Unknown>` Index Object. All metrics for index objects are exclusive metrics, as no hierarchical representation of index objects is meaningful.

Mapping Data Addresses to Program Data Objects

Once a PC from a hardware counter event corresponding to a memory operation has been processed to successfully backtrack to a likely causal memory-referencing instruction, the Analyzer uses instruction identifiers and descriptors provided by the compiler in its hardware profiling support information to derive the associated program data object.

The term *data object* is used to refer to program constants, variables, arrays and aggregates such as structures and unions, along with distinct aggregate elements, described in source code. Depending on the source language, data object types and their sizes vary. Many data objects are explicitly named in source programs, while others may be unnamed. Some data objects are derived or aggregated from other (simpler) data objects, resulting in a rich, often complex, set of data objects.

Each data object has an associated scope, the region of the source program where it is defined and can be referenced, which may be global (such as a load object), a particular compilation unit (an object file), or function. Identical data objects may be defined with different scopes, or particular data objects referred to differently in different scopes.

Data-derived metrics from hardware counter events for memory operations collected with backtracking enabled are attributed to the associated program data object type and propagate to any aggregates containing the data object and the artificial `<Total>`, which is considered to contain all data objects (including `<Unknown>` and `<Scalars>`). The different subtypes of `<Unknown>` propagate up to the `<Unknown>` aggregate. The following section describes the `<Total>`, `<Scalars>`, and `<Unknown>` data objects.

Data Object Descriptors

Data objects are fully described by a combination of their declared type and name. A simple scalar data object `{int i}` describes a variable called `i` of type `int`, while `{const+pointer+int` p} describes a constant pointer to a type int called p. Spaces in the type names are replaced with underscore (_), and unnamed data objects are represented with a name of dash (-), for example: {double_precision_complex -}.

An entire aggregate is similarly represented {structure:foo_t} for a structure of type foo_t. An element of an aggregate requires the additional specification of its container, for example, {structure:foo_t}.{int i} for a member i of type int of the previous structure of type foo_t. Aggregates can also themselves be elements of (larger) aggregates, with their corresponding descriptor constructed as a concatenation of aggregate descriptors and ultimately a scalar descriptor.

While a fully-qualified descriptor may not always be necessary to disambiguate data objects, it provides a generic complete specification to assist with data object identification.

The <Total> Data Object

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

The <Scalars> Data Object

While aggregate elements have their performance metrics additionally attributed into the metric value for their associated aggregate, all of the scalar constants and variables have their performance metrics additionally attributed into the metric value for the artificial <Scalars> data object.

The <Unknown> Data Object and Its Elements

Under various circumstances, event data can not be mapped to a particular data object. In such cases, the data is mapped to the special data object named <Unknown> and one of its elements as described below.

- Module with trigger PC not compiled with -xhwcprof

  No event-causing instruction or data object was identified because the object code was not compiled with hardware counter profiling support.
- Backtracking failed to find a valid branch target

  No event-causing instruction was identified because the hardware profiling support information provided in the compilation object was insufficient to verify the validity of backtracking.
- Backtracking traversed a branch target

  No event-causing instruction or data object was identified because backtracking encountered a control transfer target in the instruction stream.

- No identifying descriptor provided by the compiler

  Backtracking determined the likely causal memory-referencing instruction, but its associated data object was not specified by the compiler.
- No type information

  Backtracking determined the likely event-causing instruction, but the instruction was not identified by the compiler as a memory-referencing instruction.
- Not determined from the symbolic information provided by the compiler

  Backtracking determined the likely causal memory-referencing instruction, but it was not identified by the compiler and associated data object determination is therefore not possible. Compiler temporaries are generally unidentified.
- Backtracking was prevented by a jump or call instruction

  No event-causing instructions were identified because backtracking encountered a branch or call instruction in the instruction stream.
- Backtracking did not find trigger PC

  No event-causing instructions were found within the maximum backtracking range.
- Could not determine VA because registers changed after trigger instruction

  The virtual address of the data object was not determined because registers were overwritten during hardware counter skid.
- Memory-referencing instruction did not specify a valid VA

  The virtual address of the data object did not appear to be valid.

Mapping Performance Data to Memory Objects

Memory objects are components in the memory subsystem, such as cache-lines, pages, and memory-banks. The object is determined from an index computed from the virtual and/or physical address as recorded. Memory objects are predefined for virtual pages and physical pages, for sizes of 8 KB, 64 KB, 512 KB, and 4 MB. You can define others with the `mobj_define` command in the `er_print` utility. You can also define custom memory objects using the Add Memory Objects dialog box in the Analyzer, which you can open by clicking the Add Custom Memory Object button in the Set Data Presentation dialog box.

CHAPTER 7

Understanding Annotated Source and Disassembly Data

Annotated source code and annotated disassembly code are useful for determining which source lines or instructions within a function are responsible for poor performance, and to view commentary on how the compiler has performed transformations on the code. This section describes the annotation process and some of the issues involved in interpreting the annotated code.

This chapter covers the following topics:

How the Tools Find Source Code

In order to display annotated source code and annotated disassembly code, the Performance Analyzer and `er_print` utility must have access to the source code and load object files used by the program on which an experiment was run.

Load object files are first looked for in the `archives` directory of the experiment. If they are not found there, they are looked for using the same algorithm as source and object files, described below.

In most experiments, source and object files are recorded in the form of full paths. Java source files also have a package name which lists the directory structure to the file. If you view an experiment on the same system where it was recorded, the source files and load object can be found using the full paths. When experiments are moved or looked at on a different machine, those full paths might not be accessible.

Two complementary methods are used to locate source and object files: path mapping and searching a path. The same methods are used to find load object files if they are not found in the `archives` subdirectory.

You can set path maps and search paths to help the tools find the files referenced by your experiment. In the Analyzer, use the Set Data Preferences dialog box to set path maps in the Pathmaps tab, and use the Search Path tab to set the search path as described in "Setting Data Presentation Options" on page 107. For the `er_print` utility, use the `pathmap` and `setpath` directives described in "Commands That Control Searching For Source Files" on page 133.

Path mapping is applied first and specifies how to replace the beginning of a full file path with a different path. For example, if a file is specified as `/a/b/c/sourcefile`, and a `pathmap` directive specifies mapping `/a/` to `/x/y/`, the file could be found in `/x/y/b/c/sourcefile`. A `pathmap` directive that maps `/a/b/c/` to `/x/`, would allow the file to be found in `/x/sourcefile`.

If path mapping does not find the file, the search path is used. The search path gives a list of directories to be searched for a file with the given base name, which is `sourcefile` in the example above. You can set the search path with the `setpath` command, and append a directory to the search path with the `addpath` command. For Java files the package name is tried and then the base name is tried.

Each directory in the search path is used to construct a full path to try. For Java source files two full paths are constructed, one for the base name and one for the package name. The tools apply the path mapping to each of the full paths and if none of the mapped paths point to the file, the next search path directory is tried.

If the file is not found in the search path and no path mapping prefix matched the original full path, the original full path is tried. If any path map prefix matched the original full path, but the file was not found, the original full path is not tried.

Note that the default search path includes the current directory and the experiment directories, so one way to make source files accessible is to copy them to either of those places, or to put symbolic links in those places pointing to the current location of the source file.

Annotated Source Code

Annotated source code for an experiment can be viewed in the Performance Analyzer by selecting the Source tab in the left pane of the Analyzer window. Alternatively, annotated source code can be viewed without running an experiment, using the `er_src` utility. This section of the manual describes how source code is displayed in the Performance Analyzer. For details on viewing annotated source code with the `er_src` utility, see "Viewing Source/Disassembly Without An Experiment" on page 209.

Annotated source in the Analyzer contains the following information:

- The contents of the original source file
- The performance metrics of each line of executable source code
- Highlighting of code lines with metrics exceeding a specific threshold
- Index lines
- Compiler commentary

Performance Analyzer Source Tab Layout

The Source tab is divided into columns, with fixed-width columns for individual metrics on the left and the annotated source taking up the remaining width on the right.

Identifying the Original Source Lines

All lines displayed in black in the annotated source are taken from the original source file. The number at the start of a line in the annotated source column corresponds to the line number in the original source file. Any lines with characters displayed in a different color are either index lines or compiler commentary lines.

Index Lines in the Source Tab

A source file is any file compiled to produce an object file or interpreted into byte code. An object file normally contains one or more regions of executable code corresponding to functions, subroutines, or methods in the source code. The Analyzer analyzes the object file, identifies each executable region as a function, and attempts to map the functions it finds in the object code to the functions, routines, subroutines, or methods in the source file associated with the object code. When the analyzer succeeds, it adds an index line in the annotated source file in the location corresponding to the first instruction in the function found in the object code.

The annotated source shows an index line for every function, including inline functions, even though inline functions are not displayed in the list displayed by the Function tab. The Source tab displays index lines in red italics with text in angle-brackets. The simplest type of index line corresponds to the function's default context. The default source context for any function is defined as the source file to which the first instruction in that function is attributed. The following example shows an index line for a C function `icputime`.

```
                    578. int
                    579. icputime(int k)
0.        0.        580. {
                         <Function: icputime>
```

As can be seen from the above example, the index line appears on the line following the first instruction. For C source, the first instruction corresponds to the opening brace at the start of the function body. In Fortran source, the index line for each subroutine follows the line containing the `subroutine` keyword. Also, a `main` function index line follows the first Fortran source instruction executed when the application starts, as shown in the following example:

```
                                            1. ! Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
                                            2. ! @(#)omptest.f 1.11 10/03/24 SMI
                                            3. ! Synthetic f90 program, used for testing openmp directives and the
                                            4. !      analyzer
                                            5.
0.       0.       0.       0.               6.       program omptest
                                              <Function: MAIN>
                                            7.
                                            8. !$PRAGMA C (gethrtime, gethrvtime)
```

Sometimes, the Analyzer might not be able to map a function it finds in the object code with any programming instructions in the source file associated with that object code; for example, code may be `#included` or inlined from another file, such as a header file.

Also displayed in red are special index lines and other special lines that are not compiler commentary. For example, as a result of compiler optimization, a special index line might be created for a function in the object code that does not correspond to code written in any source file. For details, refer to "Special Lines in the Source, Disassembly and PCs Tabs" on page 203.

Compiler Commentary

Compiler commentary indicates how compiler-optimized code has been generated. Compiler commentary lines are displayed in blue, to distinguish them from index lines and original source lines. Various parts of the compiler can incorporate commentary into the executable. Each comment is associated with a specific line of source code. When the annotated source is written, the compiler commentary for any source line appears immediately preceding the source line.

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

```
0.       0.       0.       0.          28.       SUBROUTINE dgemv_g2 (transa, m, n, alpha, b, ldb,   &
                                       29.      &                   c, incc, beta, a, inca)
                                       30.       CHARACTER (KIND=1) :: transa
                                       31.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                       32.       REAL      (KIND=8) :: alpha, beta
                                       33.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                       34.       INTEGER            :: i, j
                                       35.       REAL      (KIND=8) :: tmr, wtime, tmrend
                                       36.       COMMON/timer/ tmr
                                       37.
                               Function wtime_ not inlined because the compiler has not seen
                               the body of the routine
0.       0.       0.       0.          38.       tmrend = tmr + wtime()


                               Function wtime_ not inlined because the compiler has not seen
                               the body of the routine
                               Discovered loop below has tag L16
0.       0.       0.       0.          39.       DO WHILE(wtime() < tmrend)
```

```
                             Array statement below generated loop L4
0.       0.       0.       0.        40.       a(1:m) = 0.0
                                     41.

                             Source loop below has tag L6
0.       0.       0.       0.        42.       DO j = 1, n       ! <=-----\ swapped loop indices

                             Source loop below has tag L5
                             L5 cloned for unrolling-epilog.  Clone is L19
                             All 8 copies of L19 are fused together as part of unroll and jam
                             L19 scheduled with steady-state cycle count = 9
                             L19 unrolled 4 times
                             L19 has 9 loads, 1 stores, 8 prefetches, 8 FPadds,
                             8 FPmuls, and 0 FPdivs per iteration
                             L19 has 0 int-loads, 0 int-stores, 11 alu-ops, 0 muls,
                             0 int-divs and 0 shifts per iteration
                             L5 scheduled with steady-state cycle count = 2
                             L5 unrolled 4 times
                             L5 has 2 loads, 1 stores, 1 prefetches, 1 FPadds, 1 FPmuls,
                             and 0 FPdivs per iteration
                             L5 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls,
                             0 int-divs and 0 shifts per iteration
0.210    0.210    0.210    0.        43.          DO i = 1, m
4.003    4.003    4.003    0.050     44.             a(i) = a(i) + b(i,j) * c(j)
0.240    0.240    0.240    0.        45.          END DO
0.       0.       0.       0.        46.       END DO
                                     47.       END DO
                                     48.
0.       0.       0.       0.        49.       RETURN
0.       0.       0.       0.        50.       END
```

You can set the types of compiler commentary displayed in the Source tab using the Source/Disassembly tab in the Set Data Presentation dialog box; for details, see “Setting Data Presentation Options” on page 107.

Common Subexpression Elimination

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

Loop Optimizations

The compiler can do several types of loop optimization. Some of the more common ones are as follows:

- Loop unrolling
- Loop peeling

- Loop interchange
- Loop fission
- Loop fusion

Loop unrolling consists of repeating several iterations of a loop within the loop body, and adjusting the loop index accordingly. As the body of the loop becomes larger, the compiler can schedule the instructions more efficiently. Also reduced is the overhead caused by the loop index increment and conditional check operations. The remainder of the loop is handled using loop peeling.

Loop peeling consists of removing a number of loop iterations from the loop, and moving them in front of or after the loop, as appropriate.

Loop interchange changes the ordering of nested loops to minimize memory stride, to maximize cache-line hit rates.

Loop fusion consists of combining adjacent or closely located loops into a single loop. The benefits of loop fusion are similar to loop unrolling. In addition, if common data is accessed in the two pre-optimized loops, cache locality is improved by loop fusion, providing the compiler with more opportunities to exploit instruction-level parallelism.

Loop fission is the opposite of loop fusion: a loop is split into two or more loops. This optimization is appropriate if the number of computations in a loop becomes excessive, leading to register spills that degrade performance. Loop fission can also come into play if a loop contains conditional statements. Sometimes it is possible to split the loops into two: one with the conditional statement and one without. This can increase opportunities for software pipelining in the loop without the conditional statement.

Sometimes, with nested loops, the compiler applies loop fission to split a loop apart, and then performs loop fusion to recombine the loop in a different way to increase performance. In this case, you see compiler commentary similar to the following:

```
Loop below fissioned into 2 loops
Loop below fused with loop on line 116
[116]    for (i=0;i<nvtxs;i++) {
```

Inlining of Functions

With an inline function, the compiler inserts the function instructions directly at the locations where it is called instead of making actual function calls. Thus, similar to a C/C++ macro, the instructions of an inline function are replicated at each call location. The compiler performs explicit or automatic inlining at high optimization levels (4 and 5). Inlining saves the cost of a function call and provides more instructions for which register usage and instruction scheduling can be optimized, at the cost of a larger code footprint in memory. The following is an example of inlining compiler commentary.

```
                Function initgraph inlined from source file ptralias.c
                    into the code for the following line
0.       0.         44.       initgraph(rows);
```

Note – The compiler commentary does not wrap onto two lines in the Source tab of the Analyzer.

Parallelization

If your code contains Sun, Cray, or OpenMP parallelization directives, it can be compiled for parallel execution on multiple processors. The compiler commentary indicates where parallelization has and has not been performed, and why. The following shows an example of parallelization computer commentary.

```
0.       6.324       9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)
                   Loop below parallelized by explicit user directive
                   Loop below interchanged with loop on line 12
0.010    0.010     [10]            do i = 2, n-1

                   Loop below not parallelized because it was nested in a parallel loop
                   Loop below interchanged with loop on line 12
0.170    0.170      11.               do j = 2, i
```

For more details about parallel execution and compiler-generated body functions, refer to "Overview of OpenMP Software Execution" on page 173.

Special Lines in the Annotated Source

Several other annotations for special cases can be shown under the Source tab, either in the form of compiler commentary, or as special lines displayed in the same color as index lines. For details, refer to "Special Lines in the Source, Disassembly and PCs Tabs" on page 203.

Source Line Metrics

Source code metrics are displayed, for each line of executable code, in fixed-width columns. The metrics are the same as in the function list. You can change the defaults for an experiment using a `.er.rc` file; for details, see "Commands That Set Defaults" on page 146. You can also change the metrics displayed and highlighting thresholds in the Analyzer using the Set Data Presentation dialog box; for details, see "Setting Data Presentation Options" on page 107.

Annotated source code shows the metrics of an application at the source-line level. It is produced by taking the PCs (program counts) that are recorded in the application's call stack, and mapping each PC to a source line. To produce an annotated source file, the Analyzer first determines all of the functions that are generated in a particular object module (`.o` file) or load object, then scans the data for all PCs from each function. In order to produce annotated source, the Analyzer must be able to find and read the object module or load object to determine the mapping from PCs to source lines, and it must be able to read the source file to produce an annotated copy, which is displayed. See "How the Tools Find Source Code" on page 191 for a description of the process used to find an experiment's source code.

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

Interpreting Source Line Metrics

Metrics for an instruction must be interpreted as metrics accrued while waiting for the instruction to be executed. If the instruction being executed when an event is recorded comes from the same source line as the leaf PC, the metrics can be interpreted as due to execution of that source line. However, if the leaf PC comes from a different source line than the instruction being executed, at least some of the metrics for the source line that the leaf PC belongs to must be interpreted as metrics accumulated while this line was waiting to be executed. An example is when a value that is computed on one source line is used on the next source line.

The issue of how to interpret the metrics matters most when there is a substantial delay in execution, such as at a cache miss or a resource queue stall, or when an instruction is waiting for a result from a previous instruction. In such cases the metrics for the source lines can seem to be unreasonably high, and you should look at other nearby lines in the code to find the line responsible for the high metric value.

Metric Formats

The four possible formats for the metrics that can appear on a line of annotated source code are explained in Table 7–1.

TABLE 7–1 Annotated Source-Code Metrics

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

TABLE 7–1 Annotated Source-Code Metrics *(Continued)*

Metric	Significance
`0.000`	At least one PC from this line appeared in the data, but the computed metric value rounded to zero.
`1.234`	The metrics for all PCs attributed to this line added up to the non-zero numerical value shown.

Annotated Disassembly Code

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

- If the object module is not known, the Analyzer disassembles the instructions for just the specified function, and does not show any source lines in the disassembly.
- If the object module is known, the disassembly covers all functions within the object module.
- If the source file is available, and line number data is recorded, the Analyzer can interleave the source with the disassembly, depending on the display preference.
- If the compiler has inserted any commentary into the object code, it too, is interleaved in the disassembly if the corresponding preferences are set.

Each instruction in the disassembly code is annotated with the following information.

- A source line number, as reported by the compiler
- Its relative address
- The hexadecimal representation of the instruction, if requested
- The assembler ASCII representation of the instruction

Where possible, call addresses are resolved to symbols (such as function names). Metrics are shown on the lines for instructions, and can be shown on any interleaved source code if the corresponding preference is set. Possible metric values are as described for source-code annotations, in Table 7–1.

The disassembly listing for code that is `#included` in multiple locations repeats the disassembly instructions once for each time that the code has been `#included`. The source code is interleaved only for the first time a repeated block of disassembly code is shown in a file. For example, if a block of code defined in a header called `inc_body.h` is `#included` by four functions named `inc_body` , `inc_entry`, `inc_middle`, and `inc_exit`, then the block of disassembly instructions appears four times in the disassembly listing for `inc_body.h`, but the source code is interleaved only in the first of the four blocks of disassembly instructions. Switching to Source tab reveals index lines corresponding to each of the times that the disassembly code was repeated.

Index lines can be displayed in the Disassembly tab. Unlike with the Source tab, these index lines cannot be used directly for navigation purposes. However, placing the cursor on one of the instructions immediately below the index line and selecting the Source tab navigates you to the file referenced in the index line.

Files that `#include` code from other files show the included code as raw disassembly instructions without interleaving the source code. However, placing the cursor on one of these instructions and selecting the Source tab opens the file containing the `#included` code. Selecting the Disassembly tab with this file displayed shows the disassembly code with interleaved source code.

Source code can be interleaved with disassembly code for inline functions, but not for macros.

When code is not optimized, the line numbers for each instruction are in sequential order, and the interleaving of source lines and disassembled instructions occurs in the expected way. When optimization takes place, instructions from later lines sometimes appear before those from earlier lines. The Analyzer's algorithm for interleaving is that whenever an instruction is shown as coming from line *N*, all source lines up to and including line *N* are written before the instruction. One effect of optimization is that source code can appear between a control transfer instruction and its delay slot instruction. Compiler commentary associated with line *N* of the source is written immediately before that line.

Interpreting Annotated Disassembly

Interpreting annotated disassembly is not straightforward. The leaf PC is the address of the next instruction to execute, so metrics attributed to an instruction should be considered as time spent waiting for the instruction to execute. However, the execution of instructions does not always happen in sequence, and there might be delays in the recording of the call stack. To make use of annotated disassembly, you should become familiar with the hardware on which you record your experiments and the way in which it loads and executes instructions.

The next few subsections discuss some of the issues of interpreting annotated disassembly.

Instruction Issue Grouping

Instructions are loaded and issued in groups known as instruction issue groups. Which instructions are in the group depends on the hardware, the instruction type, the instructions already being executed, and any dependencies on other instructions or registers. As a result, some instructions might be underrepresented because they are always issued in the same clock cycle as the previous instruction, so they never represent the next instruction to be executed. And when the call stack is recorded, there might be several instructions that could be considered the next instruction to execute.

Instruction issue rules vary from one processor type to another, and depend on the instruction alignment within cache lines. Since the linker forces instruction alignment at a finer granularity than the cache line, changes in a function that might seem unrelated can cause different alignment of instructions. The different alignment can cause a performance improvement or degradation.

The following artificial situation shows the same function compiled and linked in slightly different circumstances. The two output examples shown below are the annotated disassembly listings from the `er_print` utility. The instructions for the two examples are identical, but the instructions are aligned differently.

In the following output example the instruction alignment maps the two instructions `cmp` and `bl,a` to different cache lines, and a significant amount of time is used waiting to execute these two instructions.

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                             1. static int
                             2. ifunc()
                             3. {
                             4.     int i;
                             5.
                             6.     for (i=0; i<10000; i++)
                                <function: ifunc>
   0.010     0.010              [ 6]    1066c:  clr         %o0
   0.        0.                 [ 6]    10670:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    10674:  inc         784, %o5
                             7.         i++;
   0.        0.                 [ 7]    10678:  inc         2, %o0
## 1.360     1.360              [ 7]    1067c:  cmp         %o0, %o5
## 1.510     1.510              [ 7]    10680:  bl,a        0x1067c
   0.        0.                 [ 7]    10684:  inc         2, %o0
   0.        0.                 [ 7]    10688:  retl
   0.        0.                 [ 7]    1068c:  nop
                             8.     return i;
                             9. }
```

In the next output example, the instruction alignment maps the two instructions `cmp` and `bl,a` to the same cache line, and a significant amount of time is used waiting to execute only one of these instructions.

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                             1. static int
                             2. ifunc()
                             3. {
                             4.     int i;
                             5.
                             6.     for (i=0; i<10000; i++)
                                <function: ifunc>
   0.        0.                 [ 6]    10684:  clr         %o0
   0.        0.                [ 6]    10688:  sethi       %hi(0x2400), %o5
   0.        0.                [ 6]    1068c:  inc         784, %o5
                             7.        i++;
   0.        0.                [ 7]    10690:  inc         2, %o0
## 1.440     1.440             [ 7]    10694:  cmp         %o0, %o5
   0.        0.                [ 7]    10698:  bl,a        0x10694
   0.        0.                [ 7]    1069c:  inc         2, %o0
   0.        0.                [ 7]    106a0:  retl
   0.        0.                [ 7]    106a4:  nop
                             8.     return i;
                             9. }
```

Instruction Issue Delay

Sometimes, specific leaf PCs appear more frequently because the instruction that they represent is delayed before issue. This appearance can occur for a number of reasons, some of which are listed below:

- The previous instruction takes a long time to execute and is not interruptible, for example when an instruction traps into the kernel.
- An arithmetic instruction needs a register that is not available because the register contents were set by an earlier instruction that has not yet completed. An example of this sort of delay is a load instruction that has a data cache miss.
- A floating-point arithmetic instruction is waiting for another floating-point instruction to complete. This situation occurs for instructions that cannot be pipelined, such as square root and floating-point divide.
- The instruction cache does not include the memory word that contains the instruction (I-cache miss).

Attribution of Hardware Counter Overflows

Apart from TLB misses on some platforms, the call stack for a hardware counter overflow event is recorded at some point further on in the sequence of instructions than the point at which the overflow occurred, for various reasons including the time taken to handle the interrupt generated by the overflow. For some counters, such as cycles or instructions issued, this delay does not matter. For other counters, such as those counting cache misses or floating point operations, the metric is attributed to a different instruction from that which is responsible for the overflow. Often the PC that caused the event is only a few instructions before the recorded PC, and the instruction can be correctly located in the disassembly listing. However, if there is a branch target within this instruction range, it might be difficult or impossible to tell which instruction corresponds to the PC that caused the event. For hardware counters that count memory access events, the Collector searches for the PC that caused the event if the counter name is prefixed with a plus, +. The data recorded in this way supports dataspace profiling. See "Dataspace Profiling and Memoryspace Profiling" on page 165 and "`-h` *counter_definition_1*`...[,`*counter_definition_n*`]`" on page 60 for more information.

Systems that have processors with counters that are labeled with the `precise` keyword allow memoryspace profiling without any special compilation of binaries. For example the SPARC T2, SPARC T3, and SPARC T4 processors provide several precise counters. Run the `collect –h` command and look for the `precise` keyword to determine if your system allows memoryspace profiling.

For example, running the following command on a system with the SPARC T4 processor shows the precise raw counters available:

```
% collect -h | & grep -i precise | grep -v alias
    Instr_ld[/{0|1|2|3}],1000003 (precise load-store events)
    Instr_st[/{0|1|2|3}],1000003 (precise load-store events)
    SW_prefetch[/{0|1|2|3}],1000003 (precise load-store events)
    Block_ld_st[/{0|1|2|3}],1000003 (precise load-store events)
    DC_miss_L2_L3_hit_nospec[/{0|1|2|3}],1000003 (precise load-store events)
    DC_miss_local_hit_nospec[/{0|1|2|3}],1000003 (precise load-store events)
    DC_miss_remote_L3_hit_nospec[/{0|1|2|3}],1000003 (precise load-store events)
    DC_miss_nospec[/{0|1|2|3}],1000003 (precise load-store events)
```

Special Lines in the Source, Disassembly and PCs Tabs

The Performance Analyzer displays some lines in the Source, Disassembly and PCs tabs that do not directly correspond to lines of code, instructions, or program counters. The following sections describe these special lines.

Outline Functions

Outline functions can be created during feedback-optimized compilations. They are displayed as special index lines in the Source tab and Disassembly tab. In the Source tab, an annotation is displayed in the block of code that has been converted into an outline function.

```
                   Function binsearchmod inlined from source file ptralias2.c into the
0.       0 .         58.         if( binsearchmod( asize, &element ) ) {
0.240    0.240       59.             if( key != (element << 1) ) {
0.       0.          60.                 error |= BINSEARCHMODPOSTESTFAILED;
                        <Function: main -- outline code from line 60 [_$o1B60.main]>
0.040    0.040     [ 61]                 break;
0.       0.          62.             }
0.       0.          63.         }
```

In the Disassembly tab, the outline functions are typically displayed at the end of the file.

```
                        <Function: main -- outline code from line 85 [_$o1D85.main]>
0.       0.             [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.       0.             [ 86] 100001038:  bset        4, %i3
0.       0.             [ 85] 10000103c:  or          %i5, 1, %l7
0.       0.             [ 85] 100001040:  sllx        %l7, 12, %l5
0.       0.             [ 85] 100001044:  call        printf ! 0x100101300
```

```
0.       0.             [ 85] 100001048:  add         %l5, 336, %o0
0.       0.             [ 90] 10000104c:  cmp         %i3, 0
0.       0.             [ 20] 100001050:  ba,a        0x1000010b4
                        <Function: main -- outline code from line 46 [_$o1A46.main]>
0.       0.             [ 46] 100001054:  mov         1, %i3
0.       0.             [ 47] 100001058:  ba          0x100001090
0.       0.             [ 56] 10000105c:  clr         [%i2]
                        <Function: main -- outline code from line 60 [_$o1B60.main]>
0.       0.             [ 60] 100001060:  bset        2, %i3
0.       0.             [ 61] 100001064:  ba          0x10000109c
0.       0.             [ 74] 100001068:  mov         1, %o3
```

The name of the outline function is displayed in square brackets, and encodes information about the section of outlined code, including the name of the function from which the code was extracted and the line number of the beginning of the section in the source code. These mangled names can vary from release to release. The Analyzer provides a readable version of the function name. For further details, refer to “Outline Functions” on page 184.

If an outline function is called when collecting the performance data for an application, the Analyzer displays a special line in the annotated disassembly to show inclusive metrics for that function. For further details, see “Inclusive Metrics” on page 208.

Compiler-Generated Body Functions

When a compiler parallelizes a loop in a function, or a region that has parallelization directives, it creates new body functions that are not in the original source code. These functions are described in “Overview of OpenMP Software Execution” on page 173.

The compiler assigns mangled names to body functions that encode the type of parallel construct, the name of the function from which the construct was extracted, the line number of the beginning of the construct in the original source, and the sequence number of the parallel construct. These mangled names vary from release to release of the microtasking library, but are shown demangled into more comprehensible names.

The following shows a typical compiler-generated body function as displayed in the functions list in machine mode.

```
7.415      14.860      psec_ -- OMP sections from line 9 [_$s1A9.psec_]
3.873       3.903      craydo_ -- MP doall from line 10 [_$d1A10.craydo_]
```

As can be seen from the above examples, the name of the function from which the construct was extracted is shown first, followed by the type of parallel construct, followed by the line number of the parallel construct, followed by the mangled name of the compiler-generated body function in square brackets. Similarly, in the disassembly code, a special index line is generated.

```
0.       0.            <Function: psec_ -- OMP sections from line 9 [_$s1A9.psec_]>
0.       7.445         [24]    1d8cc:  save        %sp, -168, %sp
0.       0.            [24]    1d8d0:  ld          [%i0], %g1
0.       0.            [24]    1d8d4:  tst         %i1
```

```
0.       0.            <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_]>
0.       0.030         [ ?]    197e8:  save        %sp, -128, %sp
0.       0.            [ ?]    197ec:  ld          [%i0 + 20], %i5
0.       0.            [ ?]    197f0:  st          %i1, [%sp + 112]
0.       0.            [ ?]    197f4:  ld          [%i5], %i3
```

With Cray directives, the function may not be correlated with source code line numbers. In such cases, a `[ ?]` is displayed in place of the line number. If the index line is shown in the annotated source code, the index line indicates instructions without line numbers, as shown below.

```
                     9. c$mic  doall shared(a,b,c,n) private(i,j,k)

                   Loop below fused with loop on line 23
                   Loop below not parallelized because autoparallelization
                     is not enabled
                   Loop below autoparallelized
                   Loop below interchanged with loop on line 12
                   Loop below interchanged with loop on line 12
3.873     3.903         <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_],
                      instructions without line numbers>
0.        3.903     10.            do i = 2, n-1
```

Note – Index lines and compiler-commentary lines do not wrap in the real displays.

Dynamically Compiled Functions

Dynamically compiled functions are functions that are compiled and linked while the program is executing. The Collector has no information about dynamically compiled functions that are written in C or C++, unless the user supplies the required information using the Collector API function `collector_func_load()`. Information displayed by the Function tab, Source tab, and Disassembly tab depends on the information passed to `collector_func_load()` as follows:

- If information is not supplied, `collector_func_load()` is not called; the dynamically compiled and loaded function appears in the function list as `<Unknown>`. Neither function source nor disassembly code is viewable in the Analyzer.
- If no source file name and no line-number table is provided, but the name of the function, its size, and its address are provided, the name of the dynamically compiled and loaded function and its metrics appear in the function list. The annotated source is available, and the disassembly instructions are viewable, although the line numbers are specified by `[?]` to indicate that they are unknown.
- If the source file name is given, but no line-number table is provided, the information displayed by the Analyzer is similar to the case where no source file name is given, except that the beginning of the annotated source displays a special index line indicating that the function is composed of instructions without line numbers. For example:

```
1.121     1.121          <Function func0, instructions without line numbers>
                      1. #include        <stdio.h>
```

- If the source file name and line-number table is provided, the function and its metrics are displayed by the Function tab, Source tab, and Disassembly tab in the same way as conventionally compiled functions.

For more information about the Collector API functions, see "Dynamic Functions and Modules" on page 50.

For Java programs, most methods are interpreted by the JVM software. The Java HotSpot virtual machine, running in a separate thread, monitors performance during the interpretive execution. During the monitoring process, the virtual machine may decide to take one or more interpreted methods, generate machine code for them, and execute the more-efficient machine-code version, rather than interpret the original.

For Java programs, there is no need to use the Collector API functions; the Analyzer signifies the existence of Java HotSpot-compiled code in the annotated disassembly listing using a special line underneath the index line for the method, as shown in the following example.

```
                   11.    public int add_int () {
                   12.       int       x = 0;
2.832     2.832      <Function: Routine.add_int: HotSpot-compiled leaf instructions>
0.        0.         [ 12] 00000000: iconst_0
0.        0.         [ 12] 00000001: istore_1
```

The disassembly listing only shows the interpreted byte code, not the compiled instructions. By default, the metrics for the compiled code are shown next to the special line. The exclusive and inclusive CPU times are different than the sum of all the inclusive and exclusive CPU times shown for each line of interpreted byte code. In general, if the method is called on several occasions, the CPU times for the compiled instructions are greater than the sum of the CPU times for the interpreted byte code, because the interpreted code is executed only once when the method is initially called, whereas the compiled code is executed thereafter.

The annotated source does not show Java HotSpot-compiled functions. Instead, it displays a special index line to indicate instructions without line numbers. For example, the annotated source corresponding to the disassembly extract shown above is as follows:

```
                     11.    public int add_int () {
2.832     2.832        <Function: Routine.add_int(), instructions without line numbers>
0.        0.         12.       int       x = 0;
                       <Function: Routine.add_int()>
```

Java Native Functions

Native code is compiled code originally written in C, C++, or Fortran, called using the Java Native Interface (JNI) by Java code. The following example is taken from the annotated disassembly of file `jsynprog.java` associated with demo program `jsynprog`.

```
                     5. class jsynprog
                        <Function: jsynprog.<init>()>
0.       5.504          jsynprog.JavaCC() <Java native method>
0.       1.431          jsynprog.JavaCJava(int) <Java native method>
0.       5.684          jsynprog.JavaJavaC(int) <Java native method>
0.       0.             [  5] 00000000: aload_0
0.       0.             [  5] 00000001: invokespecial <init>()
0.       0.             [  5] 00000004: return
```

Because the native methods are not included in the Java source, the beginning of the annotated source for `jsynprog.java` shows each Java native method using a special index line to indicate instructions without line numbers.

```
0.       5.504          <Function: jsynprog.JavaCC(), instructions without line
                           numbers>
0.       1.431          <Function: jsynprog.JavaCJava(int), instructions without line
                           numbers>
0.       5.684          <Function: jsynprog.JavaJavaC(int), instructions without line
                           numbers>
```

Note – The index lines do not wrap in the real annotated source display.

Cloned Functions

The compilers have the ability to recognize calls to a function for which extra optimization can be performed. An example of such is a call to a function where some of the arguments passed are constants. When the compiler identifies particular calls that it can optimize, it creates a copy of the function, which is called a clone, and generates optimized code.

In the annotated source, compiler commentary indicates if a cloned function has been created:

```
0.       0.       Function foo from source file clone.c cloned,
                   creating cloned function _$c1A.foo;
                   constant parameters propagated to clone
0.       0.570     27.    foo(100, 50, a, a+50, b);
```

Note – Compiler commentary lines do not wrap in the real annotated source display.

The clone function name is a mangled name that identifies the particular call. In the above example, the compiler commentary indicates that the name of the cloned function is `_$c1A.foo`. This function can be seen in the function list as follows:

```
0.350     0.550     foo
0.340     0.570     _$c1A.foo
```

Each cloned function has a different set of instructions, so the annotated disassembly listing shows the cloned functions separately. They are not associated with any source file, and therefore the instructions are not associated with any source line numbers. The following shows the first few lines of the annotated disassembly for a cloned function.

```
0.       0.           <Function: _$c1A.foo>
0.       0.           [?]    10e98:  save        %sp, -120, %sp
0.       0.           [?]    10e9c:  sethi       %hi(0x10c00), %i4
0.       0.           [?]    10ea0:  mov         100, %i3
0.       0.           [?]    10ea4:  st          %i3, [%i0]
0.       0.           [?]    10ea8:  ldd         [%i4 + 640], %f8
```

Static Functions

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

```
0.       0.       <static>@0x18780
0.       0.       <static>@0x20cc
0.       0.       <static>@0xc9f0
0.       0.       <static>@0xd1d8
0.       0.       <static>@0xe204
```

In the PCs tab, the above functions are represented with an offset, as follows:

```
0.       0.       <static>@0x18780 + 0x00000818
0.       0.       <static>@0x20cc + 0x0000032C
0.       0.       <static>@0xc9f0 + 0x00000060
0.       0.       <static>@0xd1d8 + 0x00000040
0.       0.       <static>@0xe204 + 0x00000170
```

An alternative representation in the PCs tab of functions called within a stripped library is:

```
<library.so> -- no functions found + 0x0000F870
```

Inclusive Metrics

In the annotated disassembly, special lines exist to tag the time taken by outline functions.

The following shows an example of the annotated disassembly displayed when an outline function is called:

```
0.      0.        43.         else
0.      0.        44.         {
0.      0.        45.                 printf("else reached\n");
0.      2.522         <inclusive metrics for outlined functions>
```

Branch Target

An artificial line, `<branch target>`, shown in the annotated disassembly listing, corresponds to a PC of an instruction where the backtracking to find its effective address fails because the backtracking algorithm runs into a branch target.

Annotations for Store and Load Instructions

When you compile with the `-xhwcprof` option, the compilers generate additional information for store (`st`) and load (`ld`) instructions. You can view the annotated `st` and `ld` instructions in disassembly listings.

Viewing Source/Disassembly Without An Experiment

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

```
er_src [ -func | -{source,src} item tag | -{disasm,dis} item tag |
-{cc,scc,dcc} com_spec | -outfile filename | -V ] object
```

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

item is the name of a function or of a source or object file used to build the executable or shared object. *item* can also be specified in the form *function'file'*, in which case `er_src` displays the source or disassembly of the named function in the source context of the named file.

tag is an index used to determine which *item* is being referred to when multiple functions have the same name. It is required, but is ignored if not necessary to resolve the function.

The special item and tag, `all -1`, tells `er_src` to generate the annotated source or disassembly for all functions in the object.

Note – The output generated as a result of using `all -1` on executables and shared objects may be very large.

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

`-func`

List all the functions from the given object.

`-{source,src}` *item tag*

Show the annotated source for the listed item.

`-{disasm,dis}` *item tag*

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

`-{cc,scc,dcc}` *com-spec*

Specify which classes of compiler commentary classes to show. *com-spec* is a list of classes separated by colons. The *com-spec* is applied to source compiler commentary if the `-scc` option is used, to disassembly commentary if the `-dcc` option is used, or to both source and disassembly commentary if `-cc` is used. See "Commands That Control the Source and Disassembly Listings" on page 129 for a description of these classes.

The commentary classes can be specified in a defaults file. The system wide `er.rc` defaults file is read first, then an `.er.rc` file in the user's home directory, if present, then an `.er.rc` file in the current directory. Defaults from the `.er.rc` file in your home directory override the system defaults, and defaults from the `.er.rc` file in the current directory override both home and system defaults. These files are also used by the Analyzer and the `er_print` utility, but only the settings for source and disassembly compiler commentary are used by the `er_src` utility. See "Commands That Set Defaults" on page 146 for a description of the defaults files. Commands in a defaults file other than `scc` and `dcc` are ignored by the `er_src` utility.

`-outfile` *filename*

Open the file *filename* for output of the listing. By default, or if the filename is a dash (-), output is written to `stdout`.

`-V`

Print the current release version.

◆ ◆ ◆ CHAPTER 8

Manipulating Experiments

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

This chapter covers the following topics:

- "Manipulating Experiments" on page 211
- "Labeling Experiments" on page 212
- "Other Utilities" on page 216

Manipulating Experiments

Experiments are stored in a directory that is created by the Collector. To manipulate experiments, you can use the usual UNIX commands `cp`, `mv` and `rm` and apply them to the directory. You cannot do so for experiments from releases earlier than Forte Developer 7 (Sun ONE Studio 7, Enterprise Edition for Solaris). Three utilities which behave like the UNIX commands have been provided to copy, move and delete experiments. These utilities are `er_cp`(1), `er_mv`(1) and `er_rm`(1), and are described below.

The data in the experiment includes archive files for each of the load objects used by your program. These archive files contain the absolute path of the load object and the date on which it was last modified. This information is not changed when you move or copy an experiment.

Copying Experiments With the `er_cp` Utility

Two forms of the `er_cp` command exist:

```
er_cp [-V] experiment1 experiment2
er_cp [-V] experiment-list directory
```

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

Moving Experiments With the `er_mv` Utility

Two forms of the `er_mv` command exist:

```
er_mv [-V] experiment1 experiment2
er_mv [-V] experiment-list directory
```

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

Deleting Experiments With the `er_rm` Utility

Removes a list of experiments or experiment groups. When experiment groups are removed, each experiment in the group is removed then the group file is removed.

The syntax of the `er_rm` command is as follows:

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

The `-f` option suppresses error messages and ensures successful completion, whether or not the experiments are found. The `-V` option prints the version of the `er_rm` utility. This command removes experiments created with earlier releases of the tools.

Labeling Experiments

The `er_label` command enables you to define part of an experiment and assign a name or label to it. The label captures the profiling events that occur during one or more periods of time in the experiment that you define with start time and stop time markers.

You can specify time markers as the current time, the current time plus or minus a time offset, or as an offset relative to the start time of the experiment. Any number of time intervals can specified in a label, and additional intervals can be added to a label after it is created.

The `er_label` utility expects that intervals are specified with pairs of markers: a start time followed by a stop time. The utility ignores markers that occur out of sequence, such as a stop marker specified before any start marker, a start marker that follows a previous start marker with no intervening stop marker, or a stop marker that follows a previous stop marker with no intervening start marker.

You can assign labels to experiments by running the `er_label` command at the command line or by executing it in scripts. Once you have added labels to an experiment, you can use the labels for filtering. For example, you might filter the experiment to include or exclude the profiling events in the time periods defined by the label as described in "Using Labels for Filtering" on page 115.

Note – You should not create a label name that is the same as any other keyword that can be used in filtering because it might create conflicts and unexpected results. You can use the `er_print -describe` command to see the keywords for an experiment.

`er_label` Command Syntax

The syntax of the `er_label` command is:

```
er_label -o experiment-name -n label-name -t {start|stop}[=time-specification] [-C comment
```

The options are defined as follows:

`-o` *experiment-name* is a required option that specifies the name of the experiment that you want to label. Only one experiment name can be specified, and experiment groups are not supported. The `-o` option can appear anywhere on the command line.

`-n` *label-name* is a required option that specifies the label name. The label must be alphanumeric and contain no spaces, but can be any length. If the label already exists in the experiment, the time markers and comments specified are added to the label. The `-n` option can appear anywhere on the command line.

`-C` *comment* is an optional comment about the label. The comment can be enclosed in quotation marks or not, depending on the requirements of your shell or script. You can use multiple `-C` options for a single label, and the comments will be concatenated with a space between them when the label is displayed. You can use multiple comments, for example, to provide information about each time interval in the label. You might want to include a delimiter such as a semicolon at the end of each comment when using multiple comments in a label.

`-t start|stop` =*time-specification* is a specification of the start or stop point used to define a time range within the experiment. If =*time-specification* is omitted, a marker for current time is created.

The *time-specification* can be specified in one of the following forms:

hh:mm:ss.uuu

Specifies the time relative to the beginning of the experiment where the start or stop marker should be placed. You must specify at least seconds, and can optionally specify hours, minutes, and subseconds.

The time values you specify are interpreted as follows:

nn

If you specify an integer (without colons), it is interpreted as seconds. If the value is greater than 60 the seconds are converted to mm:ss in the label. For example, `-t start=120` places a start marker at 02:00 after the beginning of the experiment.

nn.nn

If you include a decimal of any precision, it is interpreted as a fraction of a second and saved to nanosecond precision. For example, `-t start=120.3` places a start maker at 02:00.300 or 2 minutes and 300 nanoseconds after the beginning of the experiment.

nn:nn

If you specify the time using nn:nn format, it is interpreted as mm:ss, and if the value of mm is greater than 60, the time is converted to hh:mm:ss. The number you specify for ss must be between 0 and 59 or an error occurs. For example, `-t start=90:30` places a start maker at 01:30:30 or 1 hour, 30 minutes, 30 seconds after the beginning of the experiment.

nn:nn:nn

If you specify the time using nn:nn:nn format, it is interpreted as hh:mm:ss. The numbers you specify for minutes and seconds must be between 0 and 59 or an error occurs. For example, `-t stop=01:45:10` places a stop maker at 1 hour, 45 minutes and 10 seconds after the beginning of the experiment.

`@`

Specifies the current time to place a marker in the experiment at the moment when the `er_label` command is executed. The current time is set once in a single invocation of the command, so any additional markers that use the `@` are set relative to that original timestamp value.

`@+`*offset*

Specifies a time after the current timestamp, where *offset* is a time that uses the same *hh:mm:ss.uuu* rules described above. This time format places a marker at the specified time after the original timestamp. For example, `-t stop=@+180` places a stop marker at 3 minutes after the current time.

`@-`*offset*

Specifies a time before the current timestamp, where *offset* is a time that uses the same *hh:mm:ss.uuu* rules described above. This time format places a marker at the specified time before the original timestamp. For example, `-t start=@-20:00` places a start marker at 20 minutes before the current time. If the experiment has not been running for at least 20 minutes, the marker is ignored.

You can use multiple `-t` specifications in a single `er_label` command, or multiple `-t` specifications in separate commands for the same label name, but they should occur in pairs of `-t start` and `-t stop` markers.

If the `-t start` or `-t stop` option is not followed by any time specification, `=@` is assumed for the specification. You must include a time specification for one of the markers.

`er_label` Examples

EXAMPLE 8–1 Defining a label with time markers relative to the beginning of the experiment

To define a label named `snap` in the experiment `test.1.er` that covers the part of a run from 15 seconds after the start of the experiment for a duration of 10 minutes, use the following command:

```
% er_label -o test.1.er -n snap -t start=15 -t stop=10:15
```

Alternatively, you can specify the markers for the interval in separate commands:

```
% er_label -o test.1.er -n snap -t start=15
% er_label -o test.1.er -n snap -t stop=10:15
```

EXAMPLE 8–2 Defining a label with time markers relative to the current time

To define a label named `last5mins` in the experiment `test.1.er` that covers the part of a run from 5 minutes ago to the current time:

```
% er_label -o test.1.er -n last5mins -t start=@-05:00 -t stop
```

Using `er_label` in Scripts

One use of `er_label` is to support profiling a server program that is being driven by a client as an independent process or processes. In this usage model, you start the server with the `collect` command to start creating an experiment on the server. Once the server is started and ready to accept requests from a client, you can run a client script that makes requests to drive the server and runs `er_label` to label the portions of the experiment where the client requests occur.

The following sample client script produces a time label in a `test.1.er` experiment for each request run against the server. Each of the five labels created marks off the time spent processing the named request.

```
for REQ in req1 req2 req3 req4 req5
         do
```

```
          echo "========================================================="
          echo " $REQ started at ‘date‘"

          er_label -o test.1.er -n $REQ -t start=@
          run_request $REQ
          er_label -o test.1.er -n $REQ -t stop=@
          done
```

The following sample script shows an alternative usage that produces a single label named `all`, which includes all the requests.

```
for REQ in req1 req2 req3 req4 req5
          do

          echo "========================================================="
          echo " $REQ started at ‘date‘"

          er_label -o test.1.er -n all -t start=@
          run_request $REQ
          er_label -o test.1.er -n all -t stop
          done
```

Note that no time specification follows `-t stop` in the second invocation of `er_label`, so it defaults to `stop=@`.

You can create more complex scripts, and run multiple scripts simultaneously on the same node or on different nodes. If the experiment is located in shared directory accessible by all the nodes, the scripts can mark intervals in the same experiment. The labels in each script can be the same or different.

Other Utilities

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

The `er_archive` Utility

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

```
er_archive [-nqAF] experiment
er_archive -V
```

The `er_archive` utility is automatically run when an experiment completes normally, or when the Performance Analyzer or `er_print` utility is started on an experiment. The `er_archive` utility is also run automatically by `er_kernel` if a kernel profiling session is terminated by Ctrl-C or a `kill` command. It reads the list of shared objects referenced in the experiment, and constructs an archive file for each. Each output file is named with a suffix of `.archive`, and contains function and module mappings for the shared object.

If the target program terminates abnormally, the `er_archive` utility might not be run by the Collector. If you want to examine the experiment from an abnormally-terminated run on a different machine from the one on which it was recorded, you must run the `er_archive` utility on the experiment, on the machine on which the data was recorded. To ensure that the load objects are available on the machine to which the experiment is copied, use the `-A` option.

An archive file is generated for all shared objects referred to in the experiment. These archives contain the addresses, sizes and names of each object file and each function in the load object, as well as the absolute path of the load object and a time stamp for its last modification.

If the shared object cannot be found when the `er_archive` utility is run, or if it has a time stamp differing from that recorded in the experiment, or if the `er_archive` utility is run on a different machine from that on which the experiment was recorded, the archive file contains a warning. Warnings are also written to `stderr` whenever the `er_archive` utility is run manually (without the `-q` flag).

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

-n

Archive the named experiment only, not any of its descendants.

–q

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

–A

Request writing of all load objects into the experiment. This argument can be used to generate experiments that are more readily copied to a machine other than the one on which the experiment was recorded.

–F

Force writing or rewriting of archive files. This argument can be used to run `er_archive` by hand, to rewrite files that had warnings.

–V

Write version number information for the `er_archive` utility and exit.

The `er_export` Utility

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

```
er_export [-V] experiment
```

The `er_export` utility converts the raw data in an experiment into ASCII text. The format and the content of the file are subject to change, and should not be relied on for any use. This utility is intended to be used only when the Performance Analyzer cannot read an experiment; the output allows the tool developers to understand the raw data and analyze the failure. The `–V` option prints version number information.

CHAPTER 9

Kernel Profiling

This chapter describes how you can use the Oracle Solaris Studio performance tools to profile the kernel while Oracle Solaris is running a load. Kernel profiling is available if you are running Oracle Solaris Studio software on Oracle Solaris 10 or Oracle Solaris 11. Kernel profiling is *not* available on Linux systems.

This chapter covers the following topics:

Kernel Experiments

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

The `er_kernel` utility uses DTrace, a comprehensive dynamic tracing facility that is built into the Oracle Solaris operating system.

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

Setting Up Your System for Kernel Profiling

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

Normally, DTrace is restricted to user root. To run er_kernel utility as a user other than root, you must have specific privileges assigned, and be a member of group sys. To assign the necessary privileges, edit the line for your username as follows in the file /etc/user_attr:

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

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

Running the er_kernel Utility

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

To display a usage message, run the er_kernel command without arguments.

▼ Profiling the Kernel

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

   ```
   % er_kernel -p on
   ```

2. **Run whatever load you want in a separate shell.**

3. **When the load completes, terminate the er_kernel utility by typing Ctrl-C.**

4. **Load the resulting experiment, named ktest.1.er by default, into the Performance Analyzer or the er_print utility.**

   Kernel clock profiling produces two metrics: KCPU Cycles (metric name kcycles), for clock profile events recorded in the kernel founder experiment, and KUCPU Cycles (metric name kucycles) for clock profile events recorded in user process subexperiments, when the CPU is in user-mode. In the Performance Analyzer, the metrics are shown for kernel functions in the Functions tab, for callers and callees in the Callers-Callees tab, and for instructions in the Disassembly tab. The Source tab does not show data, because kernel modules, as shipped, do not usually contain file and line symbol table information (stabs).

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

You can add a -t *duration* argument, which will cause the er_kernel utility to terminate itself according to the time specified by *duration*.

The -t *duration* can be specified as a single number, with an optional m or s suffix, to indicate the time in minutes or seconds at which the experiment should be terminated. By default, the duration is in seconds. The *duration* can also be specified as two such numbers separated by a hyphen, which causes data collection to pause until the first time elapses, and at that time data collection begins. When the second time is reached, data collection terminates. If the second number is a zero, data will be collected after the initial pause until the end of the program's run. Even if the experiment is terminated, the target process is allowed to run to completion.

If no time duration or interval is specified, er_kernel will run until terminated. You can terminate it by pressing Ctrl-C (SIGINT), or by using the kill command and sending SIGINT, SIGQUIT, or SIGTERM to the er_kernel process. The er_kernel process terminates the experiment and runs er_archive (unless -A off is specified) when any of those signals is sent to the process. The er_archive utility reads the list of shared objects referenced in the experiment, and constructs an archive file for each object.

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

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

▼ Profiling Under Load

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

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

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

If *load* is a script, it should wait for any commands it spawns to terminate before exiting, or the experiment might be terminated prematurely.

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

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

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

▼ Profiling the Kernel and Load Together

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

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

```
% er_kernel collect load
```

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

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

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

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

Profiling the Kernel for Hardware Counter Overflows

The er_kernel utility can collect hardware counter overflow profiles for the kernel using the DTrace cpc provider, which is available only on systems running Oracle Solaris 11.

You can perform hardware counter overflow profiling of the kernel with the -h option for the er_kernel command as you do with the collect command. However, dataspace profiling is not supported so dataspace requests are ignored by er_kernel.

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

To display hardware counters on a machine whose processor supports hardware counter overflow profiling, run the er_kernel –h command with no additional arguments.

If the overflow mechanism on the chip allows the kernel to tell which counter overflowed, you can profile as many counters as the chip provides; otherwise, you can only specify one counter. The er_kernel –h output specifies whether you can use more than one counter by displaying a message such as "specify HW counter profiling for up to 4 HW counters."

The system hardware counter mechanism can be used by multiple processes for user profiling, but cannot be used for kernel profiling if any user process, or the cputrack utility, or another er_kernel process is using the mechanism. If another process is using hardware counters, er_kernel will report "HW counter profiling is not supported on this system."

For more information about hardware counter profiling, see "Hardware Counter Overflow Profiling Data" on page 26 and "`-h` *counter_definition_1*`...[,`*counter_definition_n*`]`" on page 60.

Also see the `er_print` man page for more information about hardware counter overflow profiling.

Profiling Kernel and User Processes

The `er_kernel` utility enables you to perform profiling of the kernel and applications. You can use the `-F` option to control whether or not application processes should be followed and have their data recorded.

When you use the `-F on` or `-F all` options, `er_kernel` records experiments on all application processes as well as the kernel. User processes that are detected while collecting an `er_kernel` experiment are followed, and a subexperiment is created for each of the followed processes.

Many subexperiments might not be recorded if you run `er_kernel` as a non-root user because unprivileged users usually cannot read anything about another user's processes.

Assuming sufficient privileges, the user process data is recorded only when the process is in user mode, and only the user call stack is recorded. The subexperiments for each followed process contain data for the `kucycles` metric. The subexperiments are named using the format _*process-name*`_PID_`*process-pid*`.1.er`. For example an experiment run on a `sshd` process might be named `_sshd_PID_1264.1.er`.

To follow only some user processes, you can specify a regular expression using `-F =`*regexp* to record experiments on processes whose name or PID matches the regular expression.

For example, `er_kernel -F =synprog` follows processes of a program called `synprog`.

See the `regexp`(5) man page for information about regular expressions.

The `-F off` option is set by default so that `er_kernel` does not perform user process profiling.

Note – The `-F` option of `er_kernel` is different from the `-F` option of `collect`. The `collect –F` command is used to follow only processes that are created by the target specified in the command line, while `er_kernel –F` is used to follow any or all processes currently running on the system.

Analyzing a Kernel Profile

The kernel founder experiment contains data for the `kcycles` metric. When the CPU is in system-mode the kernel call stacks are recorded. When the CPU is idle a single-frame call stack for the artificial function `<IDLE>` is recorded. When the CPU is in user-mode, a single-frame call stack attributed to the artificial function <*process-name*`_PID_`*process-pid*> is recorded. In the kernel experiment, no call stack information on the user processes is recorded.

The artificial function `<INCONSISTENT_PID>` in the kernel founder experiment indicates where DTrace events were delivered with inconsistent process IDs for unknown reasons.

If `-F` is used to specify following user processes, the subexperiments for each followed process will contain data for the `kucycles`metric. User-level call stacks are recorded for all clock profile events where that process was running in user mode.

You can use context filters in the Processes tab and the Timeline tab to filter down to the PIDs you are interested in.

Index